Do not trust me: Using malicious IdPs for analyzing and attacking Single Sign-On

Christian Mainka
Horst Görtz Institute for IT-Security
Ruhr University Bochum
Christian.Mainka@rub.de

Vladislav Mladenov
Horst Görtz Institute for IT-Security
Ruhr University Bochum
Vladislav.Mladenov@rub.de

Jörg Schwenk
Horst Görtz Institute for IT-Security
Ruhr University Bochum
Joerg.Schwenk@rub.de

Abstract—**Single Sign-On (SSO) systems simplify login procedures by using an Identity Provider (IdP) to issue authentication tokens which can be consumed by Service Providers (SPs). Traditionally, IdPs are modeled as trusted third parties. This is reasonable for centralized SSO systems like Kerberos, where each SP explicitly specifies which *single* IdP it trusts. However, a typical use case for SPs like Salesforce is that each customer is allowed to configure his own IdP. A malicious IdP should however only be able to compromise the security of those accounts on the SP for which it was configured. *If different accounts can be compromised, this must be considered as a serious attack.***

Additionally, in open systems like OpenID and OpenID Connect, the IdP for each customer account is dynamically detected in a *discovery phase*. Our research goal was to test if this phase can be used to trick a SP into using a *malicious IdP* for legitimate user accounts. Thus, by introducing a *malicious IdP* we evaluate in detail the popular and widely deployed SSO protocol OpenID. We found two novel classes of attacks, ID Spoofing (IDS) and Key Confusion (KC), on OpenID, which were not covered by previous research. *Both attack classes allow compromising the security of all accounts on a vulnerable SP, even if those accounts were not allowed to use the malicious IdP.*

As a result, we were able to compromise 12 out the most popular 17 existing OpenID implementations, including Sourceforge, Drupal, ownCloud and JIRA. We developed an open source tool *OpenID Attacker*, which enables the fully automated and fine granular testing of OpenID implementations. Our research helps to better understand the message flow in the OpenID protocol, trust assumptions in the different components of the system, and implementation issues in OpenID components. All OpenID implementations have been informed about their vulnerabilities and we supported them in fixing the issues. One year after our reports, we have evaluated 70 online websites. Some of them have upgraded their libraries and were safe from our attacks, but 26% were still vulnerable.

This is an extended version of our paper that is published at EuroS&P 2016 [1].

1. Introduction

Motivation. Password based authentication still dominates the Internet, but security problems related to passwords are obvious: Users either use weak passwords or reuse passwords between different sites, password based login is prone to simple attacks like Phishing or dictionary based attacks [2], and recently two studies on password managers [3], [4] showed all of them to be insecure. SSO schemes have been proposed to replace password based authentication, to enhance both usability and security. An non-academic overview [5] claims that 87% of U.S. customers are aware of SSO and more than half have tried it. OpenID and its successor OpenID Connect are amongst the most widespread SSO protocols. Leading companies like PayPal, Yahoo and Symantec support OpenID based authentication.

The prospect of enhanced security through the introduction of SSO schemes is combined with higher risks because SSO schemes constitute a *single point of attack*: If a weakness in an SSO scheme is detected, numerous Service Providers on the Internet may be affected simultaneously. Thus from the beginning, SSO schemes have been subject to formal security analysis [6], [7].

In view of the importance of SSO and OpenID, and of the impact a single vulnerability in a SSO system may have, we re-evaluated existing concepts for analyzing the authentication process. The question we tried to answer was: *Are the methodologies described in the literature complete in the sense that there are not other options to attack OpenID?* Unfortunately, the answer is *no* and in this paper we explain the reasons for the existing gap.

Single Sign-On. Single Sign-On (SSO) is a technique to enhance and simplify the login process on websites. Instead of managing a plethora of username/password combinations for each website, a user just needs an account at an Identity Provider (IdP) which can then be used to log in on a Service Provider (SP).

Figure 1 gives an overview of a basic SSO scenario. When a client ($\mathcal{C}$) tries to log in to a service offered by the SP, $\mathcal{C}$ sends a login request (1.) through some user agent (UA) (typically a web browser). If $\mathcal{C}$ is not yet authenticated to the SP, a token request is returned (2.). The token request contains information on the SP, the chosen IdP (e.g. the

Service Provider	Programming Language	IDS	KC	TRC	Summary: Unauthorized Access
CF OpenID	ColdFusion	⚡⚡	-	⚡	⚡⚡
DotNet OpenAuth	.NET	-	-	-	-
Drupal 6 / Drupal 7	PHP	-	⚡⚡	-	⚡⚡
dyuproject	Java	⚡⚡	-	⚡	⚡⚡
janrain	PHP, Python, Ruby	-	-	-	-
JIRA OpenID Plugin	Java	⚡⚡	-	-	⚡⚡
JOID	Java	⚡⚡	-	⚡	⚡⚡
JOpenID	Java	⚡⚡	-	-	⚡⚡
libopkele (Apache mod_auth_openid)	C++	-	-	-	-
LightOpenID	PHP	-	-	-	-
Net::OpenID::Consumer	Perl	-	-	⚡	⚡
OpenID 4 Java (WSO2)	Java	-	-	-	-
OpenID CFC	ColdFusion	⚡⚡	-	-	⚡⚡
OpenID for Node.js (everyauth, Passport)	JavaScript/NodeJS	-	-	⚡	⚡
Simple OpenID PHP Class (ownCloud 5)	PHP	⚡⚡	-	⚡	⚡⚡
Sourceforge	n.a.	⚡⚡	⚡⚡	-	⚡⚡
Zend Framework (OpenID Component)	PHP	-	⚡⚡	-	⚡⚡
Total		8	3	6	12/ 17

⚡ One account on the target is compromised. ⚡⚡ All accounts on the target are compromised.

TABLE 1: Practical evaluation results: *unauthorized access* on 12 out of 17 targets. We compromised 2 targets using the web attacker model (⚡). The other 10 targets make use of a weaker variant (⚡⚡), *without any user interaction*.

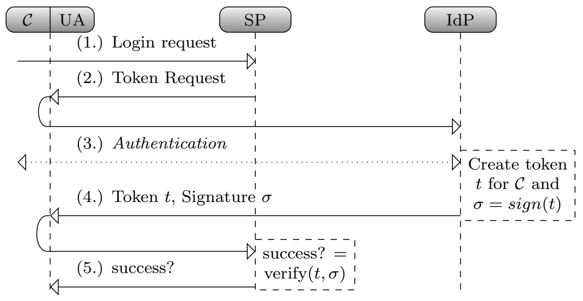

$\mathcal{C}$
UA
SP
IdP
(1.) Login request
(2.) Token Request
(3.) *Authentication*
Create token t for $\mathcal{C}$ and $\sigma = sign(t)$
(4.) Token t, Signature σ
success? = verify(t, σ)
(5.) success?

Figure 1: Single Sign-On (SSO) overview.

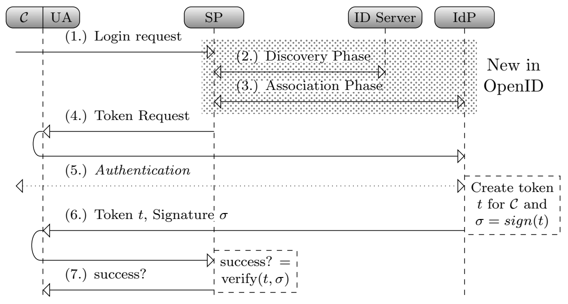


Figure 2: OpenID overview.

IdP's URL) and optionally on $\mathcal{C}$'s account name at the IdP. $\mathcal{C}$'s user agent is redirected to the IdP and forwards the token request to it. If $\mathcal{C}$ is not yet logged in at this IdP, she/he has to authenticate in Step (3.). The IdP then issues an authentication token t for $\mathcal{C}$ which is commonly protected by a cryptographic signature[1] σ. In Step (4.), t is sent back to the UA, which forwards it to the SP. Finally, the SP verifies t and, in case of successful verification, grants access to its resources in Step (5.).

OpenID. OpenID adds new components to the standard SSO information flow (Figure 2). To keep the system as open as possible, a *discovery phase* was added. Here the client enters his identity on the SP, which is used by the SP to discover the corresponding IdP. Consequentially, in the *association phase*, cryptographic keys are established on the fly. This new phase was introduced to reduce administrative overhead, especially configuring keys manually. Here an (unsigned) Diffie-Hellman key exchange is performed to establish a shared secret between SP and IdP.[2]

Are IdPs trusted third parties? Since IdPs issue signed statements about the identity of other entities on the Internet, one may be tempted to compare an IdP with a certification authority (CA), which issues X.509 certificates. While this comparison may hold for very large IdPs like Facebook, Google and Twitter, for the majority of IdPs it is not correct.

The structure of the X.509 PKI allows any CA to make trust statements on any other entity, and all consumers must trust this statement. Thus, malicious CAs, or compromised CAs (cf. the Comodo and Diginotar security breaches), compromise the security of the whole X.509 PKI.

IdPs on the other hand have a limited scope: They should only issue security tokens for users who are registered at this particular IdP, and their statements will only be trusted by certain SP, or by certain compartments of an IdP. Thus, configuring a malicious IdP for one account/compartment of a SP should not influence the security of other compartments at the same SP.

1. In many specifications, both, Message Authentication Codes (MACs) and Digital Signatures, are summarized under the term *signature*.

2. Please note that the absence of preconfigured cryptographic keys makes any OpenID connection vulnerable to *man-in-the-middle attacks*. This fact is well-known but has no relation to our attacks.

As an example, please consider Salesforce, an SaaS provider for customer relationship management software (CRM). Companies may outsource their CRM system to Salesforce, but typically want to retain control on who should be able to see and modify their customer data. Thus each company runs a separate IdP[3], and Salesforce enforces a strict separation between different company accounts/compartments (Figure 3).

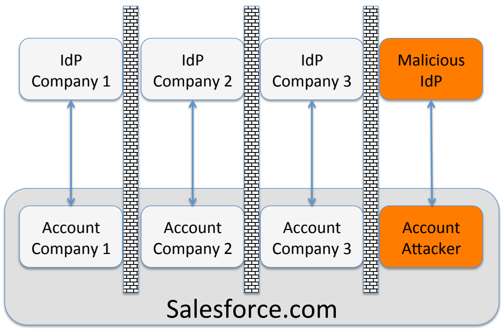


Figure 3: SPs like Salesforce enforce a strict separation between compartments. A malicious IdP can only attack its own compartment.

New SSO Attacker Paradigm. If an attacker runs a malicious IdP, we would expect that no serious harm can be done in a secure SSO system: Only the compartment on the SP configured to use this malicious IdP should be endangered. Unfortunately we were able to show that this is not the case for many OpenID implementations: here a malicious IdP can compromise *all* accounts/compartments at the SP (Figure 4).

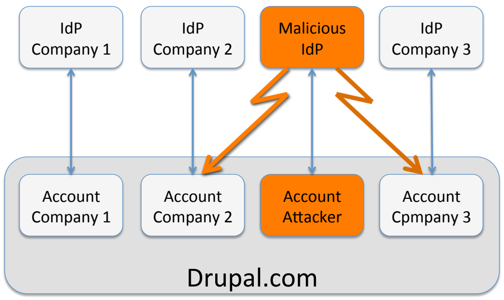


Figure 4: OpenID based SPs like Drupal allow a malicious IdP to compromise all other compartments.

Please note that the concept of a malicious IdPs compromising its own accounts is not new and trivial. In this sense only those accounts controlled by the malicious IdP can be compromised. In contrast, the presented attacks in this paper allow an attacker to log into accounts controlled by other benign IdPs.

The setup and usage of malicious IdPs is possible since protocols like OpenID introduce a novel "open" concept of delegated authentication – the user identifier is a fully qualified URL like *https://google.com/Alice*.[4] Based on this novel concept – the usage of fully qualified URLs for authentication, in OpenID the SP does not have any predefined trust relationship with a specific IdP, but only implicitly trusts an IdP because the name of this IdP was discovered through the identity/URL of a user. Thus, the concept of "compartment" for an OpenID SP resembles one subset in a partition of the user set: Each user selects exactly one IdP.

Since it is easy for anyone to run an IdP in OpenID, we extended the previously known attack methodology and systematically considered malicious IdPs. By running a malicious IdP, we enhance the attacker's capabilities: he is able to read and manipulate all messages received to and sent from the malicious IdP, even for messages that do not pass through the browser. Thus, the attacker has better control over the SSO message flow, which results in a more thorough security analysis of SSO.

Our novel approach for attacking an SP revealed two novel attack classes: ID Spoofing and Key Confusion. The effect of these attacks is devastating: We can fully compromise *all* accounts on an OpenID SP, *without any user interaction*.

What about other SSO protocols? The general concept of the malicious IdP paradigm can be applied to all current SSO protocols. However, for a successful attack the requirement is that the protocol is *open* and supports information flows similar to the discovery and association phase of OpenID. The idea of the discovery and association phase are not unique to OpenID. Modern SSO systems like OAuth [8], OpenID Connect [9], [10] and BrowserID have similar phases. As a proof-of-concept for this extensibility we manually tested OpenID Connect libraries and found vulnerabilities similar to IDS and promptly reported them[5][6]. Comprehensive research on OpenID Connect and OAuth is part of a future work. Both protocols have a different protocol structure and due to these differences (and the given page limit) this paper focuses only on the OpenID protocol, but the relevance of the attacks described herein for future generations of open SSO protocols is confirmed by the above mentioned CVEs.

