OpenID Connect: Formal Analysis, Verification and Attacks Classification

Ghada Zamzmi, Hernan Palombo
Department of Computer Science and Engineering,
University of South Florida
ghadh@mail.usf.edu, hpalombo@mail.usf.edu

Abstract

The ubiquity of the internet and the popularity of web and mobile applications has exposed the need for protocols that can identify users securely. Single sign-on (SSO) protocols have proved many benefits and have now become the standard for authentication on social platforms. OpenID Connect provides authentication and authorization in a single protocol, alleviating application developers from implementing in-house security mechanisms that could potentially be flawed. It relies on identity providers that users normally trust; it is based on open standards; and it aims to provide a solution to the *password fatigue* problem. However, the intrinsic web nature of this protocol makes it vulnerable to a large set of well-known attacks. Although the current version of the protocol claims to be secure, no other existing work (that we are aware of) has modeled and formally analyzed this protocol yet. In this study, we formalized the initial protocol specification by creating a model using AVISPA, a well-known model checker for security protocols. We ran the verification and analyzed the results. Finally, we classified known security vulnerabilities and set the grounds for further research.

Keywords: Model-checking Security Protocols, SSO, Attacks Classification

1 Introduction

The rapid growth of cloud computing along with the ubiquity of web applications has increased the number of passwords that are required to authenticate users and safeguard web resources. The average web-user has about 25 password-protected accounts and logs in using 8 passwords per day [9]. Remembering an excessive number of passwords leads to a *password fatigue* and encourages users to adopt poor security habits. *Single sign-on (SSO)* is one solution that can be adopted to address this problem. It gives the user the ability to sign in once, and gain access to several resources or websites without the need to sign in again into each of them. This solution has significant benefits: it reduces the number of times the users have to input their passwords, alleviates websites from implementing ad-hoc security mechanisms for authentication, and lowers the IT costs [18, 21]. However, the cost of identity theft is extremely high because an attacker could potentially gain access to all users applications, and the data stored in these applications.

SSO works through the interaction between three parties, a *user*, an *authorization server* (aka. *identity provider IdP*), and a *client* (aka. *relying party RP*). Figure 2 on page 3 shows an overview of OpenID interactions. With SSO, the end-user can log in into multiple clients (i.e. relying parties/websites) using the same id that is provided by the server. Figure 1 shows an example of logging in into Spotify's music service using Facebook as the identity provider.

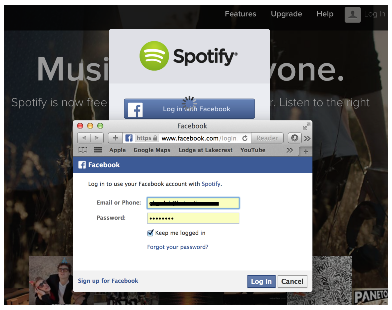


Figure 1: Users can login to Spotify (RP) using Facebook credentials (IdP)

Although SSO protocols, such as Kerberos, and Smart Cards have been around for decades, they have gained popularity recently as a result of social networking, cloud computing, and mobile proliferation. According to a survey conducted on 600 participants [20], 66% of the web-users prefer to login using SSO.

OpenID is one of the most widespread SSO protocols. According to OpenID Foundation, there are more than one billion OpenID user accounts, and over 50,000 websites accepting OpenID for logins [10]. Several security issues have been associated with OpenID since it was created in 2005. In 2007, a new version OpenID 2.0 was released with some enhancements to rectify the security issues in the previous version [10]. Nevertheless, empirical and formal analysis studies of the protocol still show a lack of security guarantee, and various privacy issues, as we will discuss in the related works Section. **OpenID Connect 1.0 (OIDC)**, the most recent version of OpendID, was released in February 2014, only a few days before we started this research [6]. It is an identity layer built on top of the OAuth 2.0 protocol. It integrates authentication and authorization into a single protocol, and it is based on OpenID 2.0, and the OAuth 2.0 [7, 10, 17]; an overview of OAuth 2.0 will be presented in the related works Section.

Authentication refers to the process of identifying the end-user (i.e, verify that someone is really who s/he claims to be). *Authorization* refers to the rules that define who is allowed to access what (i.e, it defines the level of access). Authorization should always occur after authentication. With OIDC, any clients (e.g., Spotify) can authenticate the end-user based on the id provided by the authorization server (e.g., Facebook), and gain the authority to obtain her/his profile information (i.e. claims; e.g. first/last name, contact_info, and profile image) in an interoperable, and REST-like manner [7].

1.1 Motivation

OpenID has been rapidly adopted by a huge number of clients, even though there are serious security, and privacy issues surrounding it. Besides OpenID's security considerations presented in the OpenID specification (e.g., phishing, reply, and DoS attack), other studies have reported several additional vulnerabilities. Although the new version of OIDC claims to address these issues, it has not been verified yet. Thus, we believe it is imperative to formalize the new version of the protocol and verify its security claims; this is our main objective.

1.2 Contributions

Our contributions can be summarized as follows:

1. We have created an OpenID Connect 1.0 protocol model using the Formal analysis High Level Protocol Specification Language (HLPSL) and verified this model using the Automated Validation of Internet Security Protocols and Application (AVISPA) model-checking engine. To the best of our knowledge, our paper is the first paper that formalizes OpenID Connect and verifies its security claims in a pessimistic network environment.
2. We classified the existing OpenID security attacks into four main categories based on the attack space vectors, considering the roles that could be impersonated by the attacker. The categories are: (a) a malicious user, (b) a malicious client, (c) a malicious server, and (d) a compromised communication channel. We believe that this classification can provide a better understanding of the protocol's weaknesses, and may help the OpenID community to come up with effective countermeasures.

