Bachelor Thesis

Developing a Security Analysis Tool for OpenID-based Single Sign-On Systems

Ruhr-Universität Bochum

Christian Mainka

Matr.-Nr: 108007212667

5. November 2013

Lehrstuhl für Netz- und Datensicherheit
Ruhr-Universität Bochum
Universitätsstr. 150
D-44789 Bochum

Adviser: Dipl-Ing. Vladislav Mladenov
Lehrstuhl für Netz- und Datensicherheit, Ruhr-Universität Bochum

Supervision: Prof. Dr.-Ing. Jörg Schwenk
Lehrstuhl für Netz- und Datensicherheit, Ruhr-Universität Bochum

Abstract

OpenID is an open standard which can be used for Single Sign-On. It is widely used in a lot of Service Providers (SPs) like WordPress, ownCloud, OpenStreetMap, Drupal,.... However, the current security analysis mostly concentrate on the scenario of malicious clients and network attackers. This thesis will introduce *OpenID Attacker*, an open source malicious OpenID Identity Provider (IdP), which is able to manipulate arbitrary messages of the OpenID specification. It can generate OpenID login tokens for arbitrary user identities, which can lead to serious security flaws if the SP processes them improperly.

Contents

1 Introduction

Nowadays, the Internet takes a more and more important role for private users as well as for business users. While at the end of the last millennium, the Internet was mainly used as a *connecting network for fat-clients* like email clients or PC games, the browser got more and more into the focus of the last years. Several applications like office suites or navigation software are now provided as web applications, e.g. Google Docs (17) or OpenStreetMap (12). Even the area of online games has moved into the browser (42, 45).

This application shifting from fat-clients into the browser leads to an increased number of web application accounts that must be managed by a user. Years ago, the user had just to start his game or office application and could begin to use it. In today's online applications, he has to use his browser to login into a Service Provider (SP) and authenticate himself first. The result of this is a rising number of online accounts. In addition, the user can (1) choose to reuse a password for multiple SPs, (2) choose a weak but unique password, or (3) somehow remember *secure* passwords.

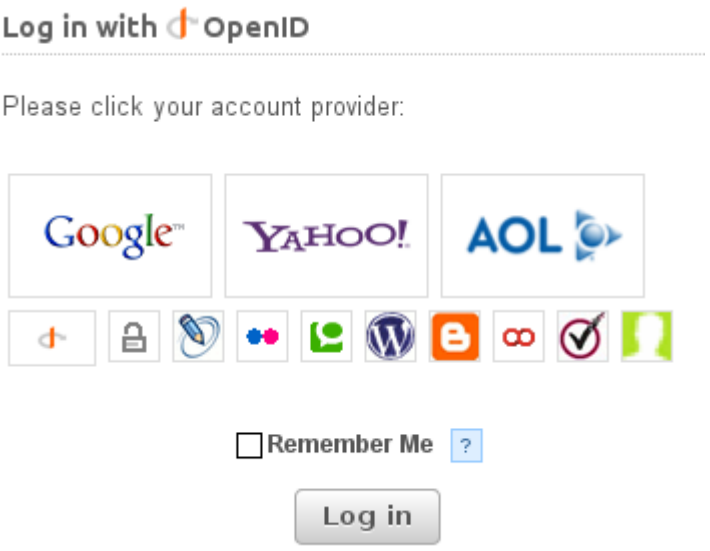

Figure 1: Using Single Sign-On to login into a SP.

To counter this problem, Single Sign-On can be used. Single Sign-On is a mechanism which separates the login process, i.e. the user's authentication, from the actual service. A so-called Identity Provider (IdP) is used to authenticate one user at multiple SPs. Thus, the user has only to authenticate once against his IdP and therefore needs only one password.

There exist different frameworks that allow Single Sign-On. Noteworthy are OAuth (7), Persona (11), SAML (27) and OpenID (36), which have been analyzed by security researchers in the last years (37, 15, 32). One of the most influencing papers related to this work is shown by Wang et al. (44). The authors developed a tool analyzing the security of Single Sign-On mechanisms by recording the traffic within a browser. The tool is online available[1] and works well for OpenID. Nevertheless, it behaves very passive. It is e.g. not possible to directly manipulate specific parts of OpenID messages, nor it is possible to manipulate the messages between the SP and the IdP, because the communication between those entities is not seen by the client/browser.

The goal of this thesis is to counter those problems by allowing a more flexible security analysis. It explains the development of an OpenID security analyis tool, which is able to manipulate arbitrary messages at any point of the OpenID protocol. The tool is named *OpenID Attacker* and is addressed to penetration testers, who want to have full flexibility on every part of the OpenID specification. OpenID Attacker is able to act as a (malicious) IdP and can therefore generate user-defined OpenID tokens, which can e.g. include arbitrary identities. If an SP improperly processes such a token, the penetration tester will be logged in with the victim's identity.

The thesis is structured as follows: The foundations needed to understand this thesis are explained in Section 2. It mostly concentrates on the OpenID specification and its protocol messages. In Section 3, the underlying attack model of OpenID Attacker is explained. This includes all features and aspects which the tool is able to manipulate. The idea of the implementation and its process, as well as the tool's model, logic and view are explained in Section 4. The tool is evaluated in Section 5 against two different widespread SPs (WordPress, ownCloud). This thesis concludes in Section 6.

[1] `http://sso-analysis.org/aaas/brm-analyzer.html`

2 Foundations

This section will give an introduction to Single Sign-On in Section 2.1. In Section 2.2, the open standard *OpenID* is explained in detail and Section 2.3 discusses related work to this thesis.

2.1 Single Sign-On

The common Internet-user makes use of many web applications. Some are for private use, others are for business use. Many web applications do not require to login for using them, e.g. the Google search engine, or a weather forecast like Yahoo Weather. However, a lot of them require a user to authenticate for accessing, e.g., his mails, images or for shopping online. In the following, web applications which offer a special service for a user are named Service Providers (SPs).

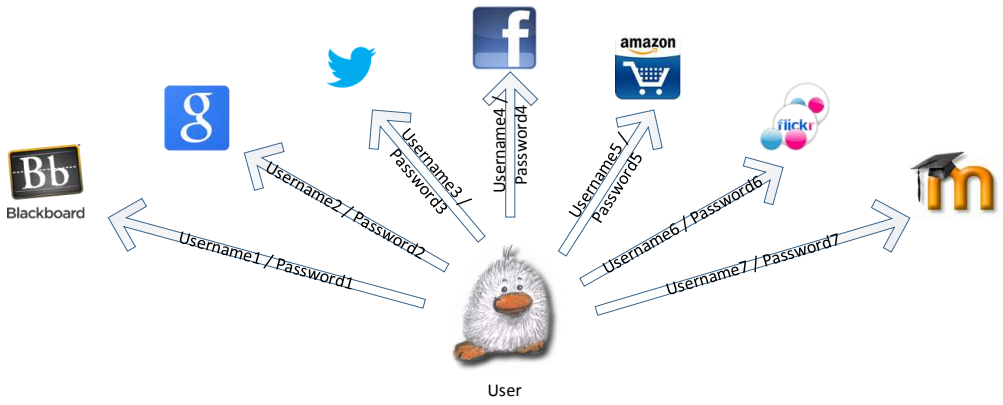

Figure 2: The login scenario for multiple SPs without Single Sign-On.

Figure 2 shows the login scenario without using Single Sign-On. The user has multiple personal accounts on each of the SPs. For accessing them, he has to remember his username and the corresponding password. The user can of course have the same username on multiple sites, but it is bad practice to reuse the same username/password combination on more than one site. There are mainly two good reasons for that:

1. The SP, or some employee at it, might be corrupted and log its users' credentials. Those credentials can then be used for trying to login into different SPs and steal the accounts.

2. Each SP must somehow store the user's credentials so that it can allow him to login. However, an attacker can try to steal those credentials by attacking the SP's database[2].

The problem of choosing unique username/password combinations is obvious: Users can easily forget them, especially if they are using strong passwords, e.g. passwords with at least 20 characters plus special characters. As a result, users tend to use weak passwords[3], which leads to another security problem.

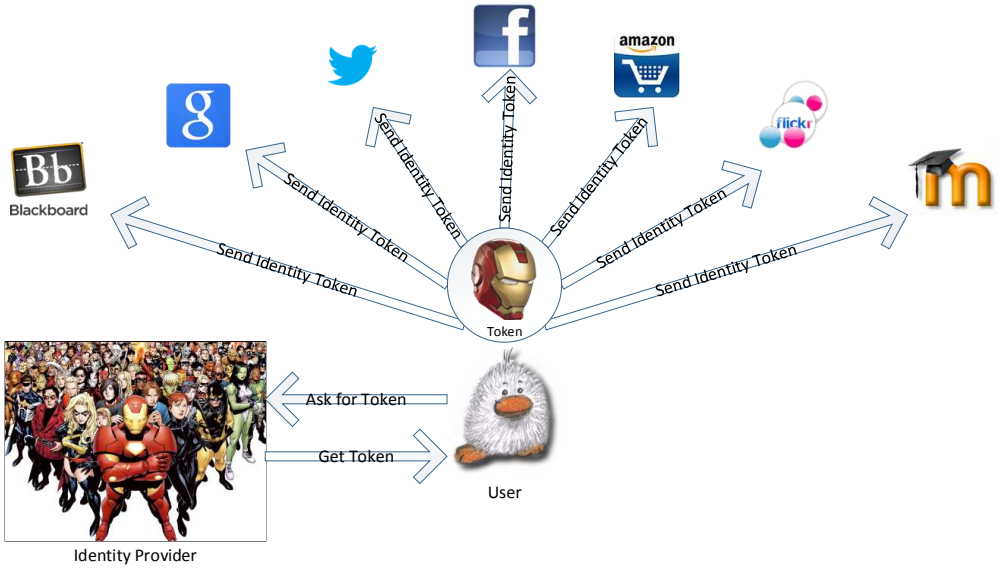

Figure 3: The login scenario for multiple SPs with Single Sign-On.

Using Single Sign-On as an authenticating mechanism can drastically reduce the number of username/password combinations. Figure 3[4]shows the idea of Single Sign-On. Instead of having one username/password combination for each SP, the user makes use of an Identity Provider (IdP). The IdP is a server administrating identities. In general, there must be a trust relationship between the SP and the IdP. Additionally, there must be a trust relationship between the user and the IdP. This chain of trust relationships (user ⇔ IdP ⇔ SP) indicates that there is an indirect relationship between the user and the SP. This is the basic

[2]An example for this was the recent hack on Adobe. See `http://blogs.adobe.com/conversations/2013/10/important-customer-security-announcement.html`

[3]See the article on `http://www.dailymail.co.uk/sciencetech/article-2223197/Revealed-The-common-passwords-used-online-year-password-STILL-tops-list.html`

[4]The Marvel superhero picture is taken from `http://desktop.freewallpaper4.me/view/original/6175/marvel-superheroes.jpg`. The Iron Man helmet is taken from `http://img.costumecraze.com/images/vendors/disguise/11677-Adult-Iron-Man-Helmet-large.jpg`.

idea of Single Sign-On. Single Sign-On separates between the service itself and the authentication of the user. In the context of Service Oriented Architectures (SOAs), Single Sign-On belongs to authentication as a service (AaaS). This allows the IdP to generate some kind of security assertion, also referred to as a security token, and to give it to the user. The user can use this token and send it to the SP. Because of the trust relationship between SP and IdP, the SP is able to verify if the token is valid in the context of the user. If this is the case, the IdP will map the identity contained within the token to a local identity used by the SP.

The advantage of Single Sign-On is, that the token itself does not contain any secret information on the user and, especially that the token does not contain the user's password.

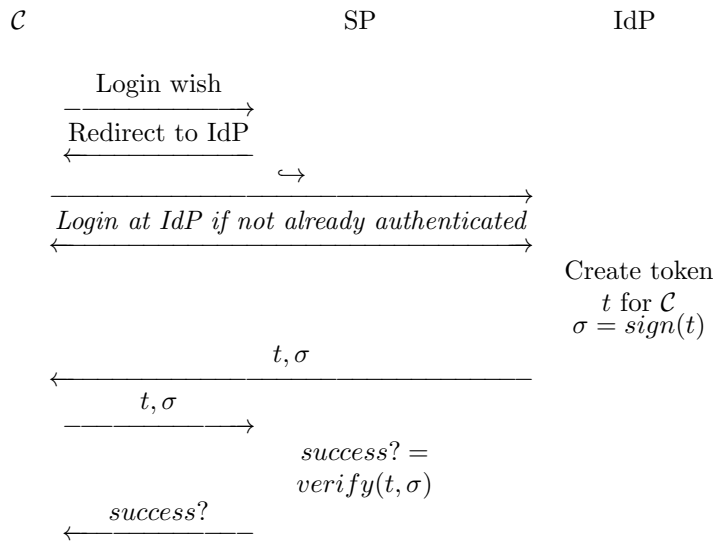

Figure 4: General overview of an exemplary Single Sign-On authentication. The protocol may also differ depending on the concrete Single Sign-On framework, but the concept is in most cases very similar.

Figure 4 gives an overview on the general workflow of an exemplary Single Sign-On authentication. The client $\mathcal{C}$ – an example user with a browser – wishes to login into some SP using a Single Sign-On mechanism. The SP then redirects $\mathcal{C}$ to the IdP. If $\mathcal{C}$ has not yet been authenticated on the IdP, he must enter his credentials. Afterwards, the IdP creates a token for $\mathcal{C}$. The token mainly contains the $\mathcal{C}$'s username and a cryptographical protection mechanism. This is commonly a signature which ensures that the token can not be manipulated and allows to determine the token's origin. Additionally, most Single Sign-On tokens also include information about their scope, e.g. for which SP the token can be

used.

There exist a lot of protocols/frameworks which allow Single Sign-On. Examples for them are OAuth (7), Persona (11), SAML (27) and OpenID (36). Although the general idea behind them remains very similar, they differ in flexibility, complexity, message flow and message format. This thesis will only look at OpenID.