Methodology. After an initial white-box analysis of the different OpenID implementations, we used our own IdP to perform realistic black-box tests against running target SP implementations. During these tests, the used IdP evolved into an automatic security testing tool **OpenID Attacker** (Section 6), allowing us now to apply all presented attack classes fully automatic on arbitrary SPs libraries. The results of both analysis phases were then verified as follows: We set up a victim account on each SP implementation, and verified in each case that we could access this account through a second (attacker-controlled) browser, running on a different PC without the victim's credentials.

3. Salesforce supports e.g. the SAML SSO system.

4. Please note that e.g. *https://google.com/Alice* is a completely different identity in comparison to *https://microsoft.com/Alice*.

5. CVE-2015-0960,CVE-2015-0959

6. http://www.connect2id.com/products/nimbus-oauth-openid-connect-sdk#thanks

The validity of all attacks found has strictly been verified in the *Web attacker model* [11]: The attacker only controls the incoming and outgoing messages to and from web applications which he controls (e.g. malicious clients, SPs and IdPs); all other network traffic, for example between the attacked SP and every honest IdP, is unknown to him. He can also freely access web applications through their interface exposed in the WWW. An attack is considered successful if the attacker gets illegitimate access to protected resource at the legitimate SP.

We do *not* assume full control over the network, for instance, we do not use the (stronger) standard cryptographic attacker model, which yields weaker results. Additionally, we do not consider phishing attacks – the attacker does not imitate a legitimate SP and we do not trick out a victim to use the attacker controlled IdP.

Results. We were able to find two novel attack classes on OpenID:

- *ID Spoofing* introduces an attacker in the role of a malicious IdP, generating tokens in the name of other (trusted) IdPs. The concept of this attack is very simple, while its impact is devastating: The malicious IdP creates a token that contains identify information belonging to an account on a different (e.g. Paypal) IdP. 8 of 17 frameworks were vulnerable to IDS. Additionally, 11 of all 70 evaluated websites, see Section 9, were susceptible against this attack.
- *Key Confusion* exploits a vulnerability in the key management implementation of the SP, resulting in the use of an untrusted key. The attacker acts as a malicious IdP and uses the association phase to establish a shared secret with the target SP. Later on, the attacker confuses the SP so that it believes to use the shared secret of another, honest IdP, while in fact, it is the one belonging the malicious IdP. 3 out of 17 implementations and 4 of the evaluated websites were vulnerable to KC.

Additionally, we adapted an attack concept known from [6] to OpenID and we are the first to evaluate this attack against existing OpenID implementations:

- *Token Recipient Confusion* introduces an attacker acting as a malicious SP. The attacker then forwards the received tokens to other SPs. The attack was successful on 6 out of 17 frameworks and 8 of the websites.

The attacks were evaluated against the 17 most popular OpenID implementations mainly taken from the official OpenID Wiki [12]. Table 1 summarizes the results: in total, we were able to compromise 12 of them. Our results show that the verification of a security token is a nontrivial task in OpenID: Dependencies between different data structures must be taken into account (e.g. association name and association key) and REST parameters must be checked with great care (Section 11).

Responsible Disclosure. All vulnerable projects have been informed and most acknowledged our findings. In case we did not receive any reaction, we filed a CVE[7]. We cooperated by proposing and providing bug fixes, which were applied in some cases[8][9][10].

Contribution. The contribution of this paper can be summarized as follows:

- We propose a novel attacker paradigm for analyzing SSO protocols: the use of a malicious IdP, which results in a more comprehensive security evaluation.
- We describe two novel attack classes on OpenID by using a malicious IdP, all strictly in the Web attacker model. These attacks provide novel insights into the problems of token verification for SPs, and of enforcing the message flow intended by the OpenID specification.
- We give a systematic overview on OpenID security and show that roughly 71% of the analyzed implementations are vulnerable, including Sourceforge, Drupal, ownCloud and JIRA.
- We develop OpenID Attacker, a free and open source malicious OpenID IdP capable of executing our novel and previous discovered attacks [13]. The tool is able to perform all attacks in a fully-automatic manner.
- In addition to our first framework analysis, we have recently evaluated 70 online websites. By applying our malicious IdP paradigm, we broke the authentication in 26% of them. Since this evaluation was after our initial library analysis, we identified websites using Drupal and the *Net:OpenID:Consumer* library that have applied our fixes. Thus, we could not break their authentication.
- As a manual proof-of-concept, we have shown that the described attack paradigms are extensible to other SSO systems like OpenID Connect.

2. Security Model

Computational Model. In OpenID, there is (in contrast to other SSO systems) an "open" trust relationship between SP and IdP: The SP trusts tokens created by any $\mathcal{I}dP$, as long as the verified client's ID belongs to the $\mathcal{I}dP$, see Figure 2. Thus, it is easy to enforce the usage of a custom IdP ($\mathcal{I}dP_{\mathcal{A}}$) into the OpenID ecosystem, which can act honestly and maliciously. The attacker can also act as malicious client or run a malicious SP ($SP_{\mathcal{A}}$).

As a result, we then control each type of communicating entities in an OpenID system. We also control (and are thus able to modify) all *types* of messages. This is especially important in the *Analyzing Mode* (cf. Section 6), where we modify certain parameters in each message type and test it against a honest instance of an SP. Please note that control over *all types of messages* should not be confused

7. CVE-2014-2048, CVE-2014-1475, CVE-2014-8249, CVE-2014-8250, CVE-2014-8251, CVE-2014-8252, CVE-2014-8253, CVE-2014-8254, CVE-2014-8265.

8. http://owncloud.org/security/advisory/?id=oC-SA-2014-002

9. http://beta.slashdot.org/journal/1083427

10. https://code.google.com/p/joid/source/detail?r=220

with control over *all messages*: we cannot access messages exchanged between honest parties (e.g. $\mathcal{I}dP$ and SP), but we can access messages exchanged with $\mathcal{I}dP_{\mathcal{A}}$ (and SP).

SSO Attacker Paradigm. The goal of the attacker is to access a protected resource to which he has no entitlement. To achieve this goal, he may use the resources of a web attacker only: he can set up his own web applications and he can lure victims to them. Furthermore, in two of three attacks described in this paper (IDS and KC) the attacker is even more powerful: by using the malicious IdP only, the attacker can break into every OpenID account on the target SP, especially into accounts that do not belong to the malicious IdP, and without any victim's interaction. Thus, there is no possibility for the victim to detect or mitigate the attacks.

In an SSO environment, the web attacker can play different roles: (1.) **Malicious client.** He can start an SSO session like any other client. Note that the attacker's identity $ID_{\mathcal{A}}$ belongs to $\mathcal{I}dP_{\mathcal{A}}$, but the victim's identity $ID_{\mathcal{V}}$ belongs to $\mathcal{I}dP_{\mathcal{V}}$. (2.) **Malicious IdP.** The malicious IdP ($\mathcal{I}dP_{\mathcal{A}}$) in our model is able to generate valid as well as malformed authentication tokens (attack tokens). (3.) **Malicious SP.** In our experiments, we never used any special properties of $SP_{\mathcal{A}}$: it is sufficient that the attacker just controls a domain (URL.$\mathcal{A}$).

In summary, due to the openness of OpenID the attacker can vary his roles in order to execute different attacks. The resources needed to execute the attacks are satisfied by the used web attacker model.

3. OpenID: Technical Background

This section gives a more detailed description of the OpenID information flow and how the authentication is applied.

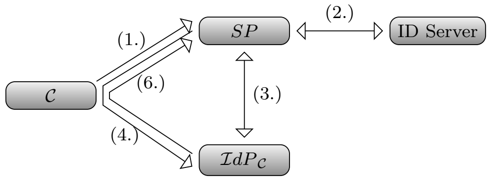


Figure 5: The OpenID protocol simplified in 5 steps. Steps are numbered according to Figure 2.

Notation. In OpenID, the identity of a client $\mathcal{C}$ is represented by a URL. Therefore, we define it as URL.ID$_{\mathcal{C}}$. Correspondingly, we define the URL of a client's IdP by URL.$\mathcal{I}dP_{\mathcal{C}}$ and for an SP, we use URL.SP.[11]

Protocol. OpenID consists of three phases: (1.) discovery, (2.) association, (3.) token processing. Figure 5 depicts a simplified login process with OpenID.

Phase 1: Discovery. The OpenID login process starts with $\mathcal{C}$ submitting his identity, called also an OpenID identifier (URL.ID$_{\mathcal{C}}$, e.g. `http://idp1.com/alice`), to the SP (Step 1.). Then, the *discovery* takes place. To discover it, the SP fetches the website URL.ID$_{\mathcal{C}}$ (`http://idp1.com/alice`) on the ID-Server (Step 2.). This website basically contains the URL of $\mathcal{C}$'s Identity Provider, namely URL.$\mathcal{I}dP_{\mathcal{C}}$ (e.g. `http://idp1.com`). Note that the separation of ID-Server and IdP offers to OpenID users the flexibility to use their own Website as an identity (e.g. `www.alice.com`) while using a public IdP (e.g. `www.google.com`)

Phase 2: Association. The *association* phase is basically a Diffie-Hellman key exchange between the SP and $\mathcal{I}dP_{\mathcal{C}}$. It is started by the SP in Step (3.) and uses the discovered value URL.$\mathcal{I}dP_{\mathcal{C}}$ (`http://idp1.com`) to determine its associate. The shared secret between SP and $\mathcal{I}dP_{\mathcal{C}}$ is later used to compute the signature for the OpenID token. The shared secret is saved on both sides by using a so-called *association* handle parameter α, which is a unique random string. α is chosen by $\mathcal{I}dP_{\mathcal{C}}$ and transmitted to the SP in the HTTP response, so that both sides use the same value to refer the shared secret.

Phase 3: Token Processing. After the association phase, the SP has all necessary information to validate an OpenID token created by $\mathcal{I}dP_{\mathcal{C}}$, so that the *token processing* phase starts. The SP responds to $\mathcal{C}$'s initial request in Step 4. This HTTP response redirects $\mathcal{C}$ to URL.$\mathcal{I}dP_{\mathcal{C}}$ (`http://idp1.com`), including the parameters from the discovery and the association phase (URL.$\mathcal{I}dP_{\mathcal{C}}, \alpha$) plus the SP's own URL (URL.SP). Then, $\mathcal{I}dP_{\mathcal{C}}$ generates a token t that contains the following parameters:

URL.ID$_{\mathcal{C}}$	The OpenID identifier used by $\mathcal{C}$, e.g. `http://idp1.com/alice`
URL.$\mathcal{I}dP$	The URL of the IdP which creates the token, e.g. `http://idp1.com`
URL.SP	The URL on which this token is going to be used, e.g. `http://sp1.net`
α	The association handle (random string) that identifies the shared secret which was used to sign the token.
σ	A signature that protects the previously mentioned parameters.

$\mathcal{I}dP_{\mathcal{C}}$ then responds with a HTTP redirect in Step 6. For this redirect, the value of URL.SP (`http://sp1.net`) is used and the token t is forwarded to its destination SP. The SP then verifies the token and $\mathcal{C}$ is logged in. This verification is described in Section 4.

4. SSO Token verification

Token verification at the SP is the most critical part within the SSO process. It consists of many steps in order to guarantee the validity of the authentication. This observation holds for SSO in general (SAML, OAuth, OpenID, OpenID Connect and BrowserID). In the following, these verification steps are discussed.

Message Parsing. Each token has a specific structure. For instance, each OpenID parameter starts with `openid.*`, and the required set of parameters must be checked by each application. At the beginning, whenever an SP receives a

11. URL.ID$_{\mathcal{C}}$ and URL.$\mathcal{I}dP_{\mathcal{C}}$ need not necessarily belong to the same domain.

message, it has to be parsed into a data object so that it can be processed further. Any error during this parsing directly affects SSO security: for instance, if some data element is present twice with different content, the second content may overwrite the first during the parsing, or vice versa. Additionally, all required parameters must be present.

Freshness. Freshness of authentication tokens is important for preventing replay attacks. It can be realized with two parameters: (1.) a nonce, which is a random value selected by the SP and/or (2.) a timestamp which defines the token's creation time or period of validity, and which is usually selected by the IdP. OpenID uses the parameter `openid.response_nonce`. It contains the creation time of the token concatenated with a random string.

Token Recipient Verification. A token t is intended for a single SP. Thus, it should be guaranteed that (1.) t can be successfully verified by a single SP only, and (2.) that t is delivered to the correct SP. OpenID uses the URL.SP parameter for purpose (1.). This parameter should be checked by the SP. For (2.), the HTTP-Receiver of the redirect message sent by the IdP is given in the OpenID parameter URL.SP. Here the IdP must check that this parameter is valid.

IdP Verification. The SP receiving a token should verify: (1.) the origin of the received token and (2.) the validity of the statements contained. (1.) is verified in three steps: (1.1) The SP must determine the unique identity of the IdP (e.g. an URL) which issued the authentication token. (1.2) The SP must fetch the corresponding key material associated to that identity. (1.3) Using this key material, the signature of the token is verified. In (2.) the SP should verify whether the IdP is allowed to make the statements in the token, for example, $\mathcal{I}dP_{\mathcal{A}}$ should not be able to issue tokens in the context of other IdPs.