1.3 Roadmap

The rest of the paper is organized as follows: Section 2 describes the OpenID protocol; Section 3 explores related works and previously discovered security issues. In Section 4 we give an overview of our approach and define the adversary model. In Section 5 we present a formalization of the protocol and explain our implementation using HLPSL. Section 6 classifies the existing security attacks into four main categories based on the role that could be compromised. Section 7 analyses our results, discuses future works, and concludes the paper.

2 OpenID Connect

At a high-level, OpenID Connect consists of four main phases[1]: login request/redirect, authentica-

[1]The authorization server discovery and client association phases occur before the four phases described. However, we have decided not to model them (see Section 4).

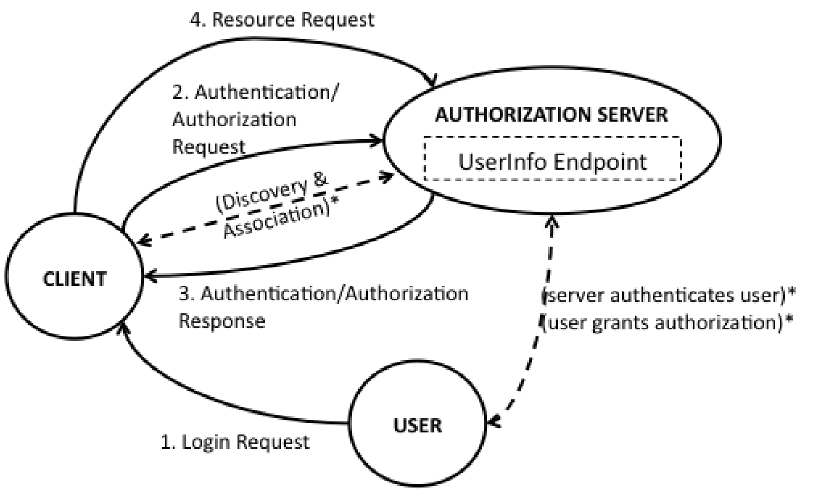

4. Resource Request
2. Authentication/ Authorization Request
AUTHORIZATION SERVER
UserInfo Endpoint
(Discovery & Association)*
CLIENT
3. Authentication/Authorization Response
(server authenticates user)*
(user grants authorization)*
USER
1. Login Request

Figure 2: High-level flow of OpenID Connect. ** Communication denoted with a dashed line is not specified in the core specification. Thus, it was not considered for this study.*

tion request, authentication response, and claims request/response [7]. These phases, which are illustrated in figure 2, are described below.

Phase 1: Login Request/Login Redirect

1.1 The user **U** should provide her/his *id* to the client **C**; The client **C** then applies normalization rules on this *id* in order to discover the authorization server's **A** location. User *id* has different types that include:

   i. URL identifier (`https://example.com/ghada`)

   ii. Email address identifier (`ghada@example.com`)

   iii. Hostname and port identifier (`example.com:8080`)

1.2. The client **C** redirects the user **U** to the authorization server **A** where s/he gets authenticated.

Phase 2: Authentication Request

2.1. In order to authenticate the user **U**, the client **C** should prepare an authentication request to send it to the authorization server **A**. This request is associated with some parameters such as *the client id*, and *the redirection URI*; redirection URI determines the location to where the response should be sent back.

2.2. . The authorization server **A** authenticates the user **U** by presenting a login prompt where **U** can enter her/his credentials.

2.3. After **U** gets authenticated, the authorization server **A** should present a user **U** with a dialogue that lists what is being consenting to. The user can grant/deny to release her/his claims.

Phase 3: Authentication/Authorization response

3.1. Once the user authorization is obtained, **A** redirects **U** back to the client **C** based on its URI redirection. The clients URI must match one of the valid redirection URIs values pre-registered at the authorization server **A**, and it should use https scheme. The authorization server **A** should also send with a redirection URI two other parameters, *the code*, and *the state*. The code is used to create the access token request. The state parameter is used to verify that the URI's state value is equal to the state value in the authorization request.

3.2. The client **C**then creates a token request by presenting its authorization code to the authorization server **A**. **A** validates the token request by verifying the validity of the authorization code. It then returns a successful response that has id/access tokens or an error response.

3.3. The client **C** receives a response with id/access tokens plus other parameters, such as expiration time of the access token, and a refresh token. Id token is a security token associated with the user authentication session. It has several parameters such as, issuer identifier, nonce to associate the client session with token, time when the token is issued, and its expiration time. The access token is a token that gives the client the ability to access and retrieve users claims from user's protected resources (aka. *UserInfo endpoint*).

Phase 4: UserInfo Request/Response

4.1. The client **C** sends a request to access a user's claims using her/his access token. Users' claims reside in the UserInfo endpoint.

4.2. The *UserInfo* endpoint validates the access token and returns the requested claims if it is valid, or an error otherwise. The client **C** must verify that the response is returned from the intended endpoint by checking its certificate. All the communication between the authorization server and the client, the user and authorization server must be secured using SSL connection [7].

3 Related works

3.1 Known Security Issues

OpenID 2.0 specification lists several possible security risks such as phishing attack, IdP masquerade by MITM, reply attack, and DoS attack (which attempt to exhaust the RPs and IdPs resources) [11]. Nash et al. [8] also provides a summary of most of the existing OpenID security issues. Tsyrklevich [25] lists some possible attacks with the OpenID protocol. These attacks can be summarized as follows,

1. Since an OpenID identifier is simply a URL, a malicious user can attack a relying party (RPs) by tricking it to request some malicious URL (e.g. to do a port scan).