2.2 OpenID

OpenID is an open standard which allows a user to authenticate against different SPs using an IdP (36). One specialty of OpenID is, that it allows decentralized Single Sign-On. Everyone can setup a custom IdP and directly use it with an arbitrary SP that supports the OpenID standard. This section will explain the OpenID standard. It will give an overview on the protocol messages and describe the different phases *discovery*, *assocation* and *token processing* in detail.

2.2.1 Protocol Overview

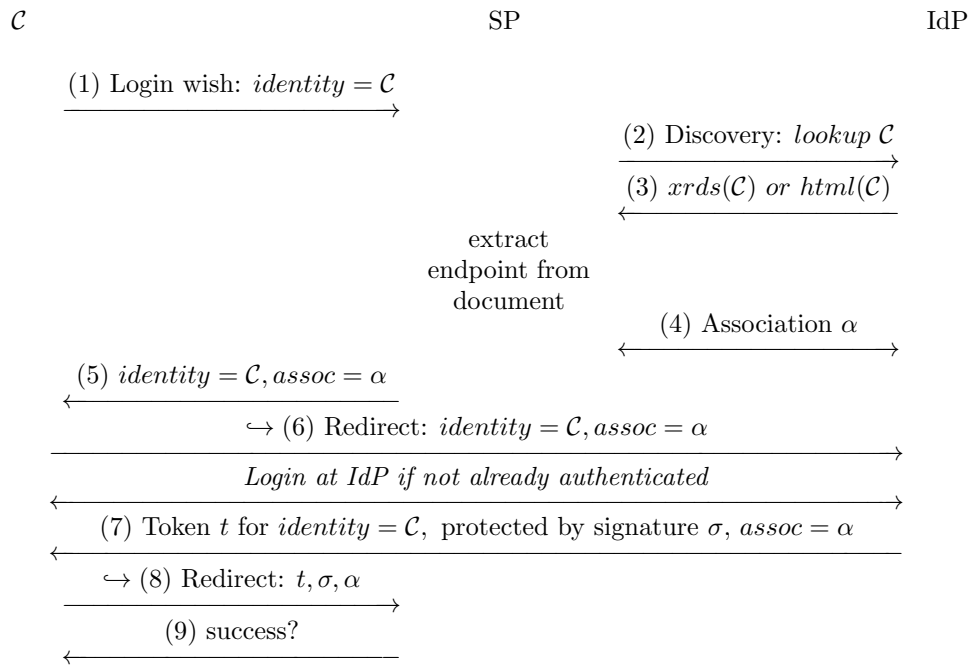

Figure 5: Simplified overview of the OpenID protocol.

A general overview on the exchanged messages of the OpenID protocol is given in Figure 5. This thesis will divide the protocol into three phases:

1. The **discovery** phase includes steps (1) to (3). In this phase, the SP collects information on the login wish of the client ($\mathcal{C}$).
2. Afterwards, the **association** phase takes part in step (4): The SP and IdP establish a shared secret, which will later be used to sign the token.
3. The last phase **processes the token** in steps (5) to (9). The token is created by the IdP and forwarded to the SP via $\mathcal{C}$'s browser.

More detailed, the OpenID login process can be described as follows:

1. $\mathcal{C}$ wishes to login into some SP. Therefore, he sends his OpenID identifier, which is a URL in most cases[5], to the SP.
2. The SP must then start the discovery on the given OpenID identifier. Therefore, he requests the document represented at the URL.
3. The according response can be either an HTML or an XRDS document. The SP extracts the included information, e.g. the IdP endpoint URL. More information on this phase is given in Section 2.2.2.
4. Using the IdP endpoint URL, the SP starts the association with the IdP. This will be explained in Section 2.2.3.
5. The SP does afterwards have all necessary information to validate an OpenID token. He responds to $\mathcal{C}$'s initial login wish (1) and sends a redirect instruction to him. This response mainly contains $\mathcal{C}$'s identity and additionally a value α which identifies the established association between SP and IdP.
6. $\mathcal{C}$ is redirected to its IdP. If not yet logged in, $\mathcal{C}$ must authenticate on the IdP.
7. The IdP creates a token for $\mathcal{C}$'s identity. The token is protected by a signature σ. In fact, this is a Hash MAC, but the OpenID specification always claimed to it as a signature, so this thesis will do this, too. σ can only be verified using the shared secret established between SP and IdP in Step (4). As the SP must be able to choose the correct key, the association value α will be included within the message. Message (7) is called the *authentication request*.
8. $\mathcal{C}$ will be redirected to the SP, whereby t, σ and α will be transmitted.
9. The SP will validate σ on t using α. If the signature is valid, the SP will map the OpenID identity to a local identity on the SP.

[5]It is also possible to use XRI identifiers (43), but this takes not part of this thesis.

2.2.2 Discovery Phase

The idea of the OpenID discovery phase is, that the SP can collect all necessary information to identify the user's OpenID identity and the corresponding IdP responsible for the association. It is specified in (36, Section 7.3). Therefore, the SP will load the OpenID URL provided by the user. The result can be an XRDS or an HTML document.

```
1  <xrds:XRDS xmlns:xrds="xri://$xrds" xmlns="xri://$xrd*($v*2.0)" ↷
       xmlns:openid="http://openid.net/xmlns/1.0">
2    <XRD version="2.0">
3      <Service priority="10">
4        <Type>http://specs.openid.net/auth/2.0/signon</Type>
5        <URI>http://my.idp.com/openid/</URI>
6        <LocalID>http://my.idp.com/openid/myname</LocalID>
7      </Service>
8      <Service priority="5">
9        <Type>http://openid.net/signon/1.0</Type>
10       <URI>http://my.idp.com/openid/</URI>
11       <openid:Delegate>http://my.idp.com/openid/myname</ ↷
             openid:Delegate>
12     </Service>
13   </XRD>
14 </xrds:XRDS>
```

Listing 1: An example XRDS document.

Listing 1 shows an example XRDS document. It is an eXtended Markup Language (XML) document which contains one or more `<Service/>` elements. Each `<Service/>` element must contain a `<Type/>` element which indicates the OpenID version and the login scenario. The `<URI/>` elements in lines 5 and 10 include the endpoint URL of the IdP. The `<Type/>` element in line 4 indicates that OpenID version 2.0 is supported by the IdP and that the `<LocalID/>` defined in line 6 should be used. Accordingly, line 9 indicates the usage of OpenID version 1.0. For 1.0, the name of the element containing the local identifier is named `< ↷ openid:Delegate>`. If the `<Type>` is set to `http://specs.openid.net/auth/2.0/server`, no `<LocalID/>` element must be provided. Instead, the SP will set `openid.identity` in its request to `http://specs.openid.net/auth/2.0/identifier_select`. This is useful if an IdP wants to provide a general XRDS document for all users. If the XRDS document contains multiple `<Service/>` elements, the IdP will use the included `priority` values, beginning with the lowest integer value.

If the returned document is an HTML document, the SP will search for the HTML `<link/>` elements shown in lines 4-6 of Listing 2.

```
1  <html>
2   <head>
3    <title>HTML Discovery</title>
4    <link rel="openid.server" href="http://my.idp.com/openid/"/>
5    <link rel="openid.delegate" href="http://my.idp.com/openid/myname ↷
         "/>
6    <link rel="openid2.provider" href="http://my.idp.com/openid/"/>
7    <link rel="openid2.local_id" href="http://my.idp.com/openid/ ↷
         myname"/>
8   </head>
9   <body/>
10 </html>
```

Listing 2: An example HTML discovery document.

The values of `openid.server` and `openid2.provider` represent the endpoint of the IdP and additionally indicate the supported OpenID version. The values of `openid.delegate` and `openid2.local_id` are optional and contain the corresponding `openid.idenitity`.

In addition, it is possible to provide the XRDS location by adding an HTTP header named `X-XRDS-Location` to the HTTP response.

2.2.3 Association Phase

OpenID uses the association phase to establish a trust relationship between SP and IdP, see (36, Section 8). As described in (36, Section 8.4), OpenID defines three association types to establish a secret MAC key: 1. *no-encryption*, 2. *DH-SHA1* and 3. *DH-SHA256*.

If *no-encryption* is used, the IdP sends the MAC key in plaintext to the SP. Therefore, *no-encryption* must be avoided unless the exchanged messages are using transport layer security.

Case 2. and 3. use Diffie-Hellman key-exchange (DHKE) to establish a common secret. This secret is then used to encrypt the MAC key chosen by the IdP using the XOR function. Afterwards, the encrypted MAC key is transmitted to the SP.

Because of the fact that the SP as well as the IdP will have to manage multiple MAC keys, both use an `assoc_handle` to store the key and later refer to it. The value of `assoc_handle` is chosen by the IdP and transmitted to the SP using the `openid.assoc_handle` parameter, see (36, Section 8.2.1).

Additionally, the IdP can define the expiration time for each association. Therefore, the IdP defines the `openid.exires_in` parameter within the association response, see (36, Section 8.2.1).

The exchanged messages and their included parameters can be seen in detail in (36, Section 8.1 and Section 8.2).

It is important to note that the association can only be established if the SP is able to store the MAC key. If this is not the case, OpenID offers a way to verify a token using direct verification (36, Section 11.4.2).

2.2.4 Token Processing Phase

The token processing phase can divided into four parts (see Figure 5).

1. Messages (5) and (6) form the *authentication request*: The SP redirects the user to the IdP and asks for a specified token, e.g. for a fixed identity $\mathcal{C}$ using the association α. See (36, Section 9) for details.
2. The IdP *generates the token* using the information of the authentication request, see (36, Section 6).
3. The token is *sent* to the user with message (7) and *redirected* to the SP in message (8). Details can be taken from (36, Section 10).
4. The SP verifies the token, see (36, Section 11).

The most interesting part for this thesis is the token generation and verification. To go into detail, the token generation consists of two parts. The first part collects all data that shall be contained within the token, e.g. the identity, the association handle, etc. In a second step, the token is signed. A pseudo code for the signature generation is shown in Listing 3.

```
1  public void sign(Token token)
2  {
3      String handle = token.getHandle();
4
5      // try shared associations first, then private
6      Association assoc = _sharedAssociations.load(handle);
7
8      if (assoc == null)
9          assoc = _privateAssociations.load(handle);
10
11     if (assoc == null) throw new ServerException(
12             "No association found for handle: " + handle);
13
14     String signedText = createSignedText(token);
15     token.setSignature(sign(signedText, assoc));
```

```
16 }
17
18 public String createSignedText(Token token)
19 {
20     StringBuffer signedText = new StringBuffer("");
21
22     String[] signedParams = token
23         .getParameterValue("openid.signed")
24         .split(",");
25
26     for (String signedParam : signedParams) {
27         signedText.append(signedParam);
28         signedText.append(':');
29         String value = getParameterValue("openid." + signedParam);
30         if (value != null) {
31             signedText.append(value);
32         }
33         signedText.append('\n');
34
35     }
36     return signedText.toString();
37 }
```

Listing 3: Pseudo code for OpenID signature generation.

In a first step, the algorithm searches for the association which belongs to the association handle parameter (`openid.assoc_handle`) within the token. If there is no such shared association (line 8), the algorithm will search for a private association connected with this handle (line 9). This is e.g. the case if the SP uses direct verification, because it is unable to store keys.

Line 14 creates the string to be signed. The algorithm for this is shown in line 18-37 and works in simple words as follows:

For each OpenID parameter contained in `openid.signed`, get the parameter value and append the string `"openid.parameterName:␣parameterValue\n"` to the resulting string. Note that the parameter names contained in `openid.signed` are not prefixed by `openid.`, therefore the prefix must be added manually (line 29).

Afterwards, this string is signed using the association, which includes the key as well as the signature algorithm (line 15).

If the SP has stored a shared association, it must perform the signature verification itself. This procedure is straightforward: The SP creates the signature value using the same algorithm as the IdP and compares the calculated value with the one stored within the token.

In the case that the SP has not stored the association, it must use *direct verification* (36, Section 11.4.2). Therefore, the SP sends the token with all OpenID parameters to the IdP. The only difference is that the `openid.mode` parameter is changed to `check_authentication`. Then, the IdP performs the signature verification and responds to the SP whether it was valid or not.

2.2.5 Extensions

The OpenID standard also allows the use of extensions (36, Section 12). The most commonly used ones are the Attribute Exchange (Ax) (35) and the Simple Registration (SReg) (34) extension. Both allow to exchange additional personal data. An example could be an email address, or the user's birthday. This information could then be added automatically to the profile on the SP, e.g. for the registration. To use an extension within a token, an OpenID namespace must be defined using the `openid.ns.prefixname` parameter. The value of this parameter points to the namespace URI of the extension. Afterwards, parameters defined by the extension can be used with `openid.prefixname.parametername`.

A detailed example can be seen in the following section.

2.2.6 Example OpenID Token

This section will show an example use-case of an OpenID token. Therefore, the authentication request and the token response during the login process on Sourceforge[6] are captured.

Listing 4 show the authentication request generated by the Sourceforge SP, which is redirected by the client to the IdP.

```
1 openid.mode: checkid_setup
2 openid.assoc_handle: myAssocHandle
3 openid.claimed_id: http://xml.nds.rub.de:8080/simpleid/www/index. ↷
    php?q=xrds/remote
4 openid.identity: http://xml.nds.rub.de:8080/simpleid/www/index.php? ↷
    q=xrds/remote
5 openid.return_to: https://sourceforge.net/account/openid_verify.php
6 openid.ns: http://specs.openid.net/auth/2.0
7 openid.ns.sreg: http://openid.net/extensions/sreg/1.1
8 openid.sreg.optional: nickname,email,fullname,country,language, ↷
    timezone
```

Listing 4: Example OpenID authentication request.

[6] `https://sourceforge.net/account/login.php`

The value of `openid.mode` is set to `checkid_setup`, because it is an authentication request. It is also possible to set it to `checkid_immediate` (36, Section 9.3), if the SP does not want the user to interact with the IdP. This is e.g. usefull for Javascript. If `checkid_immediate` is used, the SP must respond immediately that the authentication was successful or that the request cannot be processed without further user interaction. Line 2 defines the handle of the association as explained in Section 2.2.3. Line 3 and 4 define the user's identity. While OpenID 1.0 and 1.1 only define `openid.identity`, OpenID 2.0 introduces the `openid.claimed_id` parameter. There is an important difference between them:

- ▷ `openid.claimed_id` refers to the value submitted by the user within the OpenID login form.
- ▷ `openid.identity` is the value of the identity known at the IdP.