Cryptographic Token Verification. For step (1.3) above, the signed parts must be determined. The SP must be able to distinguish signed from unsigned parts within the token. For instance, in OpenID, it should be able to distinguish signed HTTP header fields from unsigned ones. Additionally, it should check if all parameters that are required to be signed are indeed signed[12]. For step (1.2) above, the right keys must be chosen. The SP uses the key material associated with the selected IdP. If this association between key material and identity can be overwritten (cf. Section 5.2), novel attacks are feasible.

5. Novel Attacks

In this section, we give generic descriptions of two novel attack classes on OpenID, which are effective against different implementations of OpenID (cf. Table 1). The only prerequisite to apply our attacks is the possibility to use an arbitrary IdP[13]. The first attack, ID Spoofing, exploits characteristics in OpenID and uses a novel concept of injection the victim's identity using a malicious IdP. The second attack class, Key Confusion, introduces different attack techniques bypassing the integrity protection of authentication tokens by enforcing the usage of wrong keys. If a target is vulnerable against IDS or KC, *all accounts on the SP can be compromised without any victim's interaction.*

5.1. ID Spoofing

IDS introduce a novel class of attacks tricking the target SP to authenticate the attacker as the victim (URL.ID$_{\mathcal{V}}$) and allow the access to victim's resources. IDS attacks target condition (2.) of the *IdP verification* step (cf. Section 4). The attacker acts as a malicious client and uses his malicious IdP to execute IDS. Since the IDS attack does not require any victim interaction and only a malicious IdP is necessary, the attacker can break into all accounts on the target SP. IDS introduces two different strategies to achieve this goal.

Strategy 1. *Identity Spoofing in the token.* The malicious IdP (URL.$\mathcal{I}dP_{\mathcal{A}}$ = `http://badidp.org`) is used to create a token t^* containing the victim's identity (URL.ID$_{\mathcal{V}}$ = `http://idp1.com/alice`). The token is sent to the target SP and the attack is successful if it accepts t^*. Given the simplicity of this attack it is surprising that it has not been described before.

In OpenID, a user's identity is represented by URL, which is controlled by exactly one IdP. In the example above, the identity URL.ID$_{\mathcal{V}}$ = `http://idp1.com/alice` belongs to $\mathcal{I}dP_{\mathcal{V}}$ with URL.$\mathcal{I}dP_{\mathcal{V}}$= `http://idp1.com`. Consequently, an IdP can make statements only for user identities bound to its domain. Thus, $\mathcal{I}dP_{\mathcal{A}}$ should in theory not be able to create a valid token t^* containing URL.ID$_{\mathcal{V}}$. For OpenID, the corresponding check should work as follows: According to the specification [15, Section 11.2], an SP should start a (second) discovery on the identity URL.ID$_{\mathcal{V}}$ contained in t^*. In this manner, SP can discover whether URL.ID$_{\mathcal{V}}$ belongs to the IdP contained in t^*, i.e. $\mathcal{I}dP_{\mathcal{A}}$ in this case. If this step is not implemented properly, an attacker is able to inject identities, which are not controlled by his malicious IdP. In this manner, the attacker can impersonate users with different, trustworthy IdPs, for example, Paypal or Yahoo, by using only his own $\mathcal{I}dP_{\mathcal{A}}$.

Strategy 2. *Identity Spoofing in the discovery phase.*

The ID-Server in OpenID's discovery phase commonly returns the URL of the IdP that is going to be used (e.g. URL.$\mathcal{I}dP_{\mathcal{C}}$ = `http://idp1.com`). Besides returning URL.$\mathcal{I}dP_{\mathcal{C}}$, OpenID has a feature to return optionally a second "local" $ID^2_{\mathcal{C}}$ in addition[14]. This value is transmitted during the discovery phase and is not part of the SSO token. Thus, it is not protected by a signature. In IDS, Strategy 2, $ID^2_{\mathcal{C}}$ is set to URL.ID$_{\mathcal{V}}$ = `http://idp1.com/alice` during the discovery phase. Later on, the malicious IdP generates a valid token containing the attacker's normal identity

12. In the context of SAML, this has been shown to be quite challenging [14].

13. Note that this is one of OpenID's main features. Nevertheless, some online SPs use a hard-coded list of supported IdPs so that our attacks are not applicable.

14. This parameter is originally intended to determine $\mathcal{C}$'s concrete identity on its IdP, e.g. if he owns several ones on it.

(URL.ID$_\mathcal{A}$ = `http://badidp.org/attacker`) and sends it to the SP.

Once the SP receives the token, it verifies the signature, which is valid. Then, if the SP uses the $ID^2_\mathcal{C}$ = URL.ID$_\mathcal{V}$ = `http://idp1.com/alice` parameter to login the user, the attacker is successfully logged in as the victim. A detailed description is given by the example of ownCloud in Section 8.1).

Strategy 3. *Identity Spoofing via Email.* In OpenID, the authentication token can contain additional data about the user like first name, last name, email, gender etc. This data is mostly used by the SPs during the registration process of new users to automatically fill out some required text fields. However, this information should not be used for the authentication since OpenID does not provide any mechanisms to verify the correctness of these statements. For instance, an attacker using a malicious IdP can issue tokens containing arbitrary email addresses like `admin@gmail.com`.

To perform the attack, the attacker uses his malicious IdP and issues a valid token containing his OpenID identity, e.g. URL.ID$_\mathcal{A}$ = `http://badidp.org/attacker` and the victim's email address. Afterwards, the woken will be sent to the SP. Since the token is valid, the verification is successful and the SP uses the email address to authenticate the user. As a result, the attacker gets access to any account on the SP.

Please note that this attack differs from the attack described in [16] since the attacker does not manipulate the authentication request and all required parameters are signed within the authentication token.

5.2. Key Confusion

KC introduces a novel class of attacks forcing the target SP to use a key of the attacker's choice for the verification of tokens. The enforced key is a legit key that is shared between the target SP and the malicious IdP. However, during the KC attack, the SP is convinced to believe, that the key belongs to the victim's (instead of the attacker's) IdP. KC attacks address the second part of the *cryptographic token verification* step (cf. Section 4). Similar to IDS, no victim interaction is necessary, and thus, all accounts on a vulnerable target can be compromised.

The idea of KC is related to the *untrusted keys* presented in [14]. But in contrast to it protocols like SAML and OAuth, in OpenID, all established keys between an IdP and an SP are considered to be trusted. In order to load and use the correct trusted key, the SP uses the association handle α. KC targets this handle and introduces strategies how to enforce the usage of wrong keys.

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

Strategy 1. *Overwriting the secret key handle of a trusted IdP.* In the case of OpenID, the key material is referenced by the association handle parameter α. Since the value of α is chosen by the IdP (and not by the SP), the attacker (acting as a malicious IdP) is able to set α to the same value as defined by the valid IdP in order to overwrite it with its own key values. The attacker may get to know the original α by starting an attempt to log in as the victim on the target SP.

Strategy 2. *Submit attacker's own key handle for signature verification.* The association α is also part of the signed token t^*. Thus, some SP implementations are tempted to use this value to verify the signature. The fact that the token may be issued by a malicious IdP clearly shows that this leads to a critical vulnerability: Suppose SP and $\mathcal{I}dP_\mathcal{V}$ share a secret identified by α. Additionally, SP and $\mathcal{I}dP_\mathcal{A}$ share a secret identified by $\boldsymbol{\beta}$. If a malicious $\mathcal{I}dP_\mathcal{A}$ issues the token $t^* = (\text{URL.ID}_\mathcal{V}, \text{URL.}\mathcal{I}dP_\mathcal{V}, \text{URL.}SP, \boldsymbol{\beta})$, and the target SP accepts this token, it is vulnerable to KC.

Strategy 3:. *Session overwriting* According to the OpenID specification [15, Section 11.2], an SP should verify that the discovered information (user's identity URL.ID and URL.$\mathcal{I}dP$) matches the presented content in the received token. If the SP provides this check, *Strategy 2* fails, because the discovered IdP ($\mathcal{I}dP_\mathcal{A}$) does not match the IdP within the authentication token ($\mathcal{I}dP_\mathcal{V}$).

Unfortunately, this check does not include a verification that the key used to sign the token belongs to the discovered IdP. This allows again to bypass the verification logic: if the attacker can *overwrite* the discovered information with the values of $\mathcal{I}dP_\mathcal{V}$ before the authentication token is received and the SP uses the key identified by β, the attack is successful.

Commonly, web applications use a session variable to store user information used across multiple pages (e.g. username, favorite color, etc). In OpenID, the discovered information can be stored in a session variable. Later on, the attacker can change it by either by manipulating some of the values in the browser or – in case of OpenID – by starting a second Discovery on $\mathcal{I}dP_\mathcal{A}$. Consequentially the SP overwrites the old discovered information with the new one.

A detailed example and explanation of the attack is given in Section 8.3.

The idea of KC can be adapted to other SSO protocols using on-the-fly trust establishment and considering all established keys as trusted, e.g. OpenID Connect or BrowserID.

5.3. Token Recipient Confusion

Token Recipient Confusion (TRC) attacks as shown in Figure 6 target a missing URL.SP parameter verification. This violates condition (2.) of the *token recipient verification* step (cf. Section 4) and allows an attacker to use a token t for one SP on a different SP. TRC requires an interaction by the victim and thus, the attack can compromise only the account of this victim (cf. IDS and KC which compromises all accounts). The original concept of this attack is taken from [6], and we have adopted it to OpenID. Note that using our malicious IdP, we can *detect* the vulnerability easily. However, the attack *exploit* does not require a malicious IdP.

Detection phase. The attacker uses $\mathcal{I}dP_\mathcal{A}$ and generates tokens containing identity $ID_\mathcal{A}$. Additionally, he sets the value of URL.SP to an arbitrary URL (different from the URL of the target SP) and sends the token to the target SP. Finally, he observes the behavior of the target SP: If the SP accepts the token, then the value of URL.SP is *not* validated, and TRC is applicable.

Again, to detect whether an SP is vulnerable to TRC, we use our malicious IdP. No victim interaction is necessary for the detection.

Exploit phase. In order to exploit the vulnerability, the attacker $\mathcal{A}$ sets up a web application running on URL.$\mathcal{A}$ (e.g. a weather forecast service), to initiate an OpenID authentication and to collect authentication tokens.

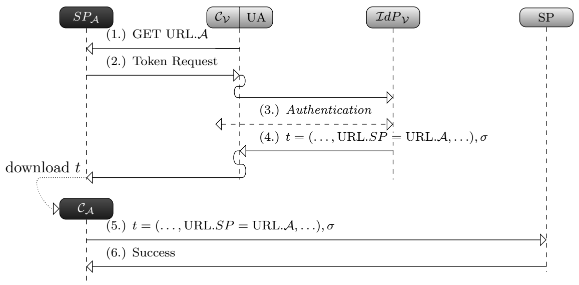

$SP_{\mathcal{A}}$
$\mathcal{C}_{\mathcal{V}}$
UA
$\mathcal{I}dP_{\mathcal{V}}$
SP
(1.) GET URL.$\mathcal{A}$
(2.) Token Request
(3.) *Authentication*
(4.) $t = (\ldots, \text{URL}.SP = \text{URL}.\mathcal{A}, \ldots), \sigma$
download t
$\mathcal{C}_{\mathcal{A}}$
(5.) $t = (\ldots, \text{URL}.SP = \text{URL}.\mathcal{A}, \ldots), \sigma$
(6.) Success

Figure 6: Token Recipient Confusion Attack.

The exact protocol flow is shown in Figure 6: (1.) The victim client ($\mathcal{C}_{\mathcal{V}}$) accesses the web application deployed on URL.$\mathcal{A}$. (2.) The attacker creates a *Token Request* containing URL.SP = URL.$\mathcal{A}$. (3.) $\mathcal{C}_{\mathcal{V}}$ authenticates to $\mathcal{I}dP_{\mathcal{V}}$. If he is already authenticated, this step is skipped. (4.) $\mathcal{I}dP_{\mathcal{V}}$ generates the token t and sends it back to $\mathcal{C}_{\mathcal{V}}$, with a redirect to URL.SP = URL.$\mathcal{A}$. The client's UA executes this redirect, and thus sends the token to $\mathcal{A}$. (5.) Finally, $\mathcal{C}_{\mathcal{A}}$ downloads the collected token t from $SP_{\mathcal{A}}$ and uses it to log in on the target SP.

TRC is a generic attack and can be adapted to other SSO protocols like SAML and OAuth, since these include parameters similar to URL.SP [6], [17]. To mitigate the TRC attack, the SP should verify whether the URL.SP parameter contained in t matches its own URL.

6. OpenID Attacker

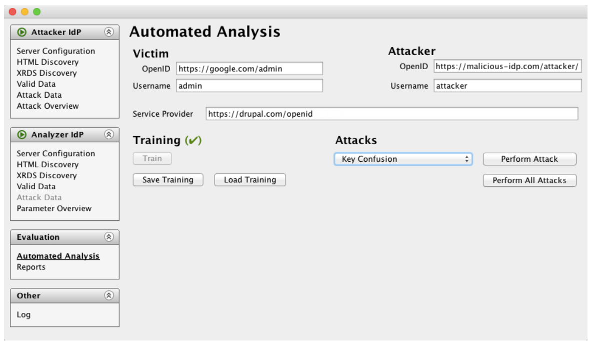
6.1. Fully Automated Analysis

OpenID Attacker's configuration of the fully automated analysis can be seen in Figure 7.