2. The connection between IdP and RP in OpenID 1.0/2.0 is performed using Diffie-Hellman shared key approach, which might be vulnerable to MITM. The attacker can trick the RP by impersonating the IdP, and sign the authentication assertions on behalf of it.

3. A malicious RP can redirect the user to a fake IdP where the user gets phished.

4. A malicious IdP can violate user's privacy via the *return_to* parameter that gives an attacker the ability to track all websites the user has logged into, and know what kind of applications s/he used.

5. After the authentication, IdP should redirect the authenticated user back to RP; in this phase, an attacker can perform a replay attack by sniffing the wire and logging in to the RP as a victim user.

6. Once the user is logged in, the attacker can use cross-site request forgery to perform some attacks against the user.

Another paper by Barth et al. [1] describes session initialization's vulnerabilities with OpenID that occur as a result of not binding the OpenID session with user browser, and proposes a defense approach to prevent them. Wang et al. [12] present the results of an extensive empirical experiment to analyze the security flaws of SSO protocols.The study focuses on attacks that may come from the users web browser, and shows that SSO protocols have some critical logic flaws that violate the purpose of authentication and allow an attacker to sign in as the trusted user. For example, the ability of the attacker to access and forge the HTTP fields that carry the authentication token. These flaws have been reported to affected companies, such as Google and Yahoo, and have been fixed. As the paper pointed out, a collaborative effort from the SSO community must be made to investigate and analyze the SSO protocols. As a result, implementers could gain a better understanding of SSO security challenges.

Despite the fact that some studies have analyzed and formalized the OpenID protocol, many of these studies are based on strong assumptions. Sun et al. [22] analyzed and formalized OpenID model by assuming that the user, RP, and IdP are always trustworthy. Notwithstanding, the results of analyzing and evaluating implementation models of 132 clients show that there are three main vulnerabilities with OpenID protocol:

1. OpenID did not bind the authentication message with the user browser,
2. OpenID did not guarantee the integrity of the authentication request,
3. OpenID did not guarantee the authenticity of the authentication request.

The study has proposed, and evaluated two countermeasures to defeat these weaknesses.

OpenID recycling is another issue that violates a user's privacy. In the OpenID community forum, Allen [24] discuses in detail the problem of OpenID recycling and its consequences. With OpenID, the identity provider can recycle user ids that are belonging to inactive accounts without informing the relying party about the change; this could give the new owner the ability to access previous owner's data, and violate the user privacy. We discuss an OpenID recycling solution in the classifcation section.

Several papers [15, 19, 25, 27] classify OpenID's issues based on different factors. Delft et al. [26] categorize the OpenID security issues into four main categories based on:

- the goal of OpenID,
- its implementation in practice,
- its specification, and
- the underlying infrastructure of the Web.

The paper provides a clear distinction of vulnerabilities that are related to OpenID. A list of possible solutions, such as, changing OpenID protocol, and its trust framework, are also presented.

Related work demonstrates that the security of OpenID (in particular) and SSO protocols (in general) seems worrisome and more extensive security analysis is required. In this paper, we are going to provide a different classification that is based on the attack vectors corresponding to the different OpenID actors. More discussion can be found in the classification section.

3.2 OAuth 2.0

OAuth 2.0 is an open authorization protocol that gives an application the ability to read the user's data from another application [30] (i.e. enables other applications to access a user information in a specific server). For example, a user tries to sign in to a specific web application using a Facebook account that supports the OAuth protocol, then this application now can access all the user data on Facebook.

OAuth 2.0 is used with OpenID Connect for authorization; it gives an application the ability to access the authenticated user data, and retrieve her/his information from a UserInfo endpoint.

3.3 Privacy

Privacy can be defined as the right to choose non-disclosure of certain personal information [12]. In OIDC context, it implies that authorization servers and clients must be responsible and ethical in the use of a customers private information. Misuse of users information can be classified into two main groups. First, we consider the case of authorization servers tracking user activities for commercial or political purposes. The former can be used to advertise products based on users behavior. The latter can have further significant implications, e.g. in countries where the government may force authorization servers to provide such information for prosecution of opposing parties. Next, we consider leaking protected information. There is no guarantee that the client will make proper use of the users information. For example, a client may store users information for later use even after the user has revoked the authorization. There is no practical way of providing a complete solution for this issue, although OIDC attempts to alleviate potential consequences by requesting the client to refresh access tokens periodically.

4 Approach

To gain confidence that OIDC enforces a solid security mechanism, we decided to create a model of the protocol's specification, and use a model-checker to assert that fundamental security properties are satisfied. The decision of using verification over traditional testing and simulation is based on the higher level of confidence that formal methods can provide. Moreover, we chose model checking over other forms of verification mainly because of its "push-button" intrinsic characteristic that allows us to obtain significant results without the complexity of the mathematical proofs. We chose to specify the protocol's model in HLPSL (High-Level Protocol Specification Language), and run it using AVISPA [16], a state-of-the-art model-checker for security protocols. AVISPA takes an input model file in HLPSL, translates it to IF (intermediate form) language, and then connects to one of the different backends that come with the tool. In HLPSL, each *role* is defined as a set of variables and a finite state automaton that represents the role's behavior; a *session* is a composition of role instantiations.