This feature can be used to save e.g. an HTML or XRDS discovery file at a custom webserver, e.g. `http://my.server.com/myid`. This document itself points to a different identity at another OpenID IdP, e.g. `http://yahoo.com`. The user can then use `http://my.server.com/myid` to login at some SP. The SP will then redirect the user to `http://yahoo.com`. If the user wants to change his OpenID provider, e.g. to `http://myopenid.com`, he just has to adjust the document a `http://my.server.com/myid`. However, to login, he can still use the same URL as before, there is no need to change anything at the SP side.

The token contains additional information about the URL the user will return to (line 5). It also defines the SReg extension as described before (line 7) and requests some attributes (line 8).

The response by the IdP is shown in Listing 5.

```
1  openid.mode: id_res
2  openid.assoc_handle: myAssocHandle
3  openid.claimed_id: http://xml.nds.rub.de:8080/simpleid/www/index.↷
       php?q=xrds/remote
4  openid.identity: http://xml.nds.rub.de:8080/simpleid/www/index.php?↷
       q=xrds/remote
5  openid.return_to: https://sourceforge.net/account/openid_verify.php
6  openid.op_endpoint: http://xml.nds.rub.de:8080/simpleid/www/
7  openid.response_nonce: 2013-10-23T12:03:35Z0
8  openid.ns: http://specs.openid.net/auth/2.0
9  openid.ns.sreg: http://openid.net/sreg/1.0
10 openid.sreg.email: Rub@nds.rub.de
11 openid.sreg.fullname: Test User
12 openid.signed: op_endpoint,claimed_id,identity,return_to,↷
       response_nonce,assoc_handle,ns.sreg,sreg.email,sreg.fullname
13 openid.sig: 3QasakorZ583wnfxZgghdplV8uXxtd7IYHJVh+gg9JU=
```

Listing 5: Example OpenID token.

The mode is set to `id_res`. The values for `openid.assoc_handle`, `openid.claimed_id`, `openid.identity` and `openid.return_to` are the same as in the authentication request (lines 2-5).

There is an additional parameter `openid.op_endpoint` which includes the URL of the IdP (line 6). The SP will compare this value to the value of the XRDS/HTML document. The IdP also adds some nonce to prevent replay attacks (line 7).

An email and fullname parameter is included using the SReg extension (line 10,11). The other optional SReg elements from the authentication request are not included.

Lastly, the `openid.signed` parameter (line 12) lists the name of the signed elements. The signature itself is stored in the `openid.sig` parameter (line 13).

2.3 Related Work

Research related to the topic of the thesis can be divided into three parts. First, there is research about protocol analysis in general. Second, analysis of Single Sign-On systems. Third, specific investigations in the field of OpenID.

The research on protocol security is, compared to the age of recent Single Sign-On protocols like OpenID, very old. Burrows et al. (6) created the BAN-logic, a formal analysis system for cryptographic protocols, in 1989. Later on, software tools for protocol verification were created and used in 1995 (25) and 1996 (24). The tools were applied e.g. on the Needham-Schroeder protocol and could automatically detect vulnerabilities.

One of the most frequently used Single Sign-On frameworks is SAML (27). SAML is based on XML and uses the XML Signature standard to protect the SAML token (8, 16). Groß (15) has analyzed the Browser/Artifact profile and identified several flaws in the SAML specification which allow connection hijacking/replay attacks, as well as Man-in-the-Middle (MitM) attacks and HTTP referrer attacks. In 2012, Somorovsky et al. (32) investigated the XML Signature validation of several SAML frameworks. By using the XML Signature Wrapping (XSW) attack technique, originally published by McIntosh and Austel (23) in 2005, they could bypass the authentication mechanism in 11 out of 14 SAML frameworks and login with arbitrary identities.

Sun and Beznosov (37) analyzed the implementation of several OAuth frameworks and found serious security flaws in Microsoft, Facebook and Google implementation. However, they do not seem to provide their analyzer tool to the public and concentrate more on XSS/CSRF and session swapping rather than on logic flaws the OAuth protocol itself.

The analysis of the OpenID protocol started with version 1.0. Tsyrklevich and Tsyrklevich (41) presented several attacks on this OpenID version at Black Hat in 2007. They identified e.g. a threat in which the endpoint URL published within the discovery phase can point to unwanted files on the local machine or can even be abused to start a Denial-of-Service (DoS) attack by enforcing the SP to download a large movie file. Additionally, they showed the attack potential of a malicious SP which redirects the user to an attacker controlled IdP, and also the potential of a malicious IdP, which can track its users. Comparable to (37), they also looked at replay and CSRF attacks. In 2008, Newman and Lingamneni (26) created a model checker for OpenID 2.0, although they simplified the OpenID protocol by removing the association phase and assuming all messages are not protected by transport layer security. Using their model checker, they could identify a session swapping vulnerability, which allows an attacker to get logged in into some SP by using the victim's account. Sovis, Kohlar, and Schwenk (33) showed in 2010 how identity information, set within OpenID messages, can be manipulated if the verification logic is improper and the authentication logic is not integrity protected. They showed a technique named parameter injection which could be used to append additional SReg and Ax extension parameters without invalidating the token's signature. However, the problem with this attack is, that the additional parameters are not processed by most SPs, because they are not requested. Thus they described the parameter forgery attack, which allows to manipulate and change the value of extension parameters which are requested and therefore processed by an SP.

Finally, Wang et al. (44) concentrated on real life Single Sign-On systems instead of a formal analysis. They developed a tool named BRM-Analyzer which handles the SP and IdP as block-boxes by analyzing only the traffic seen within the browser. Although their approach can be adopted to arbitrary Single Sign-On systems, they mainly concentrate on OpenID. However, in contrast to this thesis, the tool works rather passive – it just analyzes the browser's traffic. This restriction also includes that they do respect the threat of a malicious IdP, which the presented OpenID Attacker is.

3 Attack Model

This section will describe the used attack model for analyzing the security of OpenID. Therefore, the attacker's goal is described in Section 3.1, whereas the attacker's capabilities are described in Section 3.2 and 3.3.

3.1 Attacker's Goal

Suppose there are two Identities $\mathcal{A}$ and $\mathcal{V}$ known at a given SP. Identity $\mathcal{A}$ is owned by the attacker, therefore he can use this identity to login into the SP. The victim's identity $\mathcal{V}$ is not under the attacker's control, i.e. he does not know $\mathcal{V}$'s credentials. Thus he can not login at the SP using this identity. Given this scenario, the attacker's goal is to convince the SP that he is under the control of identity $\mathcal{V}$, i.e. to login as $\mathcal{V}$.

3.2 Attacker's Capabilities

The attacker is able to simulate a malicious IdP, notated as $\mathcal{I}dP_{\mathcal{A}}$, and a malicious client. Note that for the later implementation, it is not necessary for the client to be malicious. This is because the malicious IdP can control each of the steps performed by the client. E.g. $\mathcal{I}dP_{\mathcal{A}}$ could enforce the client to send the token to a different location by changing the redirection URL. Thus, the approach presented in this thesis does not require to write a custom OpenID client. To perform a penetration test, a valid browser will be sufficient.

Identity $\mathcal{A}$ belongs to $\mathcal{I}dP_{\mathcal{A}}$, so that $\mathcal{I}dP_{\mathcal{A}}$ can manipulate every single parameter for tokens requested by the client. Identity $\mathcal{V}$ belongs to $\mathcal{I}dP_{\mathcal{V}}$. Thus we have four participants as shown in Figure 6.

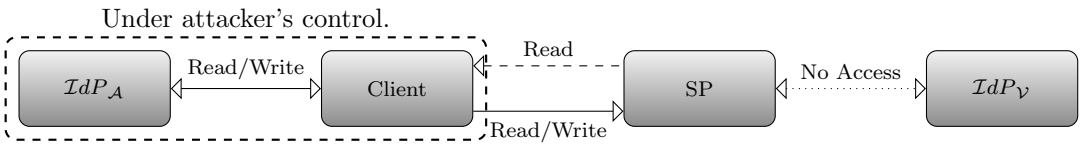

Figure 6: Overview of the attacker's capabilities.

1. The IdP of the attacker ($\mathcal{I}dP_{\mathcal{A}}$).
2. The attacker's client (Client).

3. The attacked SP (SP).
4. The IdP of the victim ($\mathcal{I}dP_{\mathcal{V}}$).

The attacker's capabilities are the following:

- ▷ The attacker can read and write/manipulate all traffic between the IdP and the client.
- ▷ The attacker can read all messages received from the SP. Additionally, he can send arbitrary, manipulated messages to the SP.
- ▷ The attacker can neither read, nor manipulate the messages exchanged between the SP and the victim's IdP.

The fact that the client can only read and not manipulate the messages received from the SP is not relying to technical issues. Theoretically, he could also manipulate the messages, but it simply does not make any sense, since the message could also be manipulated by the malicious IdP (or by the client).

An important fact of the attacker's capabilities is, that he cannot access the messages exchanged between SP and the victim's IdP. A so-called network-attacker is not in the scope of this thesis. Even the OpenID specification relies on the fact that this channel is save, see (36, Section 15.1.2). Otherwise an attacker could simply apply a MitM attack during the association phase.

3.3 Attacker's Capabilities in Detail

The previous section showed on which messages the attacker may have influence from a very abstract view. This section will now go into detail and point out which exact parts of the message are manipulatable under the given attack model. This will be explained by going through the four different OpenID phases. (1) Discovery, (2) Association, (3) Token Generation and (4) Token Verification.

3.3.1 Discovery

The discovery phase is invoked after the client has submitted its identity to the SP. Afterwards, according to the OpenID specification (36, Section 7.3), the SP has to do a discovery on this identity. The identity itself may point to an XRDS file or to an HTML file. In the latter case, the HTML can contain information, which can be resolved to an XRDS document. Therefore, there are two possibilities:

1. The HTTP response contains an `X-XRDS-Location` parameter which holds the concrete location of the XRDS document.
2. The HTML Header of the HTTP response contains a `<link/>` element holding the necessary information of the IdP endpoint URL within its attributes `rel` and `href`.
3. Optionally the local identity value.

The resulting XRDS contains the following information:

- ▷ The supported OpenID version, mainly OpenID Verion 1.0, 1.1 and 2.0.
- ▷ The endpoint URL of the IdP, which is responsible for creating tokens.
- ▷ Optionally the local identity value.

Note that the description above is just a simplification. An XRDS document can hold more than one endpoint URL and also support for multiple OpenID versions, which can be ordered by priority values. However, for a concrete penetration test, it is better to test one version after another and thus, the implemented tool will only support the use of a single endpoint.

All the parameters above are manipulatable by the attacker, and thus the developed OpenID Attacker will offer a flexible way to create XRDS documents as well as HTML documents used for the discovery.

3.3.2 Association

After the discovery phase, the SP will establish a common secret with the attacker's IdP (36, Section 8). As this is basically realized using DHKE, it is not valuable to manipulate the exchanged DHKE parameters.

Besides the established secret, the IdP can freely choose the value of the parameter `assoc_handle`. This value will be used by both parties (the SP and the IdP) to identify the key material. Therefore, it might be possible for an attacker to overwrite key material stored on the SP.

Suppose that the SP has stored the shared secret $k_{\mathcal{V}}$ with $\mathcal{I}dP_{\mathcal{V}}$ and it is saved under the value `assoc_handle=yyy`, i.e. `assoc_handle(yyy)=`$k_{\mathcal{V}}$. In the next step, the SP starts to establish a new shared secret $k_{\mathcal{A}}$ with $\mathcal{I}dP_{\mathcal{A}}$. If $\mathcal{I}dP_{\mathcal{A}}$ also uses `assoc_handle=yyy`, it might be possible that the SP overwrites $k_{\mathcal{V}}$ with $k_{\mathcal{A}}$, but still thinks that this key should be used for $\mathcal{I}dP_{\mathcal{V}}$, i.e. `assoc_handle(yyy)=`$k_{\mathcal{A}}$.

The association response also contains an expiration time, which is valuable to manipulate. E.g. by setting this to a very low value, the association may be expired very fast, forcing the SP to send the token to the IdP for direct verification instead of doing this by itself.

3.3.3 Token Generation

The token generation takes place after the client has sent a token request to the IdP (36, Section 9 and 10). For an attacker, this phase contains the greatest number of possibilities for manipulation. The attacker must be able to send arbitrary data to the SP. The data can be valid or invalid according to the OpenID specification, e.g. the signature can be removed, other parts than expected can be signed, or custom data can be added, ... The attacker is also able to change the HTTP transmission method, e.g. Get to Post and vice versa, or to send values via both methods at the same time, trying to confuse the SP logic.

More on this part will be described in Sections 4.2.4 and 4.3.1.

3.3.4 Token Direct Verification

In some cases, the SP is not able to verify the signature contained in the OpenID token itself. A reason for this might be that the SP is not able to store an association, or that the association is expired. In such cases, the token can be sent directly to the IdP, which verifies it (36, Section 11.4.2). If an attacker can trick out an SP to use the direct verification, a malicious IdP can just answer that the token is valid in every case. As a result of this, the SP will login the user with the requested identity.

4 Implementation

This Section will give an overview on the developed OpenID Attacker tool. It will start with a general overview in Section 4.1. Section 4.2 describes the tool's data model, Section 4.3 its logic and Section 4.4 the Graphical User Interface (GUI). This section concludes with a short overview on code testing for quality assurance and stability of the tool in Section 4.5.

4.1 Abstract Architecture Overview

The implemented OpenID Attacker consists in general of the three parts shown in Figure 7:

1. A **Data Model** for storing and accessing the used configurations.
2. The **Logic** or **Controller** which holds the business logic of the tool.
3. The **GUI** component which will be seen by the end-user.