We developed *OpenID Attacker* as a part of our research and as a result of our token verification model for SPs. OpenID Attacker is an open source penetration test tool that acts as an OpenID IdP and offers a Graphical User Interface (GUI) for easy configuration, see Figure 7.

As such, it is able to operate during all three phases of the OpenID SSO protocol. OpenID Attacker is free, open source and can be downloaded here [13].

The main advantage of OpenID Attacker is its flexibility – the attacks can be provided manually or full automatically. As shown in Figure 8, OpenID Attacker works in



Automated Analysis

Victim
OpenID https://google.com/admin
Username admin

Attacker
OpenID https://malicious-idp.com/attacker/
Username attacker

Service Provider https://drupal.com/openid

Training (✓)

Attacks
Key Confusion

Figure 7: OpenID Attacker supports fully automated analysis. One has just to configure the victim's and the attacker's accounts, and select one or all attacks to perform.

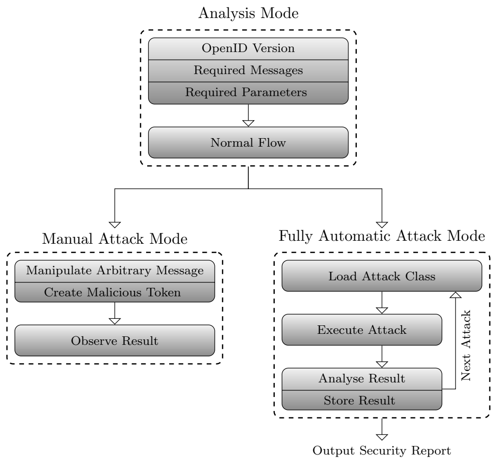


Figure 8: The three modes of OpenID Attacker.

three modes: (1.) Analysis, (2.) Manual Attack, (3.) Fully Automatic Attack.

Configuration. Since OpenID Attacker acts as a malicious IdP, it should be reachable on the Internet. In this manner, the target SP can discover OpenID Attacker, establish the keys and later on verify the token. OpenID Attacker enables the fine granular configuration of every phase: discovery, association and token processing. For instance, OpenID Attacker can set up arbitrary HTML/XRDS discovery documents, association expiration time and even association handle value.

Manual Analysis Mode. In this mode OpenID Attacker is used to analyze the normal behavior of the target SP. For this purpose, OpenID Attacker acts as benign IdP and creates valid tokens. Note that no attacks are executed, but only information about the target SP is collected. Additionally, no configuration of the target SP is needed.

Initially, we navigate our browser to the target SP and initiate the login with our IdP (URL.$\mathcal{I}dP_{\mathcal{A}}$). The SP communicates with our IdP and executes the discovery and association phase. OpenID Attacker stores all information exchanged with the SP and collects information about the supported features, e.g. OpenID version and HTML or/and

XRDS discovery. Moreover, in the *token processing* phase OpenID Attacker collects and stores information about the exact messages flow, optional messages, schema of messages, required/optional attributes and more. Thus we, in the role of a security analyst, have a very detailed overview about the implementation, information flow and all supported features.

Manual Attack Mode. In this mode, OpenID Attacker acts as a malicious IdP hand-operated by the attacker. We can start the security analysis on basis of the information stored in *Manual Analysis Mode*, manipulate parameters in each message, create malicious tokens in this manner, and then observe the results when sending it to the targeted SP. In this mode, the attacks and the evaluation of the attacks are carried out manually.

The idea behind the *Manual Mode* is the fact that new attack vectors can be inspected. This is an important fact, because the *Manual Mode* allows investigating the OpenID protocol very deeply and fine granular as every single aspect of the protocol can be manipulated. In combination with a running SP implementation in debugging mode, this mode helps to understand the source code of the SP to find implementation as well as protocol issues. We used this mode to discover the IDS, KC, and TRC attacks during a white-box analysis.

Fully Automatic Attack Mode. In this mode OpenID Attacker acts as fully automated malicious IdP penetration test tool. OpenID Attacker is reachable with two different domains (URL.$\mathcal{I}dP_{\mathcal{V}}$ and URL.$\mathcal{I}dP_{\mathcal{A}}$) so that we can simulate the entire communication with a victim's benign IdP ($\mathcal{I}dP_{\mathcal{V}}$) and attacker's malicious IdP ($\mathcal{I}dP_{\mathcal{A}}$). The execution of the fully automated testing consists of two parts: (1.) *training* and (2.) *attack execution*.

(1.) Training:. In this mode OpenID Attacker is used to analyze the normal behavior of the target SP and uses the same concept as for the manual analysis. For this purpose, OpenID Attacker acts as benign IdP and creates valid tokens both in the role of URL.$\mathcal{I}dP_{\mathcal{V}}$ and URL.$\mathcal{I}dP_{\mathcal{A}}$.

For the correct evaluation of the tested attacks it is essential for OpenID Attacker that it can: (1.) determine if the login was successful (e.g. no errors were thrown) and (2.) determine in which account it was logged in – the victim's or the attacker's one. Only access to the victim's account is considered as a successful attack. Thus, there are three main categories according the status of the tested attack: authenticated as the *attacker*, authenticated as the *victim*, *error*.

OpenID Attacker has to be "trained" in order to make a difference between the different results and to categorize them in one of the three categories. For that purpose multiple successful authentication procedures with URL.$\mathcal{I}dP_{\mathcal{V}}$ will be executed initially. Then, misconstrued messages will be sent to the target SP in order to trigger error messages. Consequentially, the same procedure is repeated with URL.$\mathcal{I}dP_{\mathcal{A}}$. During the entire communication in the training phase OpenID Attacker records the messages plus the SP's reaction and categorize them. At the end of the phase, OpenID Attacker knows the behavior of the SP and can proceed with the execution of the attacks.

In order to automate the entire training and login process, we use Selenium[15]. Selenium enables the fully automated usage of a browser, e.g. Firefox. Thus, it can start the browser, call the URL of the target SP, fill out some input fields on the loaded web page, e.g. the *openid identity*, and trigger click events in order to submit the entered data and initiate the OpenID authentication on the target SP. As a result, the authentication can be automated. Note that it is enough to enter the login URL in OpenID Attacker. Finding the OpenID login formular is also performed automatically.

(2.) Attack execution:. OpenID Attacker loads training results and it then sequentially executes the attacks defined in Section 5[16]. Then, OpenID Attacker analyzes the result of the attack in comparison to the training set by using the *simmetrics*[17] string comparison library. This allows OpenID Attacker to decide, whether a) the login was successful and b) with which account (URL.ID$_{\mathcal{A}}$ or URL.ID$_{\mathcal{V}}$) the attacker is logged in.

In conclusion, OpenID Attacker summarizes the results of all evaluated attacks and creates a security report, which can be exported as a HTML document (cf. Figure 9).

7. Methodology

Target SPs. We selected 15 open source implementations including libraries and frameworks that support OpenID, mainly taken from the official OpenID website [12][18]. We tried to cover every available language: Our list contains implementations in .NET, C++, ColdFusion, Java, JavaScript, Perl, PHP, Python, and Ruby. We added Drupal to the target list, since it is a widely used content-management system (CMS) and has a custom implementation of OpenID. The only implementation that did not permit a white-box analysis is Sourceforge [18]. We included it because it is a very prominent site supporting OpenID and because it does not use one of the inspected implementations listed on [12].

Setup. For each implementation, we created a working virtual web server/virtual CMS server, and deployed the framework in it. For Sourceforge, we used the live website.

We registered two accounts on each target as SP. As victim $\mathcal{V}$, we used an account at a trusted IdP to register a local account on the target SP. Using a second browser on a different PC we registered a second account for $\mathcal{A}$ at the target SP, associated with an account on our custom malicious IdP – the OpenID Attacker account.

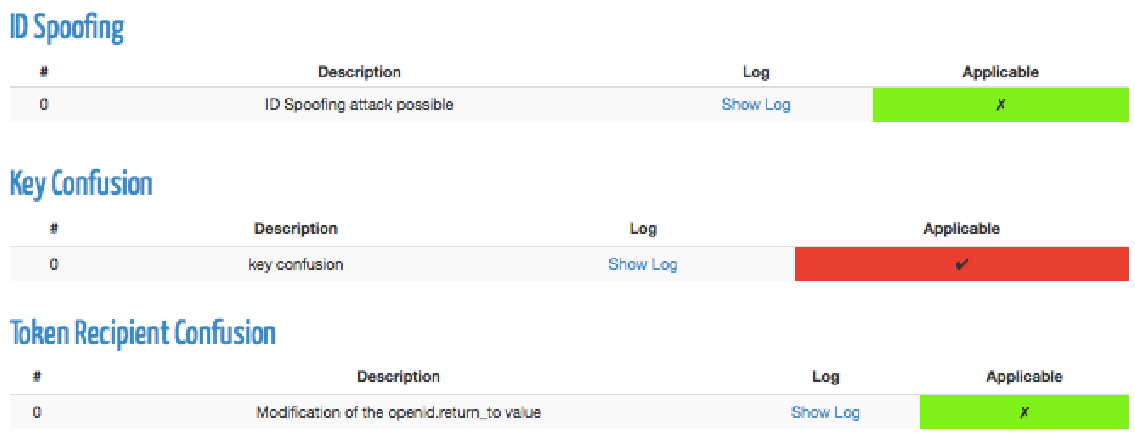
In this step, the second account was mainly used to verify that the OpenID Attacker IdP is working flawlessly and that the target is able to verify valid tokens created by our tool.

15. http://www.seleniumhq.org/

16. OpenID Attacker supports even more attacks such as XXE, replay attacks, etc.

17. http://sourceforge.net/projects/simmetrics/

18. Note that some of the libraries are listed multiple times, for example, libopkele is the module used in Apache mod_auth_openid, the listed Python Django OpenID framework uses janrain etc.



ID Spoofing

#	Description	Log	Applicable
0	ID Spoofing attack possible	Show Log	✗

Key Confusion

#	Description	Log	Applicable
0	key confusion	Show Log	✔

Token Recipient Confusion

#	Description	Log	Applicable
0	Modification of the openid.return_to value	Show Log	✗

Figure 9: The *Fully Automatic Attack Mode* outputs a security report. More details can be seen in the log.

White-Box Tests using OpenID Attacker. We used white-box tests to analyze the source code and the protocol flow of each target. OpenID Attacker, running in manual analysis and manual attack mode, was used in order to get a better understanding of OpenID is implemented on the target SP. This allowed us to develop and to apply the concepts for the attack classes described in Section 5.

Black-Box Tests using OpenID Attacker. Black-Box testing is more complicated than white-box since only the result of the attack is visible, but not the reasons for this result. One way to better understand the implementation is to record the messages in the different phases. Consequentially, via OpenID Attacker the parameters within the different phases can be varied in order to learn, which of them are processed by the SP. A simple example of such a test is to exclude the signature within the token and observe the reaction of the SP.

In order to provide a well-structured and comprehensible Black-Box test we consider all verification steps described in Section 4. According to every verification step, we developed a test suite, which we apply and analyze systematically. Finally, we summarize the results of all tests. Based on this results we can start attacks.

Exploit. Finally, we performed the attacks in the web attacker model. Note that for IDS and KC, no interaction with the victim is necessary – if the exploit works, we could login at the SP with an arbitrary identity. Only for TRC, it is necessary that victim $\mathcal{V}$ visits a web page $SP_{\mathcal{A}}$ hosted by the attacker $\mathcal{A}$. In our setting, $\mathcal{V}$ is already authenticated to the trusted IdP (stored in a session cookie), so that no explicit authentication of $\mathcal{V}$ is necessary. We verify that the token t is indeed transferred to $SP_{\mathcal{A}}$, and that we could use this token from our second browser to gain access to the target SP.

For the IDS attack we only needed to know the identity of $\mathcal{V}$. We verified that the target SP is either vulnerable for strategy 1 or strategy 2. In each case, we are logged in with the identity $\mathcal{V}$.

To verify KC attacks, we have sketched two strategies in Section 5. For following the first strategy, the precondition that an association α exists between the target SP and the trusted IdP must be fulfilled. We can get the value of α in message (4.) of Figure 5 when we try to log in with the victim's identity. This attempt will not succeed, but we can see message (4.) nonetheless.

We then established a new association between the target SP and OpenID Attacker using the same α and analyzed whether the target SP afterwards accepted our malicious tokens as valid for $\mathcal{V}$. For the second strategy, only an association β between the target SP and the malicious IdP is necessary. We verified that the SP accepted tokens containing $(\text{URL.ID}_{\mathcal{V}}, \text{URL.}\mathcal{I}dP_{\mathcal{V}})$ that were signed with the malicious association β.

8. Library Evaluation

We reported all vulnerabilities to the liable security teams and to the Computer Emergency Response Team (CERT). In case we got a response from the developers, the time to fix the reported issues ranged between a few days and several months. Furthermore, we supported the developer teams during fixing the reported issues.

Our results are summarized in Table 1: for 12 out of 17 targets, we were able to access a protected resource. On ten of the twelve targets an attacker can compromise *all* of the accounts, without any user interaction. On the other two targets the account of any victim can be compromised, if he visits a malicious website.