This approach, however, has some limitations, which are: (a) we are modeling the protocol specification and not the actual implementation; (b) even if we were modeling an implementation, it would be unfeasible to create an exact model that could capture exact behavior at the implementation level (such model would cause a state explosion problem, a well-known issue in model-checking [4]) ; finally, (c) we would not be able to define a complete set of attacks since specifying this model is a problem with unknown solution.

Furthermore, due to time limitations of our research, we decided to model the authentication phase only (phases 1-3 as described in Section 2). We believe in this way we can cover the core of the protocol and leave other phases for future work.

Nevertheless, we believe that our approach is still better than studying individual attacks because it does capture the behavior of the system and its environment as a whole, and it allows modeling a larger set of attacks in a single model. Additionally, it allows protocol's designers to think about security issues and gain a deeper understanding of the different interactions and implications.

4.1 Adversary model

We modeled the attacker using standard assumptions used in similar studies, which implies that the attacker has a full control of the network. This attacker model is normally called the *dy* (Dolev-Yao) attacker; DY attacker is a strong attacker who has the ability to overhear, intercept, and synthesize every message. This attacker is also has the ability to cause any party to apply its operators, and instructions at any time (e.g. the attacker can start new session at any party) [3, 6]. We believe that a *dy* attacker is powerful and it fits as a good model for our protocol's security analysis.

4.2 Assumptions

In our model, we assume that the client and authorization server have already done the discovery and association steps in the protocol flow. We do this mainly because it simplifies the model making it apt for verification using AVISPA. A more complete model requires a greater refinement effort and we leave that as future work.

Another important implementation decision was to assume encrypted channels established by the authorization server with the user and the client, respectively. Although this may not be the case in some implementations, it was a reasonable starting point for our project, since it allowed us to gain greater confidence that our model was right. Later, we could remove some of these assumptions and check the properties again.

5 Modeling OIDC

In this section, we present an analysis of the OIDC specification, describe the abstraction process we used to model the protocol, and explain how we implemented it using HLPSL. The complete implementation code is listed in Appendix B. The OIDC specification defines three ways in which the protocol can be implemented, i.e. an authorization flow (AF), an implicit flow (IF), and a hybrid flow (HF). The decision of what flow to implement should be based on the security assumptions that the client implementer can make about the communication between itself and the server (i.e. secured vs. non-secured channel), and the intended use of the protocol (e.g. authentication, authorization, or both). For this study, we decided to consider only the authorization flow, which the specification describes as the most common and recommended way to implement this protocol.

To model the protocol, we define 3 main roles: a *user U*, a *client C*, and an *authorization server A*. The behavior of each role is specified as a state transition system with synchronized communication over predefined channels (shown in Appendix A). We model the interaction over shared variables by defining a fourth role, *session S*. We instantiate the roles previously defined and introduce the attacker in the *environment* role. The security properties are specified within each role and enforced at the end of our model under the *goal* declaration.

A successful authentication session consists of the exchange of a total of six messages. Figure 3 shows a diagram of the interactions between the different roles.

Next, we defined variables that we consider absolutely necessary in the model:

- **Uid** - a user id
- **Cid** - a client id
- **Kac** - a shared key between the server and the client
- **Kau** - a shared key between the server and the user
- **T** - an authorization token

Additionaly, we included the redirect URIs from the user to the server (Suri) and to the client (Curi) respectively. Subsequently, we define this protocol in Alice-Bob notation, and add the variables that we just defined as the contents of each message:

1. U → C : *LoginReq, (U.Uid.Nlr)*
2. C → A : *AuthReq,* $(Cid.\{Uid\}\_Kac)$
3. C → U : *LoginRedir, (Cid.Suri)*
4. U → A : *UserLogin,* $(Uid.\{Cid.Upwd\}\_Kau)$
5. A → C : *AuthResp,* $(\{Cid.Uid.T\}\_Kac)$
6. A → U : *SuccessRedir,* $(\{Cid\}\_Kau)$

A login request **LoginReq** must include the *Uid* that the client will send to the server to identify the user. As discovered by a previous study [22], including a nonce **Nlr** is also required to prevent replay attacks.

Next, the client sends its own id *Cid* and the *Uid* to the server for authentication in the **AuthReq** message. *Uid* shall be encrypted to prevent a session swapping attack.

It follows a **LoginRedir** sent by *C* to redirect *U* to *A*. At this step, the user sends its credentials to the server. Although not specified in the protocol, to prevent a session swapping attack, it is recommended that the *Cid* also gets encrypted along with the user password. Similarly, the authentication response **AuthResp** from the server must encrypt the *Cid* and *Uid* along with the acccess token *T*. Finally, the successful authorization redirect **SuccessRedir** encrypts the *Cid* to prevent a phishing attack in the last step of the protocol.

In AVISPA, goals can be classified the goals in two main categories, *secrecy* and *authentication.* Secrecy means that some variable needs to be known only by the parties specified in the property. If the intruder learns the variable, the property is violated [23]. Authentication is enforced when role A can assert that role B is the identity that s/he claims to be based on the value of a communicated variable.

In this version of the model we decided to verify two main security goals, (a) a strong authentication[2] of the user and client by the authorization server, and (2) the secrecy of the token throughout a protocol session.

5.1 Results

We ran this model in AVISPA, using two backends. First, we used the option –satmc to select the SAT-based bounded model checker [5], and then we used –satmc for the on-the-fly explicit model checker [2]. In both cases, the result of the verification showed that the model checker found no attack traces, giving us confidence that the model satisfies the specified security properties.