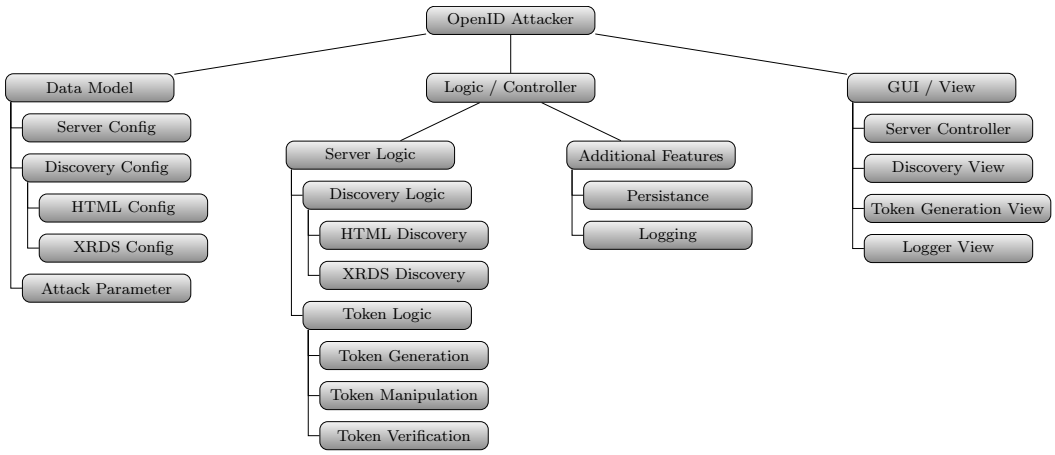

Figure 7: Abstract overview on the OpenID Attacker tool.

Each of these parts can be divided into different sub-components. The **Data Model**, which consists of the **Server Config** (e.g. the used port), the **Discovery Config** for HTML and XRDS and the **Attack Parameter** used by the **Server Logic** are described in Section 4.2. The **Logic** component can be divided into two main subcomponents: The **Server Logic** is responsible for all HTTP requests, e.g. the OpenID discovery and the token request/verfication part. **Additional Features** like **Logging** of all requests and a **Persistence** layer also belong to this component. They are described in Section 4.3. The last main component of

the OpenID Attacker is the **GUI**. It is responsible for the interaction between the user and the program. The **GUI** is described in detail in Section 4.4.

4.2 Model Description

4.2.1 Java Beans

For the implementation of the model, the OpenID Attacker tool heavily uses the concept of *Java Beans*. A Java Bean is basically a Java Class with the following properties:

1. The class is serializable. Therefore, the class itself or one of its ancestor classes must implement the Java interface *serializable*.
2. The class has a default constructor. This means that the class must offer at least a constructor without any arguments. Further constructors are also allowed.
3. The class attributes must be private and are accessible by getter and setter methods.

This concept can be realized in most cases pretty easy. However, its advantage is enormous: Most libraries, especially GUI Libraries like *Java Swing* offer a great support for Java Beans. In the case of OpenID Attacker, the GUI is developed using the Netbeans IDE (28) and the model components can simply be dropped into the *form view*. Afterwards, their attributes are easily accessible via dropdown boxes.

4.2.2 Property Change Support

A further heavily used concept in OpenID Attacker is the *property change support*. The concept is not directly related to *Java Beans*, but often combined with it. The idea of property change support is very simple: Each class attribute can be seen as a property and its value can by observed by other classes. The benefit of this is, that GUI components can attach to those classes and monitor its values easily.

Behind the scenes, the concept is very old. It is basically the *Observer Pattern* (14, Section 2). Property change support just reduced the code that is needed for its implementation. Instead of writing code to collect the observers and additionally

writing the code to notify them in each class, the property change class offers this to the developer.

Using the property change support is therefore very simple:

```
1  import java.beans.PropertyChangeListener;
2  import java.beans.PropertyChangeSupport;
3
4  public class PropertyChangeExample {
5
6      private int price = 0;
7      public static final String PROP_PRICE = "price";
8      private transient final PropertyChangeSupport ↷
           propertyChangeSupport = new PropertyChangeSupport(this);
9
10     public int getPrice() {
11         return price;
12     }
13
14     public void setPrice(int newPrice) {
15         int oldPrice = this.price;
16         this.price = newPrice;
17         propertyChangeSupport.firePropertyChange(PROP_PRICE, ↷
               oldPrice, newPrice);
18     }
19
20     public void addPropertyChangeListener(PropertyChangeListener ↷
           listener) {
21         propertyChangeSupport.addPropertyChangeListener(listener);
22     }
23
24     public void removePropertyChangeListener(PropertyChangeListener ↷
           listener) {
25         propertyChangeSupport.removePropertyChangeListener(listener ↷
               );
26     }
27 }
```

Listing 6: Example class which uses the Java property change support.

Listing 6 gives a simple example for using the property change support. It shows an example Java Class which holds a `private int price = 0;` property (line 6). The property is accessible by the corresponding getter `getPrice()` (line 10) and can be modified by the method `setPrice(int price)` (line 14).

For using the property change support, the property name is declared in line 7. The property name must always be the name of the attribute it belongs to[7]. In

[7]Of course, arbitrary names can be used. In fact, when refactoring with Netbeans using the *encapsulate field* method, the value of the propertyname is automatically the uppercase propertyname prefixed by the string `PROP_`. This leads to significant problems when trying to bind this property to a Swing component, because swing only expects the explained naming convention.

line 8, one can see the declaration of the `propertyChangeSupport` variable. The constructor just needs the observable object as an argument. Whenever the value of the variable `price` has changed, this variable can notify all observers. For adding and removing such observers, respective methods are added in lines 20-26. Lines 15-17 are showing an example of how to notify all observers. All the magic is done by the `firePropertyChange` method, which just needs the name of the property (`PROP_PRICE` plus the old and the new value of the price).

```
1  import java.beans.PropertyChangeEvent;
2  import java.beans.PropertyChangeListener;
3
4  public class PropertyChangeObserverExample implements ↷
       PropertyChangeListener {
5
6      @Override
7      public void propertyChange(PropertyChangeEvent pce) {
8          final String propertyName = pce.getPropertyName();
9          final Object newValue = pce.getNewValue();
10         final Object oldValue = pce.getOldValue();
11         final String message = String.format("%s changed from %d to ↷
               %d", propertyName, oldValue, newValue);
12         System.out.println(message);
13     }
14
15     public static void main(String[] args) {
16         PropertyChangeObserverExample observer;
17         observer = new PropertyChangeObserverExample();
18
19         PropertyChangeExample observable;
20         observable = new PropertyChangeExample();
21         observable.addPropertyChangeListener(observer);
22         observable.setPrice(10);
23     }
24 }
```

Listing 7: Example class which can observer a property.

Listing 7 gives an example how to observe the price value from Listing 6. The observer class needs to implement the `PropertyChangeListener` interface (line 4) and therefore add the method `propertyChange(PropertyChangeEvent pce)` ↷ (line 7). For observing a property, the observer class just adds itself using the provided method (line 21). If the property value is changed on the observable (line 22), the observer gets notified and the `propertyChange(..)` method (line 7) is called. In this case, the programs output will be

```
price changed from 0 to 10
```

Two additional notes:

1. Unfortunately, there does not exist any Java interface which indicates that a class is observable. This especially means that the name of the methods for adding and removing observers is not fixed. Anyway, the names defined in Listing 6 follow the common naming convention.
2. Besides observing property values, it is also possible to add *VetoableChangeSupport*. This can be used in the case, that one observer does not want the observable to change the value of a property, e.g. because it is working on it.

4.2.3 Server Config Model

The class `OpenIdServerConfiguration.class` holds the necessary properties for the basic OpenID HTTP Server, that means, it controls the server listening port and the data within the HTTP responses based an the incoming request. The class is realized as a Java Bean with property change support. It contains the following properties:

`PROP_SERVERLISTENPORT`: The port on which the server shall listen when started.

`PROP_ASSOCIATIONEXPIRATIONINSECONDS`: The value of this property defines the number of seconds, after which the association becomes expired.

`PROP_ASSOCIATIONPREFIX`: The value of the `assoc_handle` parameter set by the OpenID IdP is prefixed with this value. If it is the first `assoc_handle` using this prefix, the value of `assoc_handle` is set to the prefix itself. Otherwise, a dash followed by an ascending integer is appended.

`PROP_HTMLCONFIGURATION`: This property holds the HTML configuration of the server and is described below.

`PROP_XRDSCONFIGURATION`: This property holds the XRDS configuration of the server and is described below.

`PROP_VALIDUSER`: This property holds the configuration of the valid user. It can be basically seen as a table with key/value pairs. E.g. it holds the `openid.identity`, `openid.claimed_id` or `openid.sreg.email` data. This data will be used by the OpenID IdP when generating valid user tokens.

`PROP_ATTACKDATA`: Similar to the valid user configuration, this property holds the data used when creating malicious tokens. Although the attacking data can be defined individually in the attack parameter model, this data will be used initially to fill the attack parameters. Thus, this data is just kept for usability.

`PROP_PERFORMATTACK`: If this value is `true`, the OpenID IdP will generate malicious tokens instead of using the valid user configuration.

`PROP_INTERCEPTIDPRESPONSE`: The OpenID IdP redirects the client by using HTTP Post redirect. If this value is set to `false`, the `<form/>` will not be automatically submitted to the SP. Thus, the penetration tester can see what is exactly going to be sent to the SP without the need of any further tools like Tamper Data (1).

The model for the HTML discovery (`HtmlDiscoveryConfiguration.class`) and XRDS discovery (`XrdsDiscoveryConfiguration.class`) holds the following configuration properties:

- ▷ The `PROP_BASEURL` holds the URL under which the OpenID IdP is reachable.
- ▷ It is possible to configure the OpenID Version which the server is capable of.
- ▷ Optionally, the HTML/XRDS Document may include a local identifier URL.

4.2.4 Attack Parameter Model

One of the main capabilities of the OpenID Attacker tool is the feature of generating arbitrary OpenID tokens. To go deeper into detail, the OpenID Attacker IdP behaves exactly like any other valid IdP up to the point, where the penetration tester decides to start his attack. Then, the OpenID Attacker can generate malicious tokens. In this context, it is important to understand, that the token itself can contain a valid signature. However, *malicious token* in this context means, that OpenID Attacker is able to generate a token, which e.g. contains an identity of Yahoo or Google. A benign IdP is only allowed to generate a token for its own users, a malicious IdP is not bound to that. Of course, the token can also break any OpenID schema, like not containing a signature, signing other values as expected or simply adding arbitrary data. An additional requirement for the attacker is, that he should be able to decide, whether the parameter is trasmitted via HTTP Get or Post method.

To summarize this, an OpenID attack parameters model looks like the following:

Parameter Name	
Attack value used for signature computation?	
Valid Value	Valid HTTP Method
Attack Value	Attack HTTP Method

The valid values correspond to the identity controlled by $\mathcal{I}dP_{\mathcal{A}}$. So valid in this context means, that the $\mathcal{I}dP$ is allowed to generate tokens for this identity. On the other hand, the attack value corresponds to the victim's identity controlled by $\mathcal{I}dP_{\mathcal{V}}$.

The HTTP Method can be either *Get* (Default), *Post* or simply *Don't send*.

The boolean value *attack value used for signature computation* indicates, whether the attack value or the valid value shall be used, when computing the attack signature value. To clarify this, suppose the following:

Parameter Name	Valid Value	Attack Value
`openid.identity`	$\mathcal{I}_{\mathcal{V}}$	$\mathcal{I}_{\mathcal{A}}$
`openid.claimed_id`	$\mathcal{C}_{\mathcal{V}}$	$\mathcal{C}_{\mathcal{A}}$
`openid.signed`	`identity,claimed_id`	`identity,claimed_id`
`openid.sig`	*To be computed...*	*To be computed...*

The program will later compute the value of `openid.sig`. The question is then: which values will be used for the computation of the attack value?

For the valid value, $(\mathcal{I}_{\mathcal{V}}, \mathcal{C}_{\mathcal{V}})$ are used. However, for the attack value, it might be possible that an attacker wants to use $(\mathcal{I}_{\mathcal{A}}, \mathcal{C}_{\mathcal{V}})$ instead of $(\mathcal{I}_{\mathcal{A}}, \mathcal{C}_{\mathcal{A}})$. This can be chosen by the attacker using the property described above. Note that this could also be realized by setting the attack value to the same value as for the valid value, but this way, the attacker can distinguish between what is used for signature computation and what will be sent to the server.

To finish the attack parameter model, a class named `AttackParameterKeeper` ↷ `.class` is created which basically works as a collector for all attack parameters.

4.3 Logic Description

4.3.1 Server Logic

The basic server logic can be seen in Figure 8.

There are mainly two classes which are responsible for the logic. The first one is `CustomOpenIdProviderHandler.class`. It implements Jetty's `AbstractHandler` ↷ `.class` (9). This class's task is to parse the HTTP request and extract all OpenID parameters. Depending on the value of the `openid.mode` parameter, the request

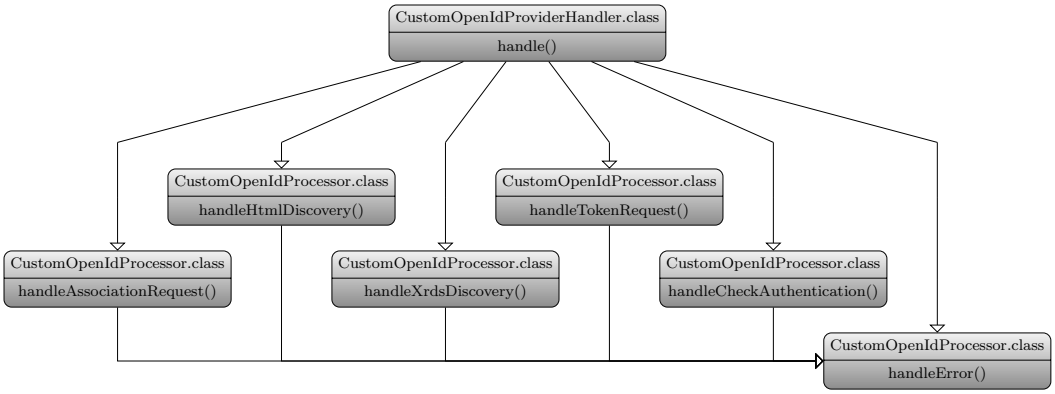

Figure 8: Server logic overview.

is delegated to a specific method of the `CustomOpenIdProcessor.class`. If an error occurs, either in the `CustomOpenIdProviderHandler.class` or in the class `CustomOpenIdProcessor.class`, it is delegated to the `handleError()` method.