8.1. ID Spoofing

Eight of the tested targets were vulnerable to IDS. Those targets were fully compromised – all OpenID accounts could be accessed without any interaction of the victim, and even worse, the victim is unable to detect and mitigate IDS.

ownCloud. OwnCloud [19] is a PHP-based, open source cloud framework. Its OpenID implementation is interesting, because ownCloud does not verify the token's signature itself. Instead, it uses the *check authentication* OpenID feature [15, Section 11.4.2]: ownCloud sends the token t to the according IdP and let it verify t (instead of verifying t itself). This means that using OpenID Attacker to send, for example, a token for a Yahoo account would lead ownCloud to send the token directly to the Yahoo server for verification, which will not accept it.

By examining the OpenID's OpenID message flow, we found out that it is vulnerable to IDS, Strategy 2. The attack works as follows

8.2. Attacking Owncloud

The attack on ownCloud is depicted in Figure 14.

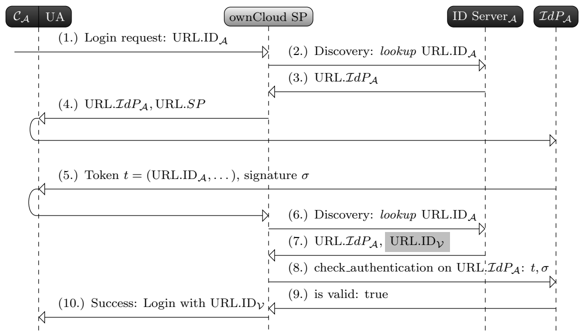

$\mathcal{C}_\mathcal{A}$
UA
ownCloud SP
ID Server$_\mathcal{A}$
$\mathcal{I}dP_\mathcal{A}$
(1.) Login request: URL.ID$_\mathcal{A}$
(2.) Discovery: *lookup* URL.ID$_\mathcal{A}$
(3.) URL.$\mathcal{I}dP_\mathcal{A}$
(4.) URL.$\mathcal{I}dP_\mathcal{A}$, URL.$SP$
(5.) Token $t = (\text{URL.ID}_\mathcal{A}, \dots)$, signature σ
(6.) Discovery: *lookup* URL.ID$_\mathcal{A}$
(7.) URL.$\mathcal{I}dP_\mathcal{A}$, URL.ID$_\mathcal{V}$
(8.) check_authentication on URL.$\mathcal{I}dP_\mathcal{A}$: t, σ
(9.) is valid: true
(10.) Success: Login with URL.ID$_\mathcal{V}$

Figure 10: The ID Spoofing attack on ownCloud: The attacker's ID server returns URL.ID$_\mathcal{V}$ on the second discovery. ownCloud uses this identity value for the login instead of the identity provided within the token.

(cf. Figure 14): When ownCloud receives the OpenID token (Step 5.), it performs a rediscovery on the contained identity. We configured OpenID Attacker to include the victim's identity URL.ID$_\mathcal{V}$ in the discovered document (Step 7.) additionally to URL.$\mathcal{I}dP_\mathcal{A}$. Afterwards, ownCloud sends the token to the attacker's $\mathcal{I}dP_\mathcal{A}$ (Step 8.) by using the discovered URL.$\mathcal{I}dP_\mathcal{A}$ and it returns that the token is valid (Step (9.). Surprisingly, instead of using the URL.ID$_\mathcal{A}$ contained in t to log in the user, ownCloud uses URL.ID$_\mathcal{V}$ (that was returned in Step 7.). We were logged in with the victim's identity.

We contacted the ownCloud security team, reported the issue and they acknowledged our work.

Sourceforge. Initially, we started a black-box testing and detected that Sourceforge was vulnerable against IDS, Strategy 1. This investigation was in 2014 and before our larger online evaluation described in Section 9. Consequentially, we contacted the support team and described the issue. Later on, they answered us that vulnerability is fixed. We analyzed Sourceforge again. Using OpenID Attacker, we found out that the IDS attack was no longer possible. However, we performed a KC attack and found out, that Sourceforge is vulnerable to this attack class. The attacks on Sourceforge showed us, how to apply our white-box analysis attacks on a black-box system. We were able to attack Sourceforge in a fully-automatic manner without knowing its exact OpenID implementation.

8.3. Key Confusion with Session Overwriting

Three targets were vulnerable to Key Confusion (KC): Drupal, Zend Framework and Sourceforge. These implementations used a key belonging to OpenID Attacker for verifying the signature instead of using the key belonging to the victim's IdP. The attack on Drupal worked as follows:

Drupal.

Drupal [20] is a free open source CMS. It is based on PHP and according to [21], it is the third most frequently used CMS. Its OpenID support is shipped with every Drupal distribution and just needs to be activated within the settings menu.

Starting our white-box analysis on Drupal, we submitted URL.ID$_\mathcal{A}$ on the login form. The SP starts the discovery on it and receives URL.$\mathcal{I}dP_\mathcal{A}$ belonging to our OpenID Attacker IdP. Drupal redirects us to it, but instead of creating a token for URL.ID$_\mathcal{A}$, it creates a token $t^* = (\text{URL.ID}_\mathcal{V}, \dots)$ containing the victim's Yahoo identity (IDS attack, strategy 1). Sending t^* to Drupal did not succeed. Drupal noticed that the originally submitted identity URL.ID$_\mathcal{A}$ differs from the value URL.ID$_\mathcal{V}$ contained in t^*. As a result, Drupal starts a second discovery on URL.ID$_\mathcal{V}$, which returns URL.$\mathcal{I}dP_\mathcal{V}$. Drupal compares this value to URL.$\mathcal{I}dP_\mathcal{A}$ returned by the first discovery. Since the values are not equal, we are not logged in. Interestingly, Drupal does *not* compare the discovered value with the value URL.$\mathcal{I}dP$ contained in t^*, thus sending a token $t^* = (\text{URL.ID}_\mathcal{V}, \text{URL.}\mathcal{I}dP_\mathcal{V}, \dots)$ also fails.

In order to prevent the second discovery process, which mitigates the attack, we analyzed the source code. We found out that Drupal uses the PHP `$_SESSION` variable to store and load URL.ID and URL.$\mathcal{I}dP$. In this manner, Drupal links both messages: the login request and the received token.

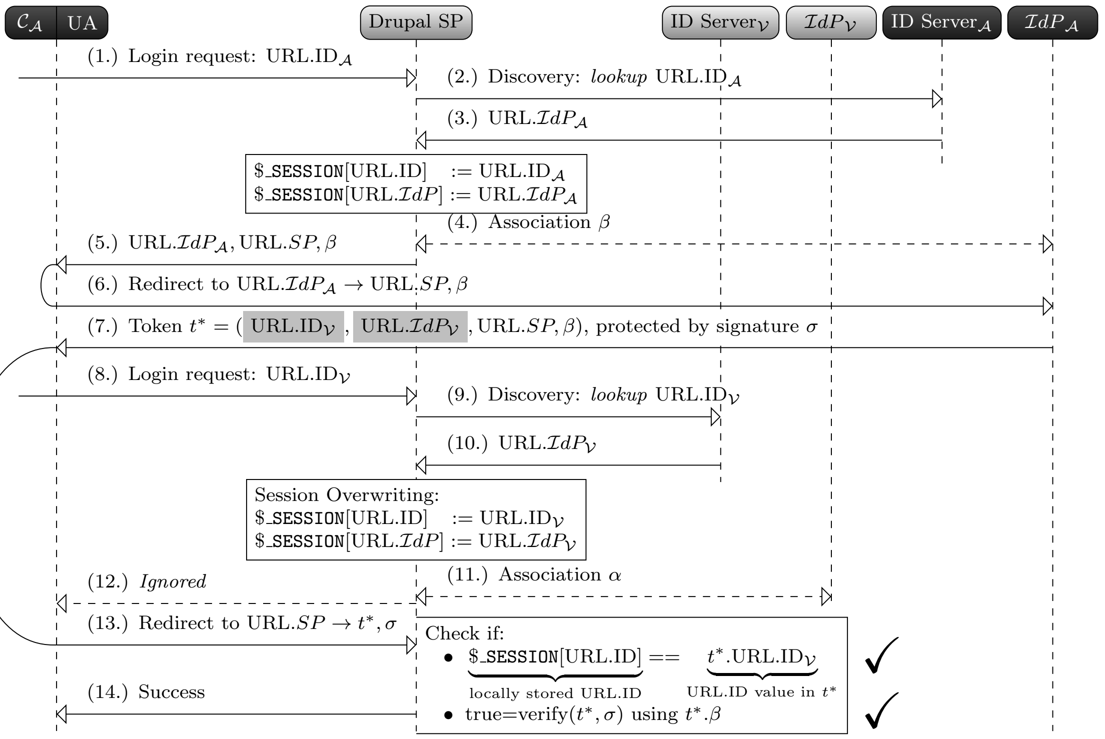
The `$_SESSION` variable is a globally available PHP array which holds arbitrary session data on a per-user basis. Whenever Drupal receives an OpenID token t^*, it first verifies if the URL.ID parameter, contained in t^*, matches the value stored in `$_SESSION`. If they differ, as in the case of the IDS attack, Drupal starts again a discovery on URL.ID contained in t^*. The discovery returns the corresponding URL.$\mathcal{I}dP$ and if these values do not match the URL.$\mathcal{I}dP$ parameter stored in `$_SESSION`, t^* is not accepted.

To finally prevent the second discovery and to bypass the verification logic, we had to overwrite the `$_SESSION` variable. The attack is shown in Figure 11 and works as follows:

(1.)-(3.) A login request with the attacker's account URL.ID$_\mathcal{A}$ is started. Drupal discovers it and stores URL.ID$_\mathcal{A}$ and URL.$\mathcal{I}dP_\mathcal{A}$ in `$_SESSION`.

(4.) Drupal starts an association with $\mathcal{I}dP_\mathcal{A}$, which returns β (using KC strategy 3).

(5.)-(7.) Drupal redirects the attacker to URL.$\mathcal{I}dP_\mathcal{A}$. OpenID Attacker creates a token $t^* = (\text{URL.ID}_\mathcal{V}, \text{URL.}\mathcal{I}dP_\mathcal{V}, \text{URL.}SP, \beta)$. Then, the attacker delays the sending of the token to Drupal.



$\mathcal{C}_{\mathcal{A}}$
UA
Drupal SP
ID Server$_{\mathcal{V}}$
$\mathcal{I}dP_{\mathcal{V}}$
ID Server$_{\mathcal{A}}$
$\mathcal{I}dP_{\mathcal{A}}$
(1.) Login request: URL.ID$_{\mathcal{A}}$
(2.) Discovery: *lookup* URL.ID$_{\mathcal{A}}$
(3.) URL.$\mathcal{I}dP_{\mathcal{A}}$
\$_SESSION[URL.ID] := URL.ID$_{\mathcal{A}}$
\$_SESSION[URL.$\mathcal{I}dP$] := URL.$\mathcal{I}dP_{\mathcal{A}}$
(4.) Association β
(5.) URL.$\mathcal{I}dP_{\mathcal{A}}$, URL.$SP$, β
(6.) Redirect to URL.$\mathcal{I}dP_{\mathcal{A}} \rightarrow$ URL.SP, β
(7.) Token $t^* =$ (URL.ID$_{\mathcal{V}}$, URL.$\mathcal{I}dP_{\mathcal{V}}$, URL.SP, β), protected by signature σ
(8.) Login request: URL.ID$_{\mathcal{V}}$
(9.) Discovery: *lookup* URL.ID$_{\mathcal{V}}$
(10.) URL.$\mathcal{I}dP_{\mathcal{V}}$
Session Overwriting:
\$_SESSION[URL.ID] := URL.ID$_{\mathcal{V}}$
\$_SESSION[URL.$\mathcal{I}dP$] := URL.$\mathcal{I}dP_{\mathcal{V}}$
(11.) Association α
(12.) *Ignored*
(13.) Redirect to URL.$SP \rightarrow t^*, \sigma$
Check if:
• $\underbrace{\text{\$\_SESSION[URL.ID]}}_{\text{locally stored URL.ID}} == \underbrace{t^*\text{.URL.ID}_{\mathcal{V}}}_{\text{URL.ID value in } t^*}$
• true=verify(t^*, σ) using $t^*.\beta$
✓
✓
(14.) Success

Figure 11: Key Confusion attack on Drupal: Before the token t^* in Step (7.) is forwarded to Drupal in Step (13.), the attacker $\mathcal{C}_{\mathcal{A}}$ starts a second login request in Step (8.) using the victim's identity URL.ID$_{\mathcal{V}}$. This overwrites the URL.ID and URL.$\mathcal{I}dP$ data stored in `$_SESSION` and prevents the second discovery.

(8.)-(10.) The attacker submits a further login request to Drupal, but this time with the victim's identity URL.ID$_{\mathcal{V}}$. Drupal starts a new discovery on it and receives URL.$\mathcal{I}dP_{\mathcal{V}}$. Both values, URL.ID$_{\mathcal{V}}$ and URL.$\mathcal{I}dP_{\mathcal{V}}$, are then stored in `$_SESSION`, overwriting URL.ID$_{\mathcal{A}}$ and URL.$\mathcal{I}dP_{\mathcal{A}}$.

(11.) Drupal starts another association with $\mathcal{I}dP_{\mathcal{V}}$, which returns α.

(12.) Drupal redirects the attacker to URL.$\mathcal{I}dP_{\mathcal{V}}$, but this redirect is not relevant for the attack.