[2]strong authentication allows to check for replay attacks [16]

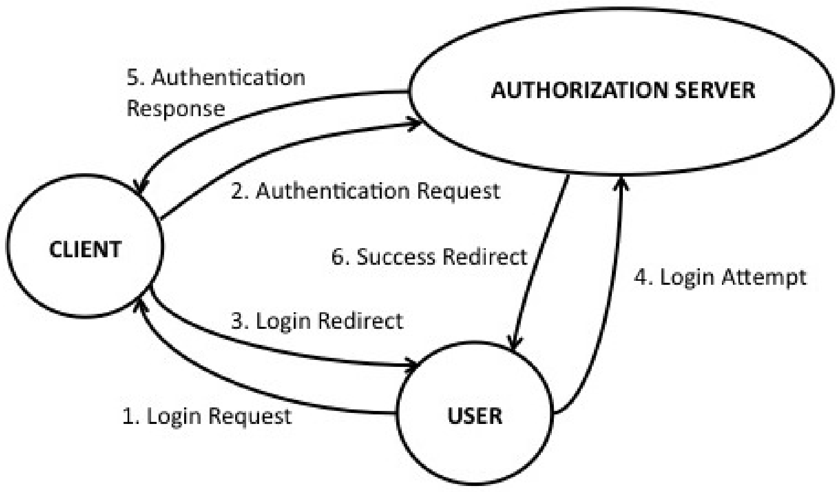


Figure 3: Simplified session model for verification in AVISPA

6 Classification of Security Issues

Formalizing all the possible security attacks related to the protocol seems unfeasible. Therefore, we limit ourselves to an attempt to classify attacks into large sets, according to the attack space from which the attack is launched. The most powerful attacker we can describe would have physical access to all three main actors machines. However, this model is too comprehensive and unrealistic for a distributed protocol like OpenID. If we were to consider a more realistic yet extremely powerful attacker, we would have to consider an attacker that has full control of the network and physical access to one of the party's machine. In this scenario, there is still no hope of enforcing any of the security goals of this protocol. For instance, if the attacker had full control of the user's machine, s/he could use a key logger to store the user's password and retrieve it trivially. A less powerful attacker could take control of a user's browser and store an access token, which s/he could use to obtain user's information from the resource server.

An attacker impersonating the authorization server would be a case of phishing attack. Phishing attacks have been studied and they are still extremely hard to prevent [8]. Moreover, an attacker impersonating a client may not be able to guess a user's password. However, s/he may be able to obtain authorization to some of the user's protected resources. This attack could still cause significant damage, since some of the claims that the attacker could request could include, for instance, sensitive financial information.

In this section, we combine the existing security issues surrounding OpenID into four distinct categories in order to provide a clear overview of these issues, and help the OpenID community to come up with effective countermeasures. The classification, which is based on the space that could be impersonated by the attacker, has four categories: a malicious user, a malicious client, a malicious server, and a compromised channel between the protocol's roles. These categories, and its associated attacks are discussed in detail in the next subsections.

6.1 A malicious User

6.1.1 A malicious OpenID Identifier

As we mentioned in Section 1, the user's identifier can have one of three different types (i.e. an email, a hostname or a URL. At the beginning of the protocol, the users should sign in using their identifier, which is used by a client to discover the location of the server. Since the user can sign in using any URL identifier, s/he can attack any client by providing a malicious one [25]. Different attacks can occur as a result of using malicious URLs to sign in to a client. Some of these attacks are presented below.

- **The Port-Scanning Client** Here, the client is used as a port-scanner by an attacker who can enter a malicious URL with an arbitrary port (e.g. `http://www.example.com:3`) and cause the client to port-scan any host on the internet. Additionally, the attacker may trick the client into scanning an internal host, such as: `https://192.168.1.15/internal/authip=1.1.1.1`. This could give the attacker the ability to access the client's internal scripts, and execute unauthorized actions [25,26].
- **Denial of Service (DoS)** Since the users are free to sign in using their URL identifier, a malicious user could exploit the URL identifier to perform DoS attack on the client in two different ways [8,25,26]. First, s/he could saturate the client with a large number of login requests in an attempt to make it unavailable. Second, s/he could also attack the client by using a URL identifier with large or endless files (e.g. `http://www.trap.com/cgi-bin/hang.pl`). This attack depends on the fact that a client does not limit the time that is allowed for a login request. Thus, limiting the number of requests, the request's time and data are obvious solution that should be considered by a client to protect itself [25].

Since the malicious user can use the identifier to perform different attacks against the client, requiring some additional standardization of the URL identifier (e.g. reject any URL that contains local numbers or IP addresses) can mitigate this attack [26]. In order to mitigate this attack, OIDC specification states that the URL should use a https scheme.

6.1.2 Cross-site Request Forgery (CSRF)

Once the server authenticates the user, s/he gets permission to access any clients unless s/he signs out; This can be exploited to perform cross-site scripting attacks. The attacker can run a script to steal the user's authorization token that may be stored in a cookie file, and use them sign in to user's different websites [8,25,26]. The attacker can also perform cross-site request forgery CSRF attacks against clients where the user is logged in. Bart et al. [26] suggests some solutions to these attacks. OpenID Connect recommends binding a state parameter, which is used to maintain the state between the request and callback with a browser's cookie. It also states that the server must implement appropriate measure against CSRF whenever it interacts with the user [7].

6.2 A malicious Client

6.2.1 Phishing

OpenID's redirection phase between the client and the authorization server can be exploited to phish the user, and steal her/his credentials [3]. There are different scenarios in which the OpenID system can get phished. One scenario occurs as a result of impersonating the client by an attacker [25,26]. A malicious client can redirect the user to a phished server page, which is identical to the official page, where the users get tricked to enter their credentials.