The responsible methods within the `CustomOpenIdProcessor.class` are decoupled from any HTTP characteristics. They mainly need an OpenID4Java Object (`ParameterList.class`) as an argument, which holds all parsed `openid.*` parameters.

Most of the server logic uses OpenID4Java classes for simplicity (21). E.g. there is a method named `handleAssociationRequest()` which only calls the OpenID4Java method
`ServerManager.associateResponse()`. The only customization on this method is implemented deeply within the OpenID4Java structure. To set arbitrary values for the `assoc_handle` and the association's expiration time, a `CustomInMemoryServer AssociationStore.class` was designed and can be configured to the needs. An object instantiation of this class is then assigned to the `ServerManager.class` in order to fulfill its job.

In contrast to that, the discovery method `handleHtmlDiscovery()` as well as the method `handleXrdsDiscovery()` are self-made, because there is only very limited support shipped with OpenID4Java and OpenID Attacker needs a lot of flexibility for creating the discovery documents to fit the requirements mentioned in Section 3.

The `handleCheckAuthentication()` method also uses an internal OpenID4Java method. However, as the tool is meant to be an attacker, the method will always return a positive response. This is due to the fact that whenever the attacker can

convince any SP to send a token to a malicious IdP (the OpenID Attacker), the SP can be convinced that the token containing the victim's identity is valid.

The biggest custom logic is implemented within the `handleTokenRequest()` ↷ method. The method works as follows:

1. It parses the requested parameters and takes them from the valid user configuration.
2. It additionally looks for requested SReg or Ax extension parameters and also takes them from the valid user configuration.
3. It generates the valid signature using the `assoc_handle` contained within the request.
4. It generetes the attack signature. For this propose, the following steps are performed:
   - ▷ For each value which has set the flag *attack value used for signature computation* (see Section 4.2.4), it replaces the valid value with the attacker value.
     - → If this flag is not set, the attack value is replaced with the valid value. This is e.g. useful, because it updates the nonce value for the attack parameter to the correct one.
   - ▷ Finally, it computes the signature again.
5. It then decides which values will be sent to the SP. If no attack is performed, only the valid values are used. In the case of an attack, the values are sent according to their specified HTTP Methods (Get/Post/Don't Send).

Note that the signature value is always computed for both, the valid values and the attack values. This is just reasoned in usability. The user of the OpenID Attacker can see that there are two different signature values.

4.3.2 Persistence

An important feature for almost every tool, which requires a configuration, is, that it must be able to store and load its configuration. In the case of OpenID Attacker, there are lots of configuration data, e.g. the server's base URL, the configured port, the valid user data, the attack user data,... For this to work, there must be a so-called persistence layer. In Java, this could be realized using the *Serializable* interface[8]. However, there are a lot of reasons why this might not

[8] `http://docs.oracle.com/javase/6/docs/api/java/io/Serializable.html`

be the best solution:

- ▷ The serialized object does not only contain the object's properties. It contains the whole object. This means, the serialized object could also include different functionality if it is a subclass of the expected object.
- ▷ The resulted output is not human readable. It is just a byte stream.

Thus, OpenID Attacker uses a different approach to serialize its configuration: Java Architecture for XML Binding (JAXB). JAXB offers a way to serialize specified properties of an object into an XML file. The XML file has the advantage, that it can be easily read and if necessary, it can also be modified manually by using an external editor. Using JAXB is quite simple. Listing 8 shows an example for storing an object configuration into a File:

```
1  public static void saveConfigToFile(File saveFile, final ↷
       OpenIdServerConfiguration configToSave) throws ↷
       XmlPersistenceError {
2    try {
3      JAXBContext jaxbContext = JAXBContext.newInstance( ↷
         OpenIdServerConfiguration.class);
4      Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
5      jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, ↷
         true);
6      jaxbMarshaller.marshal(configToSave, saveFile);
7      LOG.info(String.format("Saved␣successfully␣config␣to␣'%s'", ↷
         saveFile.getAbsoluteFile()));
8    } catch (JAXBException ex) {
9      throw new XmlPersistenceError(String.format("Could␣not␣save␣ ↷
         config␣to␣File␣'%s'", saveFile.getAbsoluteFile()), ex);
10   }
11 }
```

Listing 8: Saving XML properties using JAXB.

The example will store an `OpenIdConfigurationObject.class` object to a specified file. Therefore, a new `JAXBContext.class` object must be created (line 3). As the object should be serialized, one has to create a `Marshaller.class` object (line 4). Using its defined `marshal()` method (line 7), the XML representation will be stored.

The class to be serialized using JAXB (in this case, the `OpenIdServerConfiguration ↷ .class`) has just to be annotated with `@XmlRootElement()`. JAXB will automatically see all getters and store its value within an XML element named to the property name. E.g. if there is a method `getName()` which returns `John`, then JAXB will create the XML Element `<name>John</name>`.

An example snippet for restoring the configuration can be seen in Listing 9.

```
1  public static void mergeConfigFileToConfigObject(final File loadFile, OpenIdServerConfiguration currentConfiguration) throws XmlPersistenceError {
2    try {
3      JAXBContext jaxbContext = JAXBContext.newInstance(OpenIdServerConfiguration.class);
4      Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
5      OpenIdServerConfiguration loadedConfig = (OpenIdServerConfiguration) jaxbUnmarshaller.unmarshal(loadFile);
6      BeanUtils.copyProperties(currentConfiguration, loadedConfig);
7      LOG.info(String.format("Loaded successfully config from '%s'", loadFile.getAbsoluteFile()));
8    } catch (InvocationTargetException | IllegalAccessException | JAXBException ex) {
9      throw new XmlPersistenceError(String.format("Could not load config from File '%s'", loadFile.getAbsoluteFile()), ex);
10   }
11 }
```

Listing 9: Merging XML properties using JAXB.

This works similar to the storing method, except for creating an unmarshaller instead of a marshaller. Note that line 6 uses the `BeanUtils.copyProperties()` method. It is used to copy the loaded configuration parameters over the current ones. In general, it would also be possible to change the current configuration object with the loaded one. However, this would cause lead to trouble because of the way the Netbeans IDE creates the GUI. It is simply not possible to rebind a data object to a GUI component (more on that in Section 4.4.2). Thus, if the object is just replaced, the GUI would not notice this and still show the content of the old object. This is prevented by merging both configuration files.

4.3.3 Logging

A very important component of the OpenID Attacker is the logging functionality. It offers a smart overview on what is going on in the tool. The idea is quite simple: Whenever the IdP gets a request, it creates one log entry which indicates the current process of the tool.

Figure 9 shows the GUI of the log viewer component. Each log entry has a specific type, e.g. *XRDS* or *Association*, the time it was created and a short summarized description. It also contains the exact HTTP request and the response generated by the OpenID Attacker.

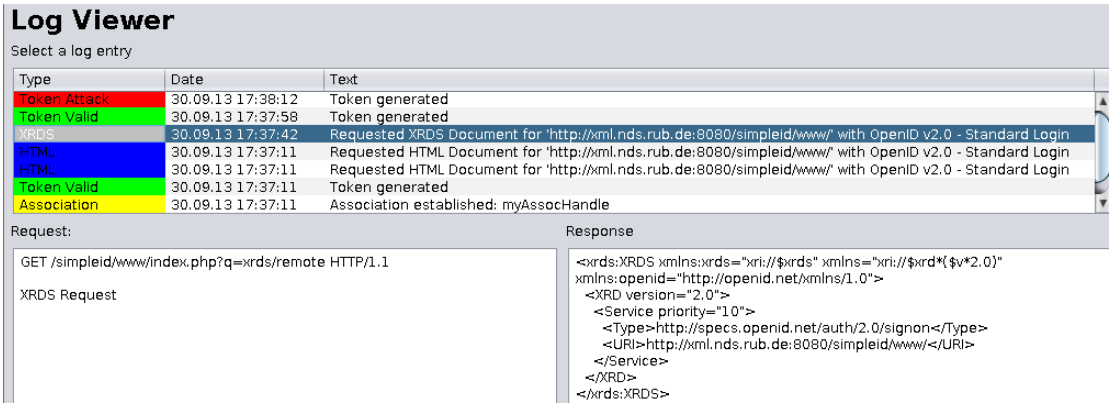

Figure 9: The OpenID Attacker log viewer.

The advantage of this log viewer is, that the penetration tester can see what is happening. E.g. he can easily find out, that the XRDS document is cached by the SP, because the SP did not request it again, or he could see that the SP does only support direct verification if it does not associate with the OpenID Attacker.

4.4 GUI Description

4.4.1 GUI Overview

The goal of this thesis is to develop a penetration test tool which allows the security analysis of OpenID. The model and the logic of the OpenID Attacker were described in the previous sections. For the usability of the OpenID Attacker, an easy-to-use GUI has been designed. The design idea is mainly taken from WS-Attacker (20): The tab-based graphical interface will guide the user from the left most tab to the right ones.

Figure 10 shows the different tabs:

Server Config: This tab gives an overview on the basic server configuration. The penetration tester can configure the server listen port, the association prefix and its expiration time. This summarizes basically the attack model for the IdP association mentioned in Section 3.3.2. Additionally, this tab offers buttons for starting and stopping the sever as well as a tabular overview on established associations and a shortened log viewer.

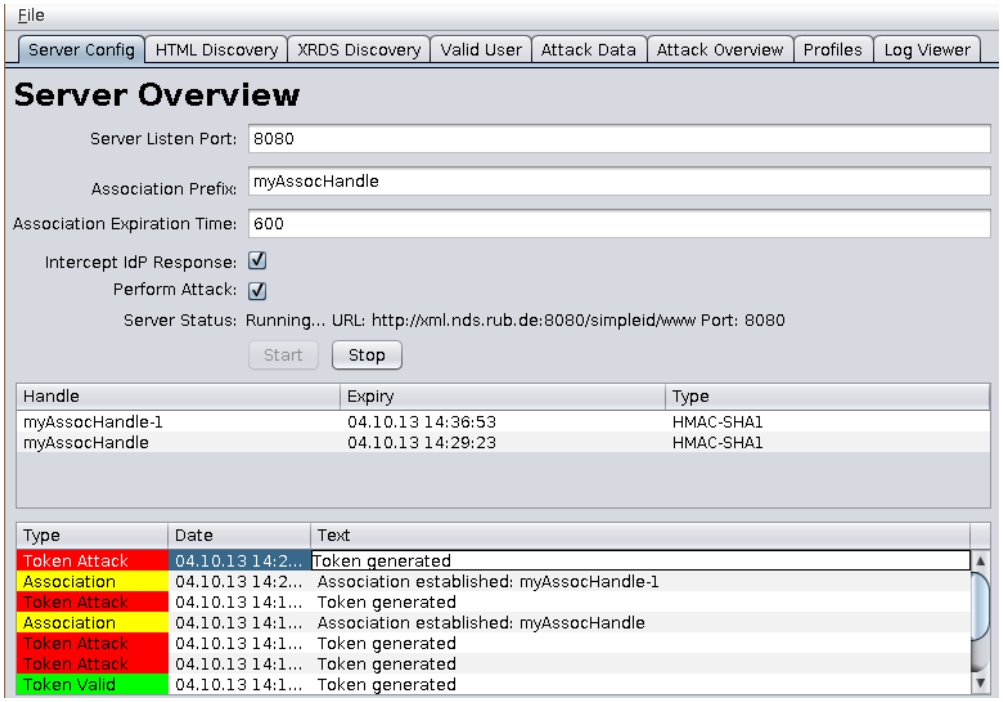

Figure 10: The tab-based GUI of OpenID Attacker.

HTML Discovery and **XRDS Discovery:** These tabs can be used to configure which data is sent during the discovery phase. The configuration options can be taken from Section 3.3.1.

Attack Overview: This tab holds the main attack configuration. It represents the attack parameter model, see Section 4.2.4. The penetration tester can configure, which parameters will be replaced with the victim values, which values are used for the signature computation and which values are send over Get/Post HTTP method to the SP.

Valid User and **Attack Data:** In these tabs, the user can setup the data used by the IdP. E.g. which `identity`, `claimed_id` or `email` is used when creating tokens. The values from the attack data represent the information corresponding to the victim's account while the valid user data belongs to the attacker's valid account.

Profiles: The profiles tab can be used to save attack configurations and restore them. It is an additional way to access the persistence layer, see Section 4.3.2. The common way is to use the *File* menu to load a configuration file. Using this tab, specific attack profiles can be saved, e.g. which values are sent using a specified method to the server. The idea of this tab is, that if the penetration tester can find a successful attack vector, he can save this specific vector and reuse it later.

Log Viewer: The logging functionality was already explained in Section 4.3.3. This tab displays the entries.

4.4.2 Beansbinding

For the creation of the GUI as described in the previous section, one concept is heavily used. It is the concept of data binding using the Java implementation named *Beansbinding* (39).

To understand this concept, one has to understand how GUIs are created with Java. Suppose there is an `Object o` which has a property named `text`. Now, a graphical input method, `GraphicalInput g`, should be created to modify this value. What the developer wants, is a graphical text field which represents `o.text`. He expected that `o.text` and `g.text` refer to the same instance of the same object. Thus, when one changes `g.text`, the model's value, `o.text` also changes. However, this is simply not possible, e.g. because `String` objects are immutable. Thus, what the developer really wants, is that the value of `g.text` and `o.text` are always synchronized. If the `o.text` changes, then `g.text` also changes and vice versa. This is called a data binding.