(13.)-(14.) The halted token t in (7.) is now sent to Drupal. Drupal verifies the signature. The interesting point at this step is that Drupal loaded the key from the database by only using β contained in t^*. It does not verify whether the association β was really established with URL.$\mathcal{I}dP_{\mathcal{V}}$. Thus, the signature is valid. Then, Drupal compares the values of URL.ID$_{\mathcal{V}}$ and URL.$\mathcal{I}dP_{\mathcal{V}}$ contained in the token with the ones stored in `$_SESSION`. Because of being equal, there is no second discovery and we are logged in with the victim's identity.

We reported the issue to the Drupal security team and suggested to fix it by fetching the key via (URL.$\mathcal{I}dP, \alpha/\beta$) instead of using α/β only. They accepted the idea and implemented it in their new release Drupal releases. For a better understanding, we added a video as a demonstration of this attack that shows the usage of OpenID Attacker [22].

8.4. Additional Findings

The findings described here did not result in a valid attack according to our model, but are worth reporting.

Unsigned OpenID Parameters. The OpenID specification [15, Section 10.1] requires the following parameters to be signed: `op_endpoint`, `return_to`, `response_nonce`, `assoc_handle`, `claimed_id` and `identity`. 4 of 17 targets (CFOpenID, OpenID CFC, OpenID 4 Node.js, Zend Framework) accept tokens in which some of these parameters were not signed, and could thus be forged by an attacker.

XML External Entity. We determined that 2 of 17 analyzed targets (OpenID CFC, Net::OpenID::Consumer) are susceptible to XXE attacks [23], [24]. Additionally, we found out that Slashdot [25] (Alexa rank 1427) was vulnerable to XXE because of using the Net::OpenID::Consumer library. Interesting was the fact that lots of implementations used regular expressions (instead of an XML parser) to process the discovery phase, thus XXE was not possible in these cases.

Replay Attack. OpenID has only one parameter containing a timestamp (`openid.response_nonce`). It contains the creation time of the token concatenated with a random string, but does not include an expiration time. Thus, the SP can decide on its own how long it accepts such a token.

The lifetime of a token is additionally limited by the lifetime of the association and the corresponding key. We found that this lifetime varies heavily: associations with Yahoo have a lifetime of 4 hours, with Google 13 hours, and with MyOpenID 14 days.

9. Online Website Evaluation

One year after our reports to the developers of vulnerable OpenID implementations, we wanted to find out, if online websites are vulnerable to the attacks discussed in this paper.

Searching Methodology. The first task for the evaluation is to identify websites offering OpenID as a login system. Since there is no *Alexa-like* database that could be queried to get a list of OpenID websites, we elaborated techniques facilitating the searching process:

- The detailed knowledge of the protocol and the according parameters in the authentication request and token can be used to improve the searching results. For instance, a possible search term is `inurl: openid.claimed_id and openid.identity`. As a result, all URLs containing this parameters, will be displayed.
- Observing the analyzed frameworks in Table 1, we estimated that the term `login?openid` is commonly used in the URLs. By using this search term, we found the most of the analyzed websites.
- By using different search engines like Google, Bing and Yahoo, we extended our list of target websites.
- A helpful search engine is *NerdyData*, which analyses the source code of websites.
- There are also websites and blog entries listing several websites supporting OpenID, but visiting them reveals that they do no longer support the protocol.

All in all, we found 137 websites.

Set-Up. Next, we analyzed the login mechanisms on the websites. In 49% of them, we cannot provide the security evaluation due to the following problems:

- The website does not offer a public user registration (closed community). Thus, the registration of our test accounts was not possible.
- Faulty implementations led to unknown errors on the website and the login via OpenID was not feasible.
- The website supports only fixed IdPs (e.g. Yahoo) so that we could not apply our malicious IdP approach.
- The website requires payment with a credit card during registration. Testing non-free accounts was considered out-of-scope during this research.
- The website contained OpenID elements, like `openid2.provider`. However, no OpenID login mechanisms were provided. For instance, Amazon uses OpenID parameters for the transport of data, but not for authentication and within a SSO login.

As a result, on 70 of 137, we were able to login with OpenID (51%). Consequentially, we evaluated the websites satisfying our methodology described in Section 7.

Results. 26% of the tested websites (18 of 70) were vulnerable to one of our three attacks. On eleven websites, IDS was possible (16%), four were vulnerable to KC (6%) and TRC affected eight websites (16%). Although this was a black-box evaluation, we could identify Drupal and Net:OpenID:Consumer implementations in 7% of the cases. The websites used an updated version (after our security report) and were no longer vulnerable. Our results are depicted in Figure 12.

Compared to the analyzed frameworks in Table 1, where 71% were vulnerable, the number of vulnerable websites is lower – 26%. The reason for the result is the fact that many of the websites (41%) use the JanRain library since it is easy to integrate and supports plethora of SSO protocols like OAuth, Facebook Connect, OpenID and OpenID Connect. Thus, IDS Strategy 1 and 2, KC and TRC are not applicable on this websites.

OXID Shopping System. During our analysis, we successfully applied the IDS attack on eleven targets. The surprising fact was that six of them used the JanRain OpenID library, so we expected them to be not vulnerable (cf. Table 1). But instead of identifying a user by the URL.ID parameter, they used the email parameter. Thus, these implementations were vulnerable to IDS Strategy 3. This example illustrates that even a *secure* implementation like JanRain can be bypassed, if it is used incorrect. We investigated the websites further and found out, that they all used the OXID Shopping System[19]. We contacted the vendor and they acknowledged our work [26].

10. Related Work

Related work can be divided into three parts: research on analysis of SSO systems, specific investigations in the field of OpenID, and development of SSO testing tools. Please note that none of the previous papers considers malicious IdPs as part of the attacker, and none of the OpenID papers considered attacks on the association phase.

SSO Security. Various vulnerabilities have been found over the last two decades. In 2003 and 2006, Groß [6], [7] analyzed the SAML Browser/Artifact profile and identified several flaws in the SAML specification that allow connection hijacking/replay attacks, as well as Man-in-the-Middle (MitM) attacks and HTTP referrer attacks. We used these attacks as model for the TRC attack. In 2008 and 2011, Armando et al. [27], [28] built a formal model of the SAML V2.0 Web Browser SSO protocol and analyzed it with the model checker SATMC. The authors found vulnerabilities in Google's SAML interface. In 2012, Somorovsky et al. [14] investigated the XML Signature validation of several SAML frameworks. By using the XML Signature Wrapping (XSW) attack technique, they bypassed the authentication mechanism in 11 out of 14 SAML frameworks.

Sun et al. [29] analyzed the implementation of nearly 100 OAuth implementations, and found serious security flaws in many of them. Their research concentrated on classical web attacks like XSS, CSRF and TLS misconfigurations. Further security flaws in OAuth based applications

19. http://www.oxid-esales.com/

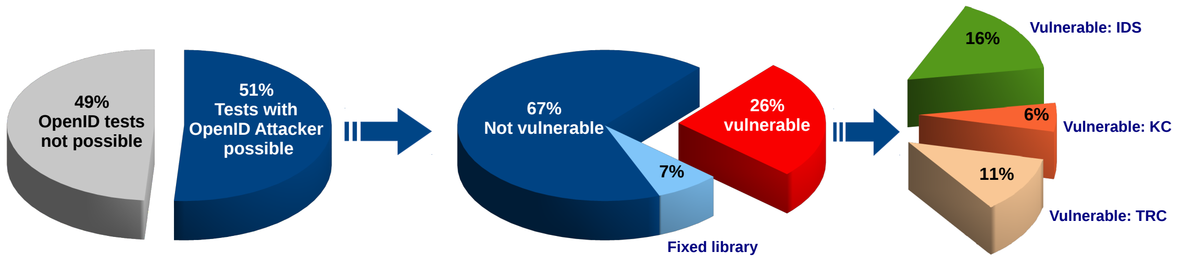


Figure 12: Statistic of our online website evaluation.

were discovered in [30]–[36], whereby the authors concentrated on individual attacks. In 2013 Wang et al. introduced a systematic process for identifying critical assumptions in SDKs, which led to the identification of exploits in constructed apps resulting in changes in the OAuth 2.0 specification [37]. Chen et al. revealed in 2014 serious vulnerabilities in OAuth applications on mobile devices caused by the developer's misinterpretation of the OAuth protocol [17].

In 2014 Fett et al. [38] built a formal model of the BrowserID protocol [39] , which allows them detect new weaknesses and vulnerabilities in BrowserID.

The concept of malicious IdPs was previously described in [40]–[42]. But please note that the described attacks are trivial in the sense that only those accounts are compromised which use (and therefore trust) this specific malicious IdP. These researches additionally investigate privacy concerns when users are using such a malicious IdP. In contrast, our malicious IdP-based attacks compromise accounts controlled by other benign IdPs (e.g. Yahoo). To the best of our knowledge, none of the previous work considered this kind of attacks, which is our main contribution.

OpenID Security. The analysis of the OpenID protocol started with version 1.0. Eugene Tsyrklevich and Vlad Tsyrklevich [43] presented several attacks on this OpenID version at Black Hat in 2007. They identified, for instance, a threat in the IdP endpoint URL (URL.$\mathcal{I}dP$) published within the discovery phase. It can point to critical files on the local machine or can even be abused in order to start a Denial-of-Service (DoS) attack by enforcing the SP to download a large movie file. Comparable to [29], they also looked at replay and CSRF attacks.

In 2008, Newman and Lingamneni [44] created a model checker for OpenID 2.0, but for simplicity, they removed the association phase out of their model. By using it, they could identify a session swapping vulnerability, which enforces the victim to log in into attacker's account on an SP. In this manner, an attacker could eavesdrop the victim's activities. In comparison to our work, the attacks presented in [44] do not result in unauthorized access. Interestingly, the authors of the paper modeled an IdP capable to make associations with legitimate SPs. However, they did not consider a malicious IdP capable to start attacks like IDS. Since KC is related to the association phase, the attack was not covered by the model checker. Later on, Sun et al. [45] provide a comprehensive formal analysis on OpenID and an empirical evaluation of 132 popular websites. The authors investigated CSRF, Man-in-the-middle attacks and the SSL support of OpenID implementations. In contrast to our work, they assumed that the SP and the IdP were trustworthy, so that they could not identify any of the attacks presented in this paper.

In 2010, Delft et al. [46] published an attack describing KC Strategy 1 – overwriting key material on the SP. However, Strategy 2 of KC was not considered. Additionally, the authors evaluated three OpenID libraries as part of their research.

Finally, Wang et al. [16] concentrated on real-life SSO systems instead of a formal analysis. They have well demonstrated the problems related to token verification with different attacks. They developed a tool named BRM-Analyzer that handles the SP and IdP as black-boxes by analyzing only the traffic visible within the browser. Their paper served as a model for our research. However, the BRM-Analyzer is rather passive (it analyzes the browser related messages), while OpenID Attacker acts as an IdP and as such, it can actively interfere with the OpenID workflow (e.g. create SSO tokens).

In 2014, Silva et al. [24] exploited an XML External Entity vulnerability in Facebook's parsing mechanism of XRDS documents during the discovery phase. The same attack is supported by the OpenID Attacker and is part of our evaluation. Simultaneously to our research, in 2014 Wang et al. [47] reported serious flaws in OAuth and OpenID, which are related to TRC.

SSO Security Tools. In 2013, Bai et al. [48] have proposed AuthScan, a framework to extract the authentication protocol specifications automatically from implementations. They found security flaws in several SSO systems. The authors concentrated on MitM attacks, Replay attacks and Guessable tokens. More complex attacks, like IDS or KC, cannot be evaluated. In the same year, Wang et al. [49] developed a tool named *InteGuard* detecting the invariance in the communication between the client and SP to prevent logical flaws in the latter one. Another tool similar to *InteGuard* is *BLOCK* [50]. Both tools should be able to detect Replay attacks and TRC. Since all HTTP messages between the adversary and the SP are valid and do not show abnormalities, neither *InteGuard* nor *BLOCK* is able to mitigate IDS, KC and XML External Entity. Zhou et al. [35] published on USENIX'14 a fully automated tool named *SSOScan* for analyzing the security of OAuth implementations and described five attacks, which can be automatically tested by the tool.

11. Lessons Learned

Trusted IdPs. When Microsoft introduced MS Passport, the first web SSO system, criticism concentrated on the closed nature of the system: only a single IdP at the domain `passport.com` was used. Subsequent approaches like MS Cardspace and SAML Web SSO allowed multiple IdPs, but still retained the idea that an IdP should only be run by trusted parties, and that trust relationship between an SP and an IdP should be established manually. With OpenID, "openness" for the first time became more important than "trustworthiness", and this resulted in new attack classes. *Lesson learned:* the establishment of trust should not be fully automated, if it is not backed up by solid cryptography (like e.g. in PKI scenarios).

Identities are Important. Attacks similar to TRC have been described before in the literature. For example Armado et al. discovered a bug in the Google SSO implementation where the identity of the target SP was omitted from the SAML assertion. Thus an assertion issued for (low-security) service A (controlled by the attacker) could be used to log into (high-security) service B. Including identities in protocol messages, and checking these values, is good engineering practice (e.g. in TLS certificate verification). *Lesson learned* from the TRC: checking identity of the SP is always important and should be enforced in any application.