Several anti-phishing techniques have been proposed to mitigate the attack and its consequences [8,26,27]. Bart et al. [26] discussed two main solutions to prevent phishing attempts. The first solution is a cookie-based solution, which works only from a single web browser. The authorization server can add to its login-page a personal icon that gets activated only when the user visits the page from an identified browser. Another solution is installing anti-phishing browser tools to help the user noticing potential attacks by checking certificate and domain addresses. These tools include: the address bar highlighting and OpenID specific add-ons[4]. OpenID Wiki page [27] also provides a list of recommendations for users to protect them from this attack. For example, the users can pre-authenticate with their server before signing in to

[3]Typically, the phishing attack requires a direct contact between the attacker and the user via a message or link, and occurs as a result of replying to a suspicious link/message with the user's credentials; however, the redirection phase with OpenID makes it possible to create a less questionable attack. As a result, the term phishing is normally used in OpenID literature.

[4]such as VeriSign's *SeatBelt3*, *Sxipper4* and Microsoft's *Identity Selector5*

any client; thus, the user does not need to re-enter the credentials again into a fake login-page which occurs as a result of redirecting the user to the server by a malicious client.

6.2.2 Realm Spoofing

OpenID's redirection phase that sends the user back from the server to the client can also be exploited to phish the user. The attacker who controls a malicious client can substitute the authentication request parameters such as the realm, and *return_to* with an *openid.realm* set to a trusted domain and the *return_to* pointing back to a fake client page [8, 25, 26]. Since the identity provider does not validate these parameters, the users will believe that they are signing into a trusted client when in fact they are signing into a malicious client [26].

To prevent the realm spoofing attack, the specification of the OpenID Connect [7] states that the *redirection_uri* parameter, which has the URI to which the response should be sent, must be included in the authentication request, and it must match one of a pre-registered, and trusted URIs at the authorization server. The response must also be encrypted. As a result, the new protocol guarantees that only the trusted clients can use OpenID-login feature in their page.

6.2.3 Session Swapping

OpenID does not suggest any mechanism to bind the authentication session to the user's browser [11]. Thus, the attacker can use her/his machine to initialize an authentication session at the client; this session gets authenticated at the server, which redirects the attacker browser back to the client. The malicious client then stores the session cookie at the user's browser, and logs the user in as an attacker [8, 26]. This type of attack exploits the security flaws of the protocol to sign in the user to a trusted client under an account controlled by the attacker. The user may not realize s/he signed in as an attacker, especially if the attacker knows the user's personal information (e.g.: username, profile image), and s/he may reveal some sensitive information at a malicious client controlled by the attacker. To defeat against this type of attack, Barth et al. [1] suggests to use a fresh nonce at the start of the protocol. The client should generate a fresh nonce, store this nonce in the browser's cookie, and assign it to the *return_to* parameter in the authentication request. After the authentication is obtained, the client should validate that the nonce in the browser's cookie is equal to the nonce included in the *return_to* parameter. OIDC specification suggests a similar technique [7].

6.3 A malicious Server

6.3.1 User Spying

OpenID is associated with a large number of privacy issues that occur as a result of using the same id to sign in to multiple clients. Since the protocol does not provide any way to hide the user's identity, a malicious authorization server can easily spy on users and track their activity on the Internet [8,26]. The server can basically log every user's transaction, and then use some data-mining tool to extract users' information; thus, violating their privacy. It still seems hard for OpenID users to avoid disclosing their surfing habits since the protocol does not suggest any way to handle this serious issue. Encrypting the users' transactions or sending them anonymously to the server is one solution that could be adopted to mitigate this issue.

6.3.2 User Recycling

Most of the large IdP companies recycle users' ids that belong to inactive accounts, and give them to new owners [24]. This can cause a large privacy issue since the new owner with a recycled id can get an access to previous owner clients and the data stored there.

Using a unique fragment with users' id is one solution that has been proposed in the OpendID 2.0 specification to handle this problem [11, 26]. The IdP server should generate a new, and unique fragment part within the URL identifier for each user; the clients then use these fragments to distinguish the current owner from the previous owner. Nevertheless, because the specification does not enforce authorization servers to adopt the fragments solution, OpenID recycling is still an issue that might happen with any user [7, 11, 26].

6.3.3 The Attacker will Focus on the Server

The more popular OpenID gets, the more lucrative it becomes for malicious agents to attack the authorization server, since stealing the users' id account gives them the ability to access all users' clients including the data stored at these clients [8, 26]. Therefore, it is imperative to concentrate on the protection of the authorization server and the users credentials stored there.

6.4 Compromised Communication Channel

6.4.1 Diffie-Hellman/ Man-in-the-middle

Diffie-Hellman is used with OpenID 1.0/2.0 to establish an agreement between the client, and the server based on a shared symmetric key [7]. Since DH key exchange is vulnerable to a MitM attack, the attacker in the middle between the server and the client can act as a server to a client and vice versa.

To avoid this attack, OpenID 2.0 specification suggests that any exchange between the client and the server should be executed over a secure SSL channel [11]. OpenID Connect suggests a similar solution [7].

6.4.2 Reply Attack

The redirect phase from the server to the client can be exploits to perform a reply attack. The attacker, who obtains the redirected message by sniffing the wire, can replay it and get signed in to the client as a user [25]. OIDC specification suggests that a redirection phase from the server to the client can be associated with a nonce value to mitigate the reply attack [7].