The only requirement for this is, that the objects must implement the property change support, see Section 4.2.2. A minimal usage example for Java Beansbinding can be seen in Listing 10:

```
1  import javax.swing.JTextField;
2  import org.jdesktop.beansbinding;
3  import org.jdesktop.beansbinding.Bindings;
4  import org.jdesktop.beansbinding.AutoBinding.UpdateStrategy;
5  import org.jdesktop.beansbinding.ELProperty;
6  import org.jdesktop.beansbinding.BeanProperty;
7  import wsattacker.sso.openid.attacker.controller.ServerController;
8
9  ...
10
11 // Controller offers method:
12 // controller.getConfig().getServerListenPort()
13 // this is the source object
14 ServerController controller = ...;
15
16 // destination object / GUI object
17 JTextField portText = new JTextField();
18
19 bindingGroup = new BindingGroup();
20
21 binding = Bindings.createAutoBinding(
22         // Synchronization strategy
23         UpdateStrategy.READ_WRITE,
```

```
24          // Source object
25          controller,
26          // Source object's property
27          ELProperty.create("${config.serverListenPort}"),
28          // Destination object
29          portText,
30          // Destination object property
31          BeanProperty.create("text")
32  );
33
34  // Add to binding group
35  bindingGroup.addBinding(binding);
36  // Activate all bindings
37  bindingGroup.bind();
```

Listing 10: Usage example of Java Beansbinding.

Note that this behavior could also be achieved by manually adding the GUI object as an observer of the model object and vice versa. Then, whenever a value is changed, the property change support can be used to adjust the other object's value. The advantage of using Beansbinding is, that it is just less code to write. No adding to the observer list is needed, no method for adjusting the value must be implemented. However, the biggest advantage is, that Beansbinding keeps track of the object. Line 25 and 27 of Listing 10 show, which object value is bound. It is `controller.getConfig().getServerListenPort()`. Now suppose the developer wants to keep track of this value without using Beansbinding. He must first keep track of the config object. If it is changed, the old property change listener must be removed and a new one must be added. The same is for the port property. This is just a lot of code to write and Beansbinding offers an elegant way to counter this problem.

To conclude, it is even smarter to use Beansbinding when using the Netbeans IDE (and maybe similar IDEs as well), because it offers a well integrated *click and bind* feature. Just right-click on a GUI element, and choose *bind*. Afterwards, the developer can simply select the property to which the GUI element should be bound, and the code is generated by the IDE.

Two additional sidemarks on Beansbinding:

1. Beansbinding already converts simple data types, e.g. `int` to `String`. It is also possible to create custom converters by extending the `Converter.` ↷ `class`[9]. This offers a lot of flexibility. E.g. in OpenID Attacker, a converter

[9] `org.jdesktop.beansbinding.Converter.class`

is used to convert the `enum` server status to a color, which indicates that the server is stopped by a red button.

2. There exists a further library named *BetterBeansbinding* (10). The project has started because the original Beansbinding has stopped being developed since March 2009. During the implementation of OpenID Attacker, it comes out, that BetterBeansbinding is much faster when binding and unbinding a lot of GUI elements on the fly. As BetterBeansbinding supports the same API as Beansbinding, it can be simply replaced by adding the BetterBeansbinding JAR to the classpath.

4.5 Code Testing

4.5.1 Unit Testing

Unit testing is a method for assuring the correctness of individual parts of source code. In contrast to *integration testing*, a unit test covers only a very small part of source code. In most cases, it covers only one specific method of a class. The idea of such a unit test is, that this method behaves as expected under specified circumstances, i.e. for a given input.

Suppose there is a method `divide(int a, int b)` which should divide an integer a by an integer b, so that the result is $\lfloor \frac{a}{b} \rfloor$. A unit test can be created by calling the method with fixed input values, e.g. $a = 7$ and $b = 2$. The test will compute `r = divide(7,2)` and check, if r is equal to the fixed value $r \stackrel{?}{=} 3$. If this is not true, the developer will be notified. Additionally, besides tests of *common method behavior*, it is essential to write tests for special cases. For the given example, there should be a test for $b = 0$. The expected behavior should be, that the message should throw an exception, indicating that a *divide-by-zero* error occurs.

The advantage of creating unit tests is, that whenever the code changes, the tests can be run to verify that at least the old functionality is given and nothing is broken. According to (22), tests are really important for the growth of a software product, because every refactoring, code cleanup or feature merge requires that at least the old functionality is not broken.

Note that creating tests is not comparable to code debugging. Whenever a software behaves strange, unwanted, or crashes occur, the developer should try to write a test for this issue. It is a bad idea to just start the debugger, understand

why the issue occurs, and then fix it. This is because the code can evolve, and maybe the issue will occur another time. Without a test that verifies the absence of a specific issue, the developer must always manually search for it.

In Java, there are two main frameworks for writing unit tests.

1. JUnit (3)
2. TestNG (4)

For JUnit, there are two commonly used versions. JUnit 3 is the legacy framework and has a lot of downsides. E.g. tests must always extend `TestCase.class`, names of test methods must have the prefix *test*, ... To counter these problems, JUnit 4 heavily uses the Java concept of Annotations[10], which was introduced with Java 5.

TestNG was developed in between JUnit 3 and JUnit 4 by a different group and uses the same Annotation concept as JUnit 4. In recent versions of JUnit 4 and TestNG, there are only very few differences[11].

4.5.2 JUnit

OpenID Attacker uses JUnit 4 tests, because the author is more familiar with this framework than with TestNG. In OpenID Attacker, every model and every logic has its own JUnit tests. To describe all these tests in detail is out of scope of this thesis. This thesis will just describe the general idea of how to write tests.

```
1 assertEquals("a", "b");
```

Listing 11: Example JUnit test using an equals assertion.

JUnit provides a set of *assert* methods. One of the most frequently used ones is `assertEquals(Object o1, Object o2)`. It compares `o1` and `o2` using its defined `equals(Object o)` method. If the method returns `false`, an error occurs. The error gives information about which value is *expected* and which value is the *actual* computed value. This leads to the first problem of the `assertEquals()` method. Which parameter refers to the expected value and which one to the actual value? The JUnit API uses the first parameter as the expected value, and the second as the actual value. For TestNG, it is vice versa, which will confuse developers

[10] `http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html`
[11] See `http://www.mkyong.com/unittest/junit-4-vs-testng-comparison/`

switching from one testing framework to the other. The syntax itself allows to change both values, thus the resulting error message will confuse a developer.

Additionally, JUnit provides the possibility to add a message `String` that will be used when a test fails. It can contain additional information why a test fails. The syntax is `assertEquals(String message, Object expected, Object actual)`. The problem of this method is, when comparing two `String` parameters, the syntax of the method accepts three `String` parameters. Again, this is confusing for people switching from TestNG to JUnit. In TestNG, the message parameter is always the last one – in JUnit it is the first one!

A further example why JUnit assert methods are not sufficient is shown in Listing 12:

```
1 assertTrue(myColorList.contains("RED");
```

Listing 12: Example for a JUnit test using a true assertion.

The test wants to verify that a given list contains an item `String "RED"`. What the developer really wants, is that the test assures, that the list contains the item. If this is not true, the fail message should show what is contained in the list instead – the developer is looking for an `assertListContainsItem(..)` method. This method does not exist in JUnit itself, it would be only possible by adding a message, which contains all list items. Thus, the developer must write this method by himself. The result will be, that there is more code to read and understand the test, and the complexity is increased.

4.5.3 Hamcrest

Hamcrest (31) is a framework for creating matchers that can be used to simplify complex unit tests and improve the test's readability. Matchers are a concept that allows to define more precisely what a test should verify. Using Hamcrest, Listing 11 can be written as follows:

```
1 assertThat("a",is(equalTo("b")));
2 // or in short
3 assertThat("a",is("b"));
```

Listing 13: A simple example of using Hamcrest matchers.

The basic method for using matchers is the `assertThat(T actual, Matcher <? super T> matcher)` method. The advantage of this method is, that it uses

asymmetric parameter types. The first parameter defines the actual object of type T. The second parameter expects a `Matcher` object. In line 3 of Listing 13, the matcher is the method `is(..)`. In line 1, there is the combination of two matchers: `is(equalTo(..))`. The concept of combining matchers allows to create very complex assertions while not loosing its readability. Listing 14 gives an example for a more complex combination of hamcrest matchers:

```
1 assertThat("OpenID", is(allOf(notNullValue(), instanceOf(String.↷
    class), equalTo("OpenID"))));
```

Listing 14: An example for the combination of multiple matchers using hamcrest.

Hamcrest itself is shipped in a very reduced manner with JUnit 4. To get the full hamcrest support, one can use the following setting in the Maven (2) *pom.xml*:

```
1  <dependencies>
2          <dependency>
3              <groupId>org.hamcrest</groupId>
4              <artifactId>hamcrest-all</artifactId>
5              <version>1.3</version>
6              <scope>test</scope>
7          </dependency>
8          <dependency>
9              <groupId>junit</groupId>
10             <artifactId>junit</artifactId>
11             <version>4.11</version>
12             <scope>test</scope>
13         </dependency>
14 </dependencies>
```

Listing 15: Including full hamcrest in a Maven project.

Note that it is very important to define the hamcrest dependency *before* defining the JUnit dependency, because Maven will only then exclude the hamcrest classes shipped with JUnit. If the dependency order is changed, one has to exclude all hamcrest classes from the JUnit JAR.

After this inclusion, hamcrest offers a large set of matchers for Beans, Collections, XML, and much more[12].

[12] `http://code.google.com/p/hamcrest/wiki/Tutorial`

5 Evaluation

This section will show the applicability of the OpenID Attacker tool. It will neither give an overview on existing attack vectors nor describe how to find new ones. It will be shown that the tool is able to login into different SPs by the example of WordPress (13) and ownCloud (30). Additionally, a step-by-step guide will show the manipulation possibilities of a penetration tester.

5.1 WordPress

WordPress is a free and open source content-management system (CMS) (13). It is based on PHP plus MySQL and according to (38), it is used by more than 20% of the top 10 million websites in September 2013. It supports OpenID login via an official plugin (40), which itself is based on the JainRain PHP OpenID library (18). The plugin supports OpenID in versions 1.0, 1.1 and 2.0. However, this section will only have a look at version 2.0 – version 1.0 will be shown in the next section.

The first step of the penetration test is to start the OpenID Attacker tool.

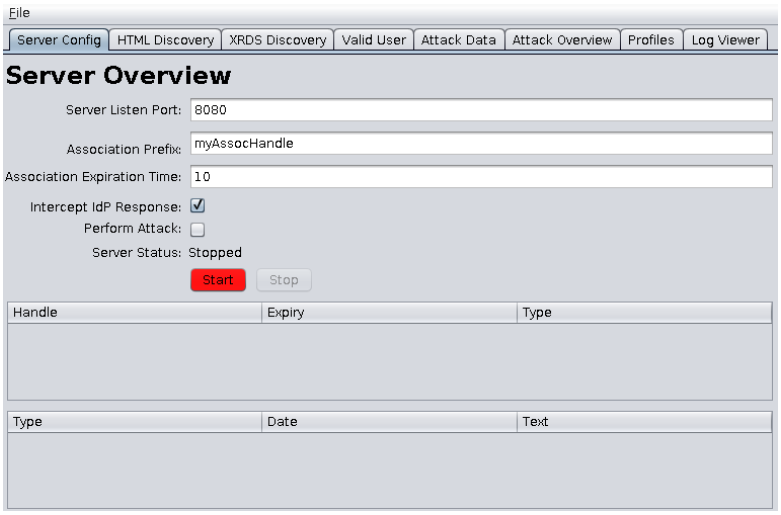

Figure 11: Server basic configuration of the OpenID Attacker.

Figure 11 shows the programm's start screen. First, one has to set up the *server listen port*, which is 8080 by default. Additionally, the penetration tester can choose a value for the *association prefix*. This value will be used when the SP will associate with the OpenID Attacker IdP. Note that this is only a prefix. If

further associations are established, the prefix is suffixed by a dash followed by a sequential integer. The penetration tester can also choose the expiration time for each established association. In this case, it is set to 10 seconds. Furthermore, the tool is configured to *intercept the IdP response* and – for the moment – to not perform any attack. Thus, the OpenID Attacker IdP behaves as a valid IdP.

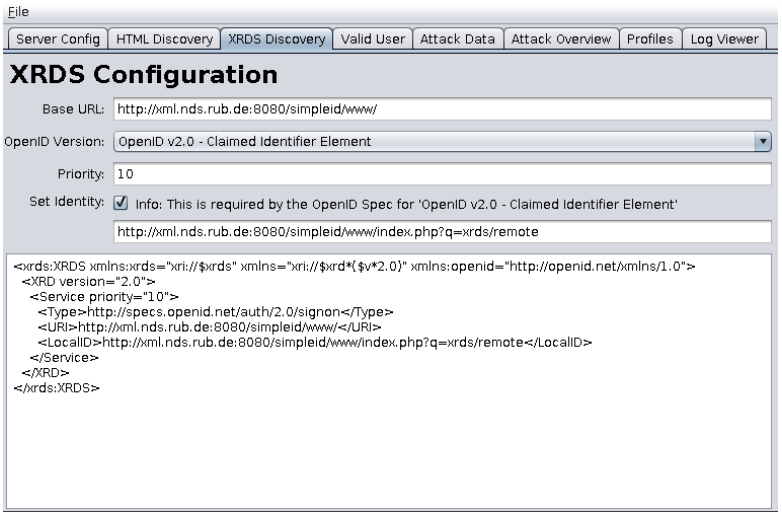

Figure 12: XRDS discovery configuration window.

The configuration for HTML discovery is skipped because it will not be used in this scenario, but in the following section. Figure 12 shows the XRDS discovery configuration. The tab can be used to configure the *base URL* of the IdP. This is the URL which will be used as an endpoint by the SP, e.g. for establishing the association. It is also possible to choose which OpenID version should be used, if an identity value should be contained and the specific value of this identity. In this scenario, OpenID version 2.0 with *claimed identifier element*[13] set to `http://xml.nds.rub.de:8080/simpleid/www/index.php?q=xrds/remote`[14] is configured. The final XML content of the XRDS document is always shown in the lower part of the tab.