References to Cryptographic Keys. KC exploits weaknesses in the association between the identity of the IdP, the key handle and the key value used for the signature verification. In OpenID the only connection between the key and the corresponding IdP is the association handle α. Unfortunately, the value of α can be freely chosen by *any* IdP. If the loading of the key occurs only on basis of α and without verifying the corresponding IdP, KC is applicable. Lessons learned: The identification of the correct cryptographic keys should be unambiguous. If keys are related to the identity of a communicating party, then this identity should be part of the key identifier (e.g., keys should be stored indexed by a pair $[IdP_{ID}, \alpha]$).

Multiple Equivalent Parameters. If two or more different parameters are used for the same purpose, then it is difficult to formally specify how to react if these two parameters have different semantics. This fact was exploited in the IDS attack (Strategy 2), which is only possible if two different strings are used as identifiers for the same entity. Similar problems have been reported in multi-layer messaging: E.g. in SOAPAction Spoofing, the SOAPAction can be specified in the HTTP and in the SOAP Header. By specifying two different values, inconsistent behavior can be triggered.

Complex Information Flow Specification. In many cases, developers of OpenID frameworks deviated from the specification, which resulted in a different, vulnerable message flow. It seems that the OpenID specification is not clear enough to unambiguously implement the desired message flow. It is an interesting open question how to formally specify the desired flow, such that computer-aided enforcement of this flow, or computer-aided checking of this flow, becomes possible.

12. Future Work

We showed that SSO protocols and implementations are a high-value attack target. Although there is a lot of research in the area of SSO [14], [29], [35] and OpenID [16], [45], [51], the number of vulnerabilities found is surprisingly high.

We believe that the concept of a malicious IdP is a threat to all open SSO protocols, thus future work includes applying the methodology developed in this paper to different protocols like OAuth, SAML an OpenID Connect.

We made the source code of OpenID Attacker public [13], encouraging researchers and penetration tester to use this tool to further improve security in SSO systems, and to adapt it to other protocols.

Acknowledgments

We would like to thank Juraj Somorovsky for fruitful discussions and his advice to extend our evaluation and Christian Koßmann for extending OpenID Attacker in his master thesis. The research was supported by the *German Ministry of Research and Education (BMBF)* as part of the `VERTRAG` research project and by the SkIDentity project of the German Federal Ministry of Economics and Technology (BMWi, FKZ: 01MD11030).

References

[1] C. Mainka, V. Mladenov, and J. Schwenk, "Do not trust me: Using malicious idps for analyzing and attacking single sign-on," in *IEEE European Symposium on Security and Privacy (Euro S&P)*. IEEE, 3 2016.

[2] J. Bonneau, "The science of guessing: analyzing an anonymized corpus of 70 million passwords," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 538–552.

[3] D. Silver, S. Jana, E. Chen, C. Jackson, and D. Boneh, "Password managers: Attacks and defenses," in *Proceedings of the 23rd Usenix Security Symposium*, 2014.

[4] Z. Li, W. He, D. Akhawe, and D. Song, "The emperor's new password manager: Security analysis of web-based password managers," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[5] Janrain. (2013) 2013 consumer research: The value of social login. Janrain. [Online]. Available: http://janrain.com/resources/industry-research/2013-consumer-research-value-of-social-login/

[6] T. Groß, "Security analysis of the saml single sign-on browser/artifact profile," in *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*. IEEE, 2003.

[7] T. Groß and B. Pfitzmann, "SAML artifact information flow revisited," Research Report RZ 3643 (99653), IBM Research, 2006.

[8] M. Jones, J. Bradley, M. Machulak, and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol," 2015. [Online]. Available: http://tools.ietf.org/html/draft-ietf-oauth-dyn-reg-29

[9] The OpenID Foundation (OIDF), *OpenID Connect Discovery 1.0*, The OpenID Foundation (OIDF) Std., February 2014. [Online]. Available: http://openid.net/specs/openid-connect-discovery-1_0.html

[10] ——, *OpenID Connect Dynamic Client Registration 1.0*, The OpenID Foundation (OIDF) Std., February 2014. [Online]. Available: http://openid.net/specs/openid-connect-registration-1_0.html

[11] A. Barth, C. Jackson, and J. C. Mitchell, "Securing frame communication in browsers," in *In Proceedings of the 17th USENIX Security Symposium*, 2008.

[12] OpenID, "Openid libraries," 2014, [online] http://wiki.openid.net/w/page/12995176/Libraries. [Online]. Available: http://wiki.openid.net/w/page/12995176/Libraries

[13] Christian Mainka and Vladislav Mladenov and Christian Koßmann, "Openid attacker, source code and executable," 2015. [Online]. Available: https://github.com/RUB-NDS/OpenID-Attacker

[14] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen, "On breaking saml: Be whoever you want to be," in *Proceedings of the 21st USENIX conference on Security symposium, Security*, vol. 12, 2012.

[15] specs@openid.net, "OpenID Authentication 2.0 – Final," Dec. 2007. [Online]. Available: https://openid.net/specs/openid-authentication-2_0.html

[16] R. Wang, S. Chen, and X. 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*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012.

[17] E. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "Oauth demystied for mobile application developers," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM – Association for Computing Machinery, November 2014. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=231728

[18] Dice Holdings, Inc., "SourceForge," 2014. [Online]. Available: https://sourceforge.net/

[19] ownCloud Inc., "ownCloud," 2014. [Online]. Available: http://owncloud.org/

[20] Drupal Team and D. Buytaert, "Drupal Open Source CMS," 2014. [Online]. Available: https://drupal.org/

[21] W3Techs – World Wide Web Technology Surveys, "Usage of content management systems for websites," 2014, accessed: 05.11.2014. [Online]. Available: http://w3techs.com/technologies/overview/content_management/all/

[22] Christian Mainka and Vladislav Mladenov, "Screencast: Attacking drupal 7 with key confusion," 2015, [online] https://www.dropbox.com/s/5np7hpujjyxn4fd/Attacking_Drupal7.mpg [MPEG2, 2:09min, 54mb]. [Online]. Available: http://bit.ly/drupalattack

[23] G. Steuck, "XXE (Xml eXternal Entity) Attack," OWASP, October 2002. [Online]. Available: http://www.securiteam.com/securitynews/6D0100A5PU.html

[24] R. Silva. (2014, 01) XXE in OpenID: one bug to rule them all, or how I found a Remote Code Execution flaw affecting Facebook's servers. http://www.ubercomp.com/posts/2014-01-16_facebook_remote_code_execution.

[25] Slashdot, "SlashDot.org," 2015. [Online]. Available: http://slashdot.org/

[26] OXID eSales AG, "Oxid security bulletins/2015-001," 2015. [Online]. Available: http://wiki.oxidforge.org/Security_bulletins/2015-001

[27] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra, "Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps," in *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008*, V. Shmatikov, Ed. Alexandria and VA and USA: ACM, 2008, pp. 1–10.

[28] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, G. Pellegrino, and A. Sorniotti, "From Multiple Credentials to Browser-Based Single Sign-On: Are We More Secure?" in *SEC*, ser. IFIP Advances in Information and Communication Technology, J. Camenisch, S. Fischer-Hübner, Y. Murayama, A. Portmann, and C. Rieder, Eds., vol. 354. Springer, 2011, pp. 68–79.

[29] S.-T. Sun and K. 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*. ACM, 2012.

[30] Egor Homakov. (2014, Februrary) How I hacked Github again.

[31] ——. (2013, Februrary) How we hacked Facebook with OAuth2 and Chrome bugs. [Online]. Available: http://homakov.blogspot.ca/2013/02/hacking-facebook-with-oauth2-and-chrome.html

[32] ——. (2013, March) OAuth1, OAuth2, OAuth...?

[33] Nir Goldshlager. (2013, February) How I Hacked Facebook OAuth To Get Full Permission On Any Facebook Account (Without App "Allow" Interaction). [Online]. Available: http://www.nirgoldshlager.com/2013/02/how-i-hacked-facebook-oauth-to-get-full.html

[34] ——. (2013, March) How I Hacked Any Facebook Account...Again! [Online]. Available: http://www.nirgoldshlager.com/2013/03/how-i-hacked-any-facebook-accountagain.html

[35] D. E. Yuchen Zhou, "Automated testing of web applications for single sign-on vulnerabilities," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/zhou

[36] E. Shernan, H. Carter, D. Tian, P. Traynor, and K. Butler, "More guidelines than rules: Csrf vulnerabilities from noncompliant oauth 2.0 implementations," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 239–260.

[37] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, "Explicating sdks: Uncovering assumptions underlying secure authentication and authorization," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534801

[38] D. Fett, R. Küsters, and G. Schmitz, "Paper: An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System," in *35th IEEE Symposium on Security and Privacy (S&P 2014)*. IEEE Computer Society, 2014.

[39] M. Corporation, "Browserid specification," https://openid.net/specs/openid-authentication-2_0.html", http://www.mozilla.org, Tech. Rep., 2011. [Online]. Available: https://github.com/mozilla/id-specs/blob/prod/browserid/index.md

[40] A. Dey and S. Weis, "Pseudoid: Enhancing privacy in federated login," *Hot topics in privacy enhancing technologies*, pp. 95–107, 2010.

[41] G. Elahi, Z. Lieber, and E. Yu, "Trade-off analysis of identity management systems with an untrusted identity provider," in *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*. IEEE, 2008, pp. 661–666.

[42] Z. A. Khattak, S. Sulaiman, and J. Manan, "A study on threat model for federated identities in federated identity management system," in *Information Technology (ITSim), 2010 International Symposium in*, vol. 2. IEEE, 2010, pp. 618–623.

[43] E. Tsyrklevich and V. Tsyrklevich, "Single sign-on for the internet: A security story," July and August 2007. [Online]. Available: https://www.blackhat.com/presentations/bh-usa-07/Tsyrklevich/Whitepaper/bh-usa-07-tsyrklevich-WP.pdf

[44] B. Newman and S. Lingamneni, "Cs259 final project: Openid (session swapping attack)," 2008. [Online]. Available: http://www.stanford.edu/class/cs259/projects/cs259-final-newmanb-slingamn/report.pdf

[45] S.-T. Sun, K. Hawkey, and K. Beznosov, "Systematically breaking and fixing openid security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures." *Computers & Security*, vol. 31, no. 4, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/compsec/compsec31.html#SunHB12

[46] B. van Delft and M. Oostdijk, "A security analysis of openid," in *Policies and Research in Identity Management*, ser. IFIP Advances in Information and Communication Technology, E. de Leeuw, S. Fischer-Hübner, and L. Fritsch, Eds. Springer Berlin Heidelberg, 2010, vol. 343, pp. 73–84. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17303-5_6

[47] W. Jing. (2014, 2 May) Serious security flaw in oauth, openid discovered. http://www.cnet.com/news/serious-security-flaw-in-oauth-and-openid-discovered/. Ph.D. student at the Nanyang Technological University in Singapore. [Online]. Available: http://www.cnet.com/news/serious-security-flaw-in-oauth-and-openid-discovered/

[48] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong, "Authscan: Automatic extraction of web authentication protocols from implementations," *NDSS, February*, 2013.

[49] L. Xing, Y. Chen, X. Wang, and S. Chen, "Integuard: Toward automatic protection of third-party web service integrations," in *Proceedings of 20th Annual Network & Distributed System Security Symposium*, 2013.

[50] X. Li and Y. Xue, "Block: A black-box approach for detection of state violation attacks towards web applications," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2076732.2076767

[51] P. Sovis, F. Kohlar, and J. Schwenk, "Security analysis of openid." in *Sicherheit*, ser. LNI, F. C. Freiling, Ed., vol. 170. GI, 2010. [Online]. Available: http://dblp.uni-trier.de/db/conf/sicherheit/sicherheit2010.html#SovisKS10

Appendix

This sections gives a more detailed description of the OpenID protocol flow.

1. Protocol

OpenID consists of three phases as shown in Figure 13. In the **discovery** phase, the SP collects information about $\mathcal{C}$'s requested identity (URL.ID$_\mathcal{C}$) and determines URL.$\mathcal{I}dP_\mathcal{C}$. In the **association** phase, the SP and the IdP establish a shared secret α intended to be used for signing and verifying the token. The **token processing** phase then includes the creation of the token by the IdP, its transport to the SP via $\mathcal{C}$'s UA, and its verification by SP. Figure 13 describes the OpenID login process more precisely:

(1.) $\mathcal{C}$ wishes to access a resource at the SP and enters his identity URL.ID$_\mathcal{C}$.

(2.) The SP then starts the discovery by requesting the document at URL.ID$_\mathcal{C}$.

(3.) A document containing URL.$\mathcal{I}dP_\mathcal{C}$ is returned.

(4.) Using URL.$\mathcal{I}dP_\mathcal{C}$, the SP can establish an association with the IdP. This is basically a Diffie-Hellman key exchange to establish a *shared secret* s. Additionally, the IdP freely chooses a string α that is used as a name for the association. It is used to reference the key material k derived from s on both sides, and has an expiration time. Note that in this phase, the SP and the IdP are directly communicating with each other, which means that a web attacker cannot interfere with this communication.

(5.) Afterwards, the SP has all necessary information to validate an OpenID token created by $\mathcal{I}dP_\mathcal{C}$. It responds to $\mathcal{C}$'s initial login request of Step (1.) and sends an *authentication request* containing URL.$\mathcal{I}dP_\mathcal{C}$, URL.$SP$ and optionally α.