To summarize the results of this section, table 1 provides an overview of the security issues based on the space of the attacker. The rightmost column indicates if this attack is still an issue in OIDC. A *Yes* indicates that the problem is solved, *No* indicates that the issue still exists, and *Maybe* indicates that some recommendations were made but a mechanism is not enforced.

7 Analysis, Conclusion and Future Work

In this report, we studied the OIDC specification, which describes an improved version of the already popular SSO protocol. We created a formal model and verified it using AVISPA, a state-of-the-art model checker for security protocols. We combined the knowledge gained in the modeling process with previous results obtained by other research studies to classify the security threats based on the attack space of the intruder. We are confident that the OIDC model is an important contribution, being the first study that we are aware of that models OIDC and verifies some of its security goals.

Because of the time limitations of our study, we had to make several important assumptions to simplify our model. We modeled only part of the specification (we omitted the discovery and association), and assumed reliable channels between the authorization server and the client, and between the authorization server and the user. Although that we would like to further analyze this protocol using more pessimistic assumptions, such as an insecure channel between the server and the client, our work sets important foundations for future analysis of this protocol. Moreover, by applying formal methods we can have greater confidence that, under the circumstances assumed in this paper, all known security attacks are successfully prevented. Consequently, we hope to motivate more studies that take advantage of recent improvements in the area of formal verification and apply them to security theory. For example, encoding security problems as symbolic representations that can be solved using state-of-the-art SMT solvers, like it is done in [13] for enforcing control flow integrity.

In addition, we believe that some of the existing vulnerabilities of the previous version of the protocol (as described in Section 3) are general issues that are intrinsic to the Internet and are present in many other protocols. Some others are related to the OIDC protocol itself and it might occur as a result of the way the protocol is designed. These specific issues can be associated to different causes related to the protocol's different parties. Therefore, our classification of the security issues surrounding OpenID into different categories based on attack space criteria is a great contribution to understand-

Category	**Threat**	**Still an issue in OIDC?**
A malicious user	Using malicious URL identifiers	No
	Cross-site forgery request	Maybe
A malicious client	Phising	Yes
	Realm spoofing	No
	Session swapping	Maybe
A malicious server	The server tracks its users	Yes
	OpenID Recycling	Maybe
A compromised channel	MitM	No
	Replay attack	Maybe

Table 1: Classification of OpenID Connect vulnerabilities

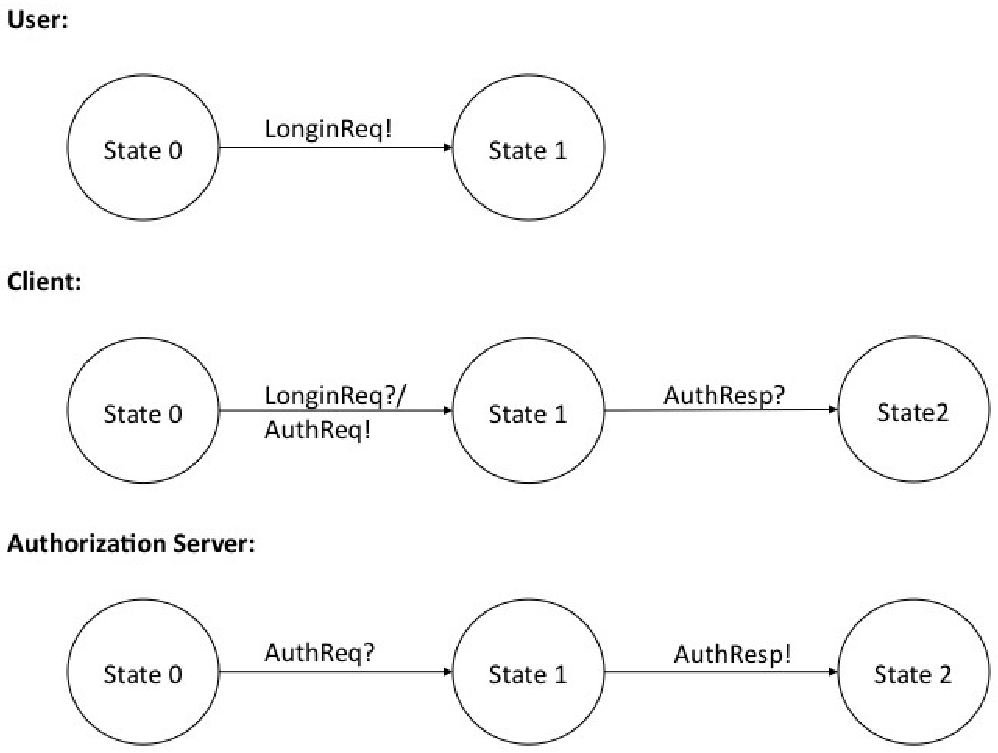
ing the risks in a more systematic way. Furthermore, this classification can serve as a framework for reasoning about threats and mechanism implementations. We can imagine that in the near future security organizations might set standard recommendations that address the different attack spaces as described in this paper. We hope our classification can serve as a starting point for development of such documentation.

For future work, we would like to investigate what are the limits of model checking as applied to security protocol implementations and what kind of properties can be verified. This would require a deep understanding of model checking algorithms and temporal logics. Moreover, a more ambitious question would be to incorporate probabilistic information to security models, such as in priced automata [14], that would allow us to draw conclusions about the feasibility of certain attacks to occur.

Acknowledgements

Our special thanks to Dr. Ligatti and Dr. Zheng at the University of South Florida/CSE for their encouragements and guidance during the entire research process.