[13] According to (36, Section 7.3.2.1.2)

[14] The reader should not be confused by this URL. The author of this thesis has first started to setup a SimpleID PHP OpenID IdP, and to keep compatibility, the same URL is used within the OpenID Attacker IdP. `http://simpleid.koinic.net/`

Figure 13: Valid user data configuration window.

Figure 13 shows the next step: The configuration of the *valid user data*, i.e. the data that will be used by the OpenID Attacker IdP if no attack is performed. Therefore, the penetration tester needs to configure at least the value for `identity` and `claimed_id`. Additionally, he can set up further information, like an `email` or a `fullname`. This data will be used if the SP uses Ax or SReg extensions. Note that the identity information in this tab can be different to the one in the XRDS configuration tab. The OpenID Attacker IdP will not use the information of the XRDS configuration[15].

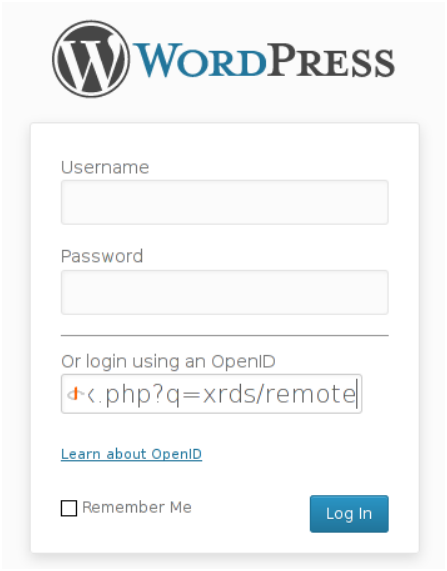

Figure 14: OpenID login form for WordPress.

Hereafter, the OpenID Attacker is configured and ready to work as a valid OpenID IdP. The server can be launched by using the *start* button, see Figure 11.

[15]This can be useful if the penetration tester wants the SP to discover different information by an XRDS discovery, than it receives within the actual token.

Using an arbitrary browser, the WordPress login page can be opened. In this scenario, a local WordPress installation is configured to use the OpenID plugin. A custom account was created and linked to the OpenID URL `http://xml.nds.rub.de:8080/simpleid/www/index.php?q=xrds/remote`. This URL is now entered into the according field, and the login button is pressed.

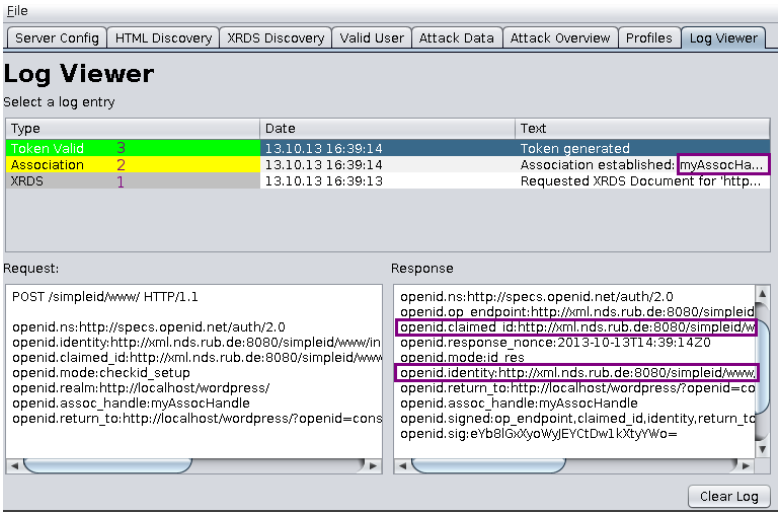

Figure 15: OpenID Attacker log viewer after sending the valid token to WordPress.

Figure 15 shows the OpenID Attacker log viewer. One can easily see the business logic of the SP:

1. First, the SP starts the discovery on the XRDS document. This corresponds to the lowermost entry within the log table.
2. Then, the SP starts the association with the OpenID Attacker IdP. The URL of the IdP is taken from the XRDS document. As it can be seen in the table, the association prefix *myAssocHandle* is used.
3. In a last step, the browser of the penetration tester is redirected to the OpenID Attacker IdP and requests a token.

The request/response pair of step 3 is shown on the lower part of the log viewer. Each other step could also be inspected in detail by selecting it from the table.

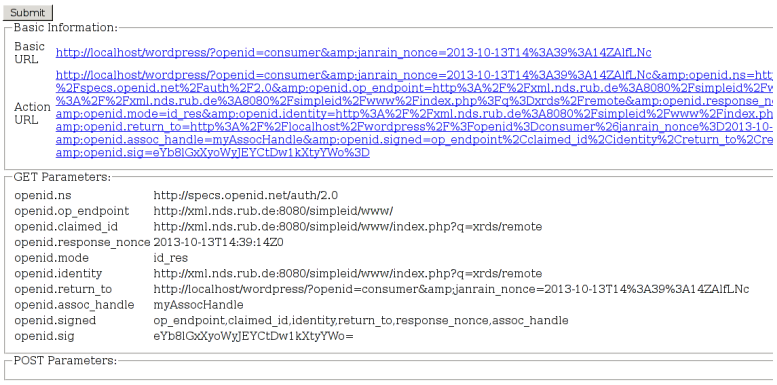

Figure 16: Token interception within the browser of the penetration tester.

Because of enabling the interception of the IdP response (see Figure 11), the token created by the OpenID Attacker IdP is not automatically forwarded to the SP. Figure 16 shows the token, and all data which should be sent to the SP, within the browser. The penetration tester can then use the submit button to submit it via HTTP Post method. Alternatively, he can click on the action URL link and send it using the HTTP Get method. Both possibilities are important, because in some cases, the SP will only accept one specific method. However, WordPress accepts both. After submitting the token, the penetration tester is successfully logged in.

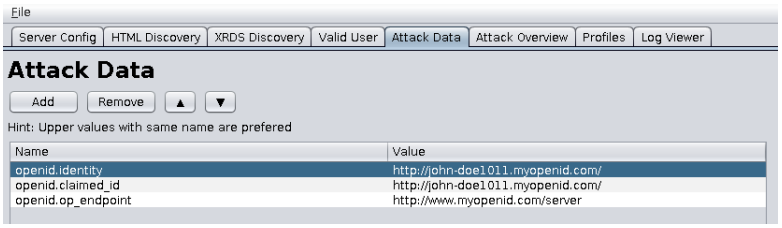

Figure 17: Attack data configuration window.

After these steps, the OpenID Attacker is configured for attacking. The penetration tester now knows that the setup was successful and that the tool can be used to login with valid tokens. For creating attack vectors, some attack data must be configured. Figure 17 shows the corresponding tab. Note that in contrast to the valid user configuration, the key/value pairs must be full parameter names, including the `openid.`-prefix[16]. All these values can be reused for different SPs as

[16]For the valid user, the tool just needs generic information. E.g. an email can be requested by Ax or SReg extensions. For comfortability, the penetration tester just needs to configure an `email`, without knowing which extension will use it. In contrast to that, for the attacking mode, the penetration tester must be able to configure this more precisely.

they will be saved on exiting the program.

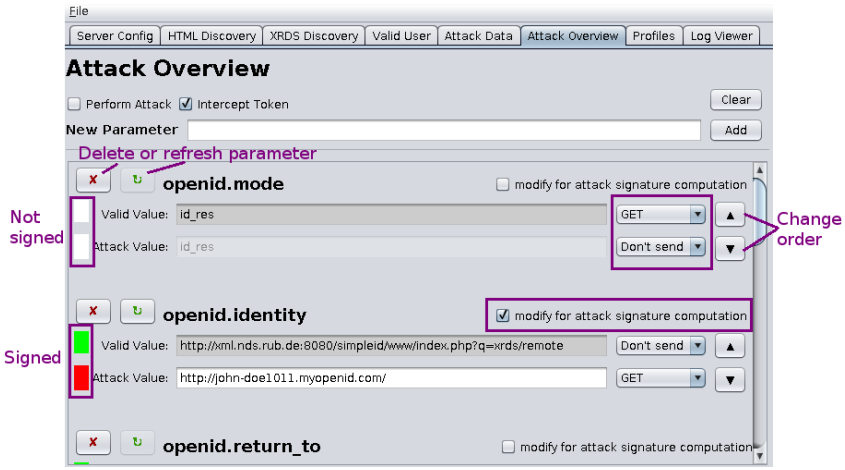

Figure 18: Attack overview and configuration window.

Figure 18 shows the concrete attack manipulation. Each OpenID parameter is shown and can be configured as described in Section 4.2.4. As an example, in the scenario, OpenID Attacker modifies the parameters `identity`, `claimed_id` and `endpoint`, whereat only the first one is shown in Figure 18. Therefore, the penetration tester has just to enable the *modify for attack signature computation* option. Afterwards, the attack value is editable and preset with the value configured in the attack data tab. It is also possible to add custom parameters, change the parameter order and its submission method (right side). On the left side, colored boxes indicate, if the values is included in the `openid.signed` parameter. A white box means that the parameter is not included, a green one indicates that it is included in the valid value of `openid.signed` and a red one correspondingly for the attack value.

Additionally, for the `openid.sig` parameter, the attack value method is changed from *Don't send* to *Get*. Note that for this special parameter, *modify for attack signature computation* is not enabled, as this would overwrite the computed signature value with a custom one.

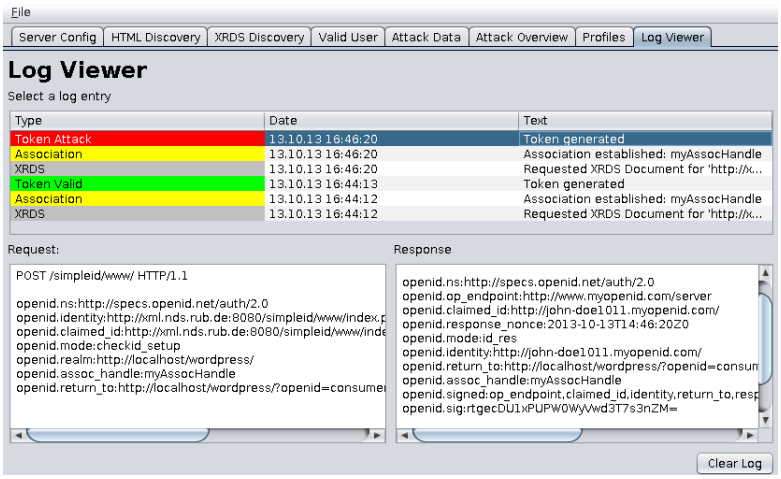

Figure 19: OpenID Attacker log viewer after sending the attack token.

After enabling the attack mode by using the *perform attack* switch, the OpenID IdP will create tokens containing the victim's identity. Figure 19 shows the log viewer after a re-login. The SP performs again the same steps as before, but the token does now include a different identity. Although the contained signature is cryptographically valid, the login attempt failed. WordPress has detected the attack.

5.2 Owncloud

OwnCloud is a free and open source web application (30). It allows its users to synchronize their personal data, like a calender or contacts, and offers the possibility to synchronize and share files between different clients. OwnCloud supports OpenID by a plugin, which is shipped with the main version and just needs to be activated. In contrast to WordPress, the ownCloud OpenID plugin only works with OpenID version 1.0 and therefore has a different working flow.

The basic configuration of this scenario is comparable to the WordPress scenario:

- A custom installation of ownCloud is deployed on the local host machine.
- A sample user account is created.
- The OpenID plugin is activated.
- The user account is linked to the OpenID URL `http://xml.nds.rub.de:8080/simpleid/www/`. Note that this is different to the WordPress scenario, because ownCloud does not support XRDS discovery.

▷ The OpenID Attacker is configured as shown in Figure 11.

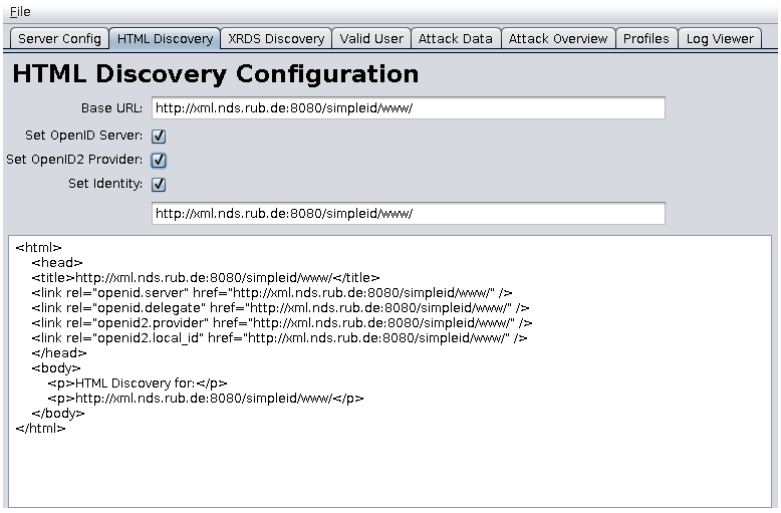

Figure 20: HTML discovery configuration window.

The main configuration difference is, that ownCloud only accepts HTML discovery. Its configuration can be seen in Figure 20. In this case, the server's base URL is the same as the identity URL. This will of course only work for a single-user IdP, but is fine for a penetration test. Additionally, OpenID version 1.0 (OpenID Server checkbox) and OpenID Version 2.0 (OpenID2 Provider checkbox) are enabled, but ownCloud will always use version 1.0.

Figure 21: OpenID login form for ownCloud.

Afterwards, the penetration tester can start the login process. OwnCloud does not provide a special OpenID login form. The user has to enter its OpenID URL just as the username. However, he additionally needs to enter an arbitrary password. This might be confusing, but ownCloud does not allow to submit a form without a password.