(6.) $\mathcal{C}$ is redirected to URL.$\mathcal{I}dP_\mathcal{C}$.

(7.) If $\mathcal{C}$ is not yet logged in, he must authenticate to $\mathcal{I}dP_\mathcal{C}$.

(8.) $\mathcal{I}dP_\mathcal{C}$ creates a token t for $\mathcal{C}$ containing $\mathcal{C}$'s identity URL.ID$_\mathcal{C}$, its own URL address URL.$\mathcal{I}dP_\mathcal{C}$ and URL.SP. $\mathcal{I}dP_\mathcal{C}$ then generates a signature σ for t using the key referenced by α. Message (8) is called the *authentication response* and is sent as an HTTP redirect to URL.SP.

(9.) The authentication response is forwarded to the SP.

(10.)-(11.) The SP can optionally start a rediscovery, for example, if it has not cached the previous discovery, cf. Step (2.)-(3.).

(14.) If the signature is valid, the SP will map URL.ID$_\mathcal{C}$ to a local identity and respond accordingly to $\mathcal{C}$.

2. Direct Verification

Establishing an association is optional according to the OpenID standard. If the communication (4.) is missing, the

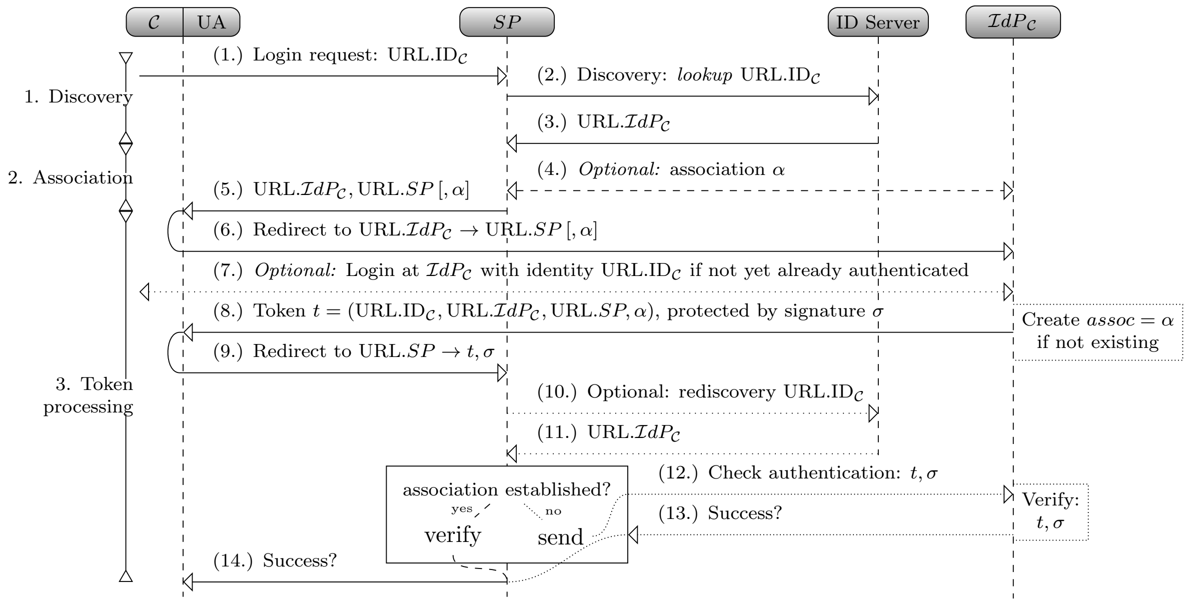

$\mathcal{C}$
UA
SP
ID Server
$\mathcal{I}dP_\mathcal{C}$
1. Discovery
2. Association
3. Token processing
(1.) Login request: URL.ID$_\mathcal{C}$
(2.) Discovery: *lookup* URL.ID$_\mathcal{C}$
(3.) URL.$\mathcal{I}dP_\mathcal{C}$
(4.) *Optional:* association α
(5.) URL.$\mathcal{I}dP_\mathcal{C}$, URL.$SP$ $[,\alpha]$
(6.) Redirect to URL.$\mathcal{I}dP_\mathcal{C} \to$ URL.SP $[,\alpha]$
(7.) *Optional:* Login at $\mathcal{I}dP_\mathcal{C}$ with identity URL.ID$_\mathcal{C}$ if not yet already authenticated
(8.) Token $t =$ (URL.ID$_\mathcal{C}$, URL.$\mathcal{I}dP_\mathcal{C}$, URL.$SP$, α), protected by signature σ
Create $assoc = \alpha$ if not existing
(9.) Redirect to URL.$SP \to t, \sigma$
(10.) Optional: rediscovery URL.ID$_\mathcal{C}$
(11.) URL.$\mathcal{I}dP_\mathcal{C}$
association established?
yes
no
verify
send
(12.) Check authentication: t, σ
Verify: t, σ
(13.) Success?
(14.) Success?

Figure 13: The OpenID protocol flow.

authentication request does not contain α, and no *shared secret* was established with $\mathcal{I}dP_\mathcal{C}$. In this case, the IdP generates a fresh key and signs the token with it. In this case, the SP will not be able to verify the authenticity of the token by itself. Instead, it must send the token directly to the IdP in Step (12.), and accepts the result of the verification from Step (13.).

3. Discovery in Detail

To receive URL.$\mathcal{I}dP_\mathcal{C}$ in Step (3.), the SP fetches the document at URL.ID$_\mathcal{C}$ (e.g. http://myserver.org). This can be either an HTML or an XRDS document. Listing 1 shows a minimal HTML document.

```
<html><head><title/>
<link rel="openid2.provider"
      href="https://myidp.com/" />
</head><body/></html>
```

Listing 1: Minimal HTML discovery document.

The element **<link/>** contains URL.$\mathcal{I}dP_\mathcal{C}$ within the `href` attribute. XRDS documents contain the same information, but stored in XML data format.

Note that Step (5.) of the protocol does not contain URL.ID$_\mathcal{C}$. This is not necessary, since $\mathcal{C}$ must authenticate to $\mathcal{I}dP_\mathcal{C}$. Consequently, $\mathcal{I}dP_\mathcal{C}$ knows the value of URL.ID$_\mathcal{C}$. However, the discovered document in Step (3.) allows optionally to include a second "local" identity URL.ID$_\mathcal{C}{}^*$ (the value of the href attribute in Listing 2):

```
<link rel="openid2.local_id"
      href="https://myidp.com/bob" />
```

Listing 2: $\mathcal{C}$'s identity stored in an HTML document.

If this is the case, steps (5.) and (6.) will include this value as well and $\mathcal{I}dP_\mathcal{C}$ is asked to use URL.ID$_\mathcal{C}{}^*$. This is, for example, useful if $\mathcal{C}$ owns multiple IDs at $\mathcal{I}dP_\mathcal{C}$.

4. Attacking Owncloud

The attack on ownCloud is depicted in Figure 14.

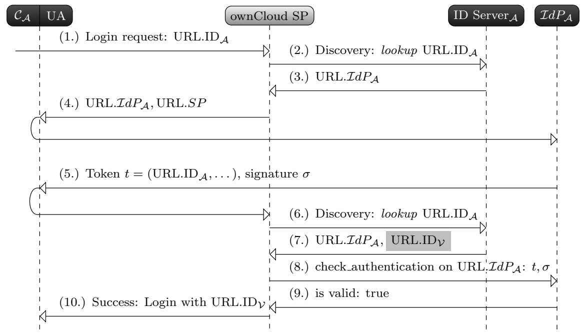

$\mathcal{C}_\mathcal{A}$
UA
ownCloud SP
ID Server$_\mathcal{A}$
$\mathcal{I}dP_\mathcal{A}$
(1.) Login request: URL.ID$_\mathcal{A}$
(2.) Discovery: *lookup* URL.ID$_\mathcal{A}$
(3.) URL.$\mathcal{I}dP_\mathcal{A}$
(4.) URL.$\mathcal{I}dP_\mathcal{A}$, URL.$SP$
(5.) Token $t =$ (URL.ID$_\mathcal{A}, \ldots$), signature σ
(6.) Discovery: *lookup* URL.ID$_\mathcal{A}$
(7.) URL.$\mathcal{I}dP_\mathcal{A}$, URL.ID$_\mathcal{V}$
(8.) check_authentication on URL.$\mathcal{I}dP_\mathcal{A}$: t, σ
(9.) is valid: true
(10.) Success: Login with URL.ID$_\mathcal{V}$

Figure 14: The ID Spoofing attack on ownCloud: The attacker's ID server returns URL.ID$_\mathcal{V}$ on the second discovery. ownCloud uses this identity value for the login instead of the identity provided within the token.

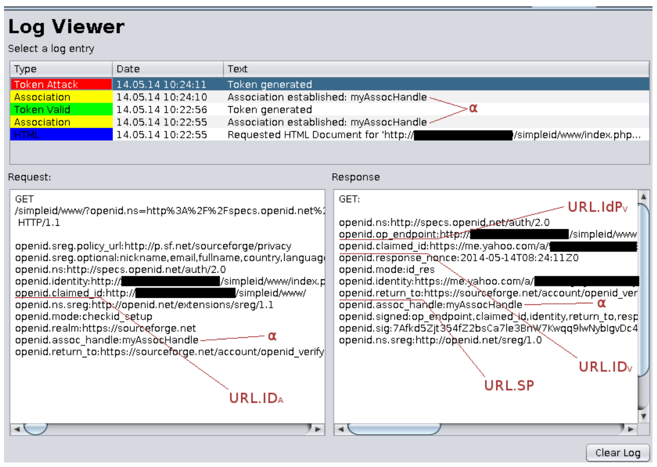
5. Log Inspection

Figure 15 shows the log view of OpenID Attacker.

6. Report

OpenID Attacker outputs a detailed report as shown in Figure 9.

Table 2 lists the notations used in this paper.



```
Log Viewer
Select a log entry

Type          Date               Text
Token Attack  14.05.14 10:24:11  Token generated
Association   14.05.14 10:24:10  Association established: myAssocHandle
Token Valid   14.05.14 10:22:56  Token generated
Association   14.05.14 10:22:55  Association established: myAssocHandle
HTML          14.05.14 10:22:55  Requested HTML Document for 'http://[redacted]/simpleid/www/index.php...
```

```
Request:

GET
/simpleid/www/?openid.ns=http%3A%2F%2Fspecs.openid.net%2
 HTTP/1.1

openid.sreg.policy_url:http://p.sf.net/sourceforge/privacy
openid.sreg.optional:nickname,email,fullname,country,languag
openid.ns:http://specs.openid.net/auth/2.0
openid.identity:http://[redacted]/simpleid/www/index.p
openid.claimed_id:http://[redacted]/simpleid/www/
openid.ns.sreg:http://openid.net/extensions/sreg/1.1
openid.mode:checkid_setup
openid.realm:https://sourceforge.net
openid.assoc_handle:myAssocHandle
openid.return_to:https://sourceforge.net/account/openid_verify
```

```
Response

GET:

openid.ns:http://specs.openid.net/auth/2.0
openid.op_endpoint:http://[redacted]/simpleid/www
openid.claimed_id:https://me.yahoo.com/a/[redacted]
openid.response_nonce:2014-05-14T08:24:11Z0
openid.mode:id_res
openid.identity:https://me.yahoo.com/a/[redacted]
openid.return_to:https://sourceforge.net/account/openid_ver
openid.assoc_handle:myAssocHandle
openid.signed:op_endpoint,claimed_id,identity,return_to,resp
openid.sig:7Afkd5Zjt354fZ2bsCa7le3BnW7Kwqq9lwNyblgvDc4
openid.ns.sreg:http://openid.net/sreg/1.0
```

Figure 15: IDS attack on Sourceforge. The OpenID Attacker *log viewer* window lists all exchanged OpenID messages. The Screenshot shows that the SP requests a token for URL.ID$_{\mathcal{A}}$, but the tools ignores the wish and responds with a token for URL.ID$_{\mathcal{V}}$.

Notation	Explanation
URL.ID	A URL representing a user's *login name*
URL.ID$_{\mathcal{C}}$	A URL representing $\mathcal{C}$'s *login name* at URL.$\mathcal{I}dP_{\mathcal{C}}$
URL.ID$_{\mathcal{A}}$	A URL representing $\mathcal{A}$'s *login name* at URL.$\mathcal{I}dP_{\mathcal{A}}$
URL.SP	The URL of the SP, e.g. http://mysp.com
URL.$SP_{\mathcal{A}}$	The URL of the attacker $\mathcal{A}$ controlled SP, e.g. http://sp.attacker.com
URL.$\mathcal{I}dP$	The URL of the user's IdP, e.g. https://www.google.com/accounts/o8/ud
URL.$\mathcal{I}dP_{\mathcal{C}}$	The URL of $\mathcal{C}$'s IdP, e.g. https://www.google.com/accounts/o8/ud.
URL.$\mathcal{I}dP_{\mathcal{A}}$	The URL of the attacker $\mathcal{A}$ controlled IdP, e.g. http://idp.attack.com.
t	The OpenID token, containing at least URL.ID, URL.SP and URL.$\mathcal{I}dP$.
σ	The signature value for token t.
α	The value α is used to identify the key to verify (t, σ). Note that α is just a reference value to the key and does not contain any key material. For the attack on Drupal, we also used β, because there are two different associations.

TABLE 2: List of notations used in this paper.