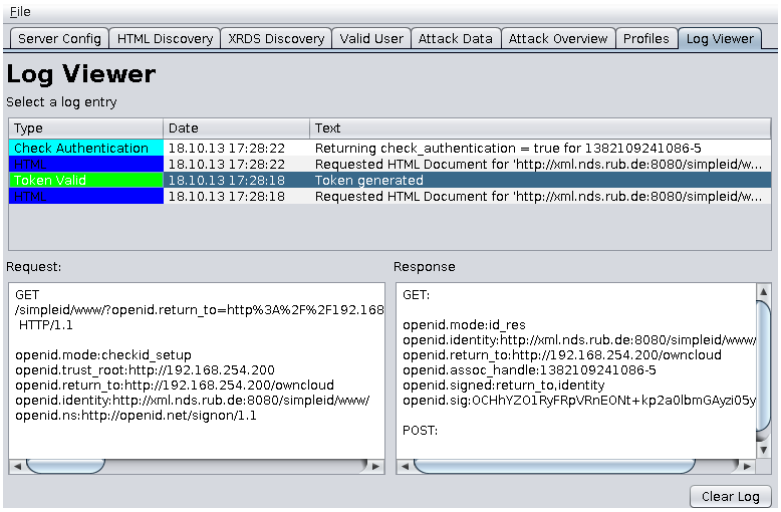

Figure 22: OpenID Attacker log viewer after sending the valid token to ownCloud.

The OpenID login workflow can is shown in Figure 22. It can be easily seen that it is totally different to Figure 15.

1. The SP starts with an HTML discovery on the given URL. He extracts the IdP URL and redirects the user's browser to it.
2. The user's browser requests a valid token of the OpenID Attacker IdP, because no attack is performed at this moment.
3. The SP receives the token, and rediscovers the contained identity.
4. As a last step, the SP verifies the token by using the OpenID provider directly, see (36, Section 11.4.1).

Some remarks about this workflow:

- ▷ The SP does not establish a shared association with the IdP. Instead, it uses the direct verification method.
- ▷ After doing the HTML dicovery, the SP does not start an XRDS discovery.

Hereafter, the account of the penetration tester is successfully logged in. The penetration tester can now start manipulating the parameters and start the penetration test.

6 Conclusion

OpenID is a widely used standard which allows Single Sign-On. However, there was no flexible tool available allowing to manipulate arbitrary messages of the OpenID specification. The promising tool of (44) by Wang et al. is only able to analyze the recorded traffic. It is not possible to do real penetration tests using the BRM-Analyzer.

This thesis closes the gap by developing OpenID Attacker, an open source OpenID security analysis tool which acts as an IdP. The idea to simulate a full-featured OpenID IdP allows penetration testers to analyze the traffic sent between client and SP as well as the traffic sent between SP and IdP. It is possible to manipulate any OpenID parameter of arbitrary exchanged messages. The tool is very flexible and allows to change the IdP behavior in every single OpenID phase, including the HTML/XRDS discovery, the association phase, and of course the token generation phase.

However, OpenID Attacker is just the beginning. It allows to create any attack vectors. In further research, the tool should be used to find and create new attack vectors. At the moment, the tool does not provide any best practices attack vectors, e.g. there are no attack vector lists comparable to those for SQL-Injection.

OpenID is used in many frameworks and libraries, and supposedly in some custom implementations of SPs out in the web. In future work, those should be examined and a heuristic for attacking them should be derived. Using OpenID Attacker, it is now possible to realize such a field study. It is now up to the penetration tester's smartness to find vulnerabilities in them.

Appendix

References

[1] Adam Judson. Tamper data. URL http://tamperdata.mozdev.org/.

[2] Apache Software Foundation. Apache maven. URL https://maven.apache.org/.

[3] Kent Beck and Erich Gamma. Junit. URL http://junit.org/.

[4] Cédric Beust and Alexandru Popescu. Testng. URL http://testng.org/.

[5] Tim Bray, Jean Paoli, Eve Maler, François Yergeau, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, November 2008. URL http://www.w3.org/TR/2008/REC-xml-20081126/.

[6] Michael Burrows, Martin Abadi, and Roger M Needham. A logic of authentication. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 426(1871):233–271, 1989.

[7] OAuth Community. Oauth community site. URL http://oauth.net/.

[8] Donald Eastlake, David Solo, and Joseph Reagle. XML-signature syntax and processing. first edition of a recommendation, W3C, February 2002. URL http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/.

[9] Eclipse Foundation. Jetty. URL http://www.eclipse.org/jetty/.

[10] Fabrizio Giudici. Betterbeansbinding. URL https://kenai.com/projects/betterbeansbinding/.

[11] Mozilla Foundation. Mozilla persona, . URL https://login.persona.org/.

[12] OpenStreetMap Foundation. Openstreetmap, . URL www.openstreetmap.org.

[13] WordPress Foundation. Wordpress, . URL http://wordpress.org/.

[14] Eric Freeman, Elisabeth Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, 1 edition, 2004. ISBN 0596007124.

[15] Thomas Groß. Security analysis of the saml single sign-on browser/artifact profile. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 298–307. IEEE, 2003.

[16] Frederick Hirsch, David Solo, Joseph Reagle, Donald Eastlake, and Thomas Roessler. XML signature syntax and processing (second edition). W3C recommendation, W3C, June 2008. URL `http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/`.

[17] Google Inc. Google docs, . URL `http://docs.google.com/`.

[18] JanRain Inc. Openid enabled, . URL `http://janrain.com/openid-enabled/`.

[19] Christian Mainka, Meiko Jensen, Juraj Somorovsky, and Jörg Schwenk. Ws-attacker. URL `http://sourceforge.net/projects/ws-attacker/`.

[20] Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. Penetration testing tool for web services security. In *SERVICES Workshop on Security and Privacy Engineering*, June 2012.

[21] Marius Scurtescu and Johnny Bufu. Openid4java. URL `http://code.google.com/p/openid4java/`.

[22] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008. ISBN 0132350882, 9780132350884.

[23] Michael McIntosh and Paula Austel. XML signature element wrapping attacks and countermeasures. In *SWS '05: Proceedings of the 2005 Workshop on Secure Web Services*, pages 20–27, New York, NY, USA, 2005. ACM Press.

[24] Catherine Meadows. Language generation and verification in the nrl protocol analyzer. In *Computer Security Foundations Workshop, 1996. Proceedings., 9th IEEE*, pages 48–61. IEEE, 1996.

[25] Jonathan K Millen. The interrogator model. In *Security and Privacy, 1995. Proceedings., 1995 IEEE Symposium on*, pages 251–260. IEEE, 1995.

[26] Ben Newman and Shivaram Lingamneni. Cs259 final project: Openid (session swapping attack), 2008. URL `http://www.stanford.edu/class/cs259/projects/cs259-final-newmanb-slingamn/report.pdf`.

[27] Organization for the Advancement of Structured Information Standards. Security assertion markup language (saml) v2.0, 2005.

[28] OWASP. Netbeans IDE 7.3.1, . URL `https://netbeans.org/downloads/7.3.1/`.

[29] OWASP. Sql injection, . URL `http://www.owasp.org/index.php/SQL_Injection`.

[30] ownCloud Inc. owncloud 5.0. URL `http://owncloud.org/`.

[31] Jon Reid, Chris Rose, Tom Denley, Steve Freeman, and Andrew Parker. hamcrest. URL `http://code.google.com/p/hamcrest/`.

[32] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. On breaking saml: Be whoever you want to be. In *Proceedings of the 21st USENIX conference on Security symposium, Security*, volume 12, pages 21–21, 2012.

[33] Pavol Sovis, Florian Kohlar, and Jörg Schwenk. Security analysis of openid. In Felix C. Freiling, editor, *Sicherheit*, volume 170 of *LNI*, pages 329–340. GI, 2010. ISBN 978-3-88579-264-2. URL `http://dblp.uni-trier.de/db/conf/sicherheit/sicherheit2010.html#SovisKS10`.

[34] specs@openid.net. Openid simple registration extension 1.0. Technical report, openid.net, June 2006. URL `http://openid.net/specs/openid-simple-registration-extension-1_0.html`.

[35] specs@openid.net. Openid attribute exchange 1.0 - final. Technical report, openid.net, December 2007. URL `http://openid.net/specs/openid-attribute-exchange-1_0.html`.

[36] specs@openid.net. Openid authentication 2.0 – final. Technical report, openid.net, December 2007. URL `https://openid.net/specs/openid-authentication-2_0.html`.

[37] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of oauth sso systems. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 378–390. ACM, 2012.

[38] W3Techs World Wide Web Technology Surveys. Usage of content management systems for websites. URL `http://w3techs.com/technologies/overview/content_management/all/`.

[39] Swinglabs. Beansbinding. URL `https://java.net/projects/beansbinding/`.

[40] DiSo Development Team. Wordpress openid plugin. URL `http://wordpress.org/plugins/openid/`.

[41] Eugene Tsyrklevich and Vlad Tsyrklevich. Single sign-on for the internet: A security story, July and August 2007. URL `https://www.blackhat.com/presentations/bh-usa-07/Tsyrklevich/Whitepaper/bh-usa-07-tsyrklevich-WP.pdf`.

[42] Ubisoft Entertainment S. A. Ubisoft. The settlers online. URL `http://www.thesettlersonline.net/`.

[43] Gabe Wachob, Drummond Reed, Les Chasen, William Tan, and Steve Churchill. Extensible resource identifier (xri) – resolution 2.0. Technical report, Organization for the Advancement of Structured Information Standards (OASIS), 2006. URL `https://www.oasis-open.org/committees/download.php/17293`.

[44] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 365–379, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4681-0. doi: 10.1109/SP.2012.30. URL `http://dx.doi.org/10.1109/SP.2012.30`.

[45] Zynga. Farmville. URL `http://www.farmville.com/`.

List of Figures

Listings

Glossary

A

AaaS authentication as a service; A SOA concept which provides an authentication service as a service. 9

Association An association between the SP and the OpenID IdP establishes a shared secret between them (36, Section 8). 10–17, 19, 22, 24, 29, 32, 36, 37, 46, 48, 53, 54

Ax Attribute Exchange; OpenID extension for exchanging identity information between endpoints (35). 16, 19, 33, 47

C

CMS content-management system; A software which helps to create an manage content collaboratory. CMSs are often web applications. 45

CSRF Cross-Site-Request-Forgery; Attack technique which tries to use web application APIs by a victim without his knowledge. 19

D

DHKE Diffie-Hellman key-exchange; Specific method for exchanging key material. 13, 23

Direct verification OpenID signature verfication performed by the OpenID IdP itself, enforced by the SP (36, Section 11.4.2). 14–16, 24, 36, 53

DoS Denial-of-Service; The unavailability of a service, which should be available. 19

G

GUI Graphical User Interface; A software component which allows a human to interact with machine by using symbols. 25, 26, 35, 36, 38–40

I

IdP Identity Provider; An entity which is responsible for creating tokens that will be used to authenticate a user against an SP. 2, 5, 6, 8–14, 16–24, 29, 30, 33, 36–38, 46–49, 51–54

J

JAXB Java Architecture for XML Binding; An interface for Java allowing to bind XML Data to Java objects. See `https://jaxb.java.net/`. 34, 35, 60

JUnit JUnit is framework for testing Java programs. 41, 42, 44, 60

M

MitM Man-in-the-Middle; An attack technique in which the attacker virtually or physically sits between to users, allowed to listen and manipulate their exchanged messages. 19, 22

MySQL MySQL is a widely used open source relational database management system. 45

O

OAuth OAuth is an open protocol which standardized API-Authoring for desktop-, web-, and mobile-applications (7). 6, 10, 19

Open source Work which are enforced by license to have source available for the public. 2, 45, 51, 54

OpenID OpenID is decentralized authentication system for web-based SPs (36). 2, 6, 7, 10–25, 29, 30, 32, 36, 45–48, 50–54

OpenID Attacker OpenID Attacker is the tool developed in this thesis for attacking the OpenID specification. 2, 6, 20, 23, 25, 26, 30, 33, 34, 36, 40, 41, 45–50, 52–54, 59

OpenID4Java OpenID4Java is library which easily allows to implement an OpenID consumber as well as an OpenID IdP (21). 32, 33

OwnCloud OwnCloud is a free and open source, PHP and MySQL based web application which allows data synchronization and cloud storage (30).. 2, 6, 45, 51–53

P

Penetration test Method for evaluating security on computer systems. 6, 21, 23, 30, 36–38, 45–50, 52–54

PHP PHP is a popular server-side scripting language and widespread in the area of web development. 45

S

SAML Security Assertion Markup Language (27). 6, 10, 18, 19

Single Sign-On Single Sign-On is a property of access controll, which allows a user to login once and request access to several systems. 2, 5–10, 18–20, 54, 59

SOA Service Oriented Architecture; Abstract model of software architecture. 9

SP Service Provider; Also known as *Relying Party*, entity that offers a service which the user wants to use. 2, 5–19, 21–24, 30, 33, 36, 38, 45–51, 53, 54, 59

SQL-Injection An attack technique which tries to inject and execute SQL statements reasoned in inadequate user input validation (29). 54

SReg Simple Registration; OpenID extension for exchanging identity information between endpoints (34). 16–19, 33, 47

T

TestNG TestNG is framework for testing Java programs. 41, 42

W

Web application A web application is an application that uses a web browser as a client.. 5, 7, 51

WordPress WordPress is a free and open source, PHP and MySQL based CMS (13).. 2, 6, 45, 48, 49, 51, 52

WS-Attacker Automatic penetration test framework (19). 37

X

XML eXtended Markup Language; textual data format to encode documents, commonly used for message exchange (5). 12, 18, 34, 35, 44, 46

XML Signature also XML Digital Signature, standard for creating signatures in XML documents (8, 16). 18, 19

XRDS eXtensible Resource Descriptor Sequence is an XML format for describing metadata as a web resource. 11–13, 17, 18, 22, 23, 36, 46–48, 52–54

XSS Cross-Site-Scripting is a web application vulnerability which tries to execute an attacker script within the context of the website owner within the user's browser. 19

XSW XML Signature Wrapping; technique for attacking signed XML documents (23). 19

Eigenständigkeitserklärung

Hiermit versichere ich, Christian Mainka (Matrikelnummer: 108007212667), dass ich die Arbeit selbständig angefertigt, außer den im Quellen- und Literaturverzeichnis sowie in den Anmerkungen genannten Hilfsmitteln keine weiteren benutzt und alle Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, unter Angabe der Quellen als Entlehnung kenntlich gemacht habe.

Ort, Datum

Christian Mainka