Appendix A: Transition Systems for the Authorization Flow



Appendix B: OIDC Model Code in HLPSL

```
%% PROTOCOL: OpenID Connect
%% PURPOSE: Web Single Sign-On
%% REFERENCE: http://openid.net/specs/openid-connect-core-1_0.html
%% MODELERS: Ghada Zamzmi, Hernan Palombo
%% LAST UPDATE: Apr 18th, 2014
%% ROLES: U is user, C is client, A is authorization server
%% ALICE_BOB:
%% U -> C : LoginReq, (U.Uid.Nlr)
%% C -> A : AuthReq, (Cid.{Uid}_Kac)
%% C -> U : LoginRedir, (Cid.Suri)
%% U -> A : UserLogin, (Uid.{Cid.Upwd}_Kau)
%% A -> C : AuthResp, ({Cid.Uid.T}_Kac)
%% A -> U : SuccessRedir, ({Cid}_Kau)
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

role user (
  U, C, A: agent, % user needs to know about other roles
  Kau, Kcu: symmetric_key,
  Uid, Upwd: text, % the user id and password
  SND_CU, RCV_CU, SND_AU, RCV_AU: channel(dy)
) played_by U def=

  local
    State : nat,
    Nlr: text, % Login Request Nonce
    Suri: text, % server URI
    Cid: text % client id

  init State := 0

  transition
  % Send Login Request
  0. State  = 0 /\ RCV_CU(start) =|>
     State':= 1
     /\ Nlr':= new()
     /\ SND_CU(U.Uid.Nlr')
%% added nonce to prevent replay attack
%      /\ witness(U, C, c_u_uid, U.Uid.Nlr')
  1. State = 1 /\ RCV_CU(Cid'.Suri') =|>
     State' := 2
%% vul' 1: i can spoof cid & suri (phishing)
%      /\ request(U, C, u_c_cid_suri, Cid'.Suri')
     /\ SND_AU(Uid.{Cid'.Upwd}_Kau.Suri')
     /\ witness(A, U, s_u_auth, Uid.Upwd.Suri')
%% (even though the uid & suri could be sent by
%% the intruder to the S, the upwd would not match
%% and would not be able to authN the i)
```

```
    /\ RCV_AU({Cid}_Kau)
end role

role client (
  C, A: agent,
  Kcu, Kac: symmetric_key,
  Cid, Suri: text,
  RCV_UC, SND_UC, SND_AC, RCV_AC: channel(dy)
) played_by C def=

  local
    State : nat,
    U: agent,  % U is a browser and the C does not know U
    Uid: text,  % OID URL (e.g., http://user.openid.com)
    Nlr: text, % Login Request Nonce
    Tok: text

 init State := 0

 transition
  % receive LoginRequest, send AuthReq
  0. State  = 0  /\ RCV_UC(U.Uid'.Nlr)
     =|>
     State':= 1
     /\ SND_AC(Cid.{Uid'}_Kac)
     /\ SND_UC(Cid.Suri)
     /\ witness(U, C, u_c_cid_suri, Cid.Suri)
     /\ witness(A, C, s_c_cid, Cid)
%     /\ request(C, U, c_u_uid, U'.Uid'.Nlr')
  1. State=1 /\ RCV_AC({Cid.Uid.Tok'}_Kac) =|>
    State':=2
      /\ secret(Tok', sec_tok, {A,C})
end role

role authorization_server (
  A, U, C: agent,
  Kac, Kau: symmetric_key,
  Uid, Upwd, Cid, Suri: text,
  SND_CA, RCV_CA, SND_UA, RCV_UA: channel(dy)
) played_by A def=

  local
    State: nat,
    Tok: text

  init State := 0

  transition
  % Receive AuthRequest
```

```
   0. State  = 0  /\ RCV_CA(Cid.{Uid}_Kac) =|>
      State':= 1
  % Receive LoginAttempt
   1. State = 1 /\ RCV_UA(Uid'.{Cid'.Upwd'}_Kau.Suri') =|>
      State':= 2
       % authN U on uid & pwd
       /\ request(A, U, s_u_auth, Uid'.Upwd'.Suri')
       % authN C on Cid received from U
       /\ request(A, C, s_c_cid, Cid')
       /\ Tok' := new()
       /\ SND_CA({Cid.Uid.Tok'}_Kac)
       /\ secret(Tok', sec_tok, {A,C})
       /\ SND_UA({Cid}_Kau)
end role

role session(
  U, C, A: agent,
  Kac, Kau, Kcu: symmetric_key, % shared-keys
  Uid, Upwd, Cid, Suri: text
) def=

  local
    SCU, RCU,
    RUC, SUC,
    SAC, RAC,
    SAU, RAU,
    SUA, RUA,
    SCA, RCA: channel (dy)

composition
  user(U, C, A, Kau, Kcu, Uid,
    Upwd, SCU, RCU, SAU, RAU)
  /\ client(C, A, Kac, Kcu, H, Cid,
    Suri, RUC, SUC, SAC, RAC)
  /\ authorization_server (A, U, C,
    Kac, Kau, H, Uid, Upwd, Cid,
    Suri, SCA, RCA, SUA, RUA)
end role

role environment() def=
  const
  u_c_cid_suri, s_u_auth,
  s_c_cid, sec_tok: protocol_id, % goals
  u, c, s: agent,
  uid: text,
  pwd: text,
  cid: text,
  suri: text,
  h: hash_func,
  kac, kau, kcu: symmetric_key

 intruder_knowledge = {u, c,  s, uid, suri}

 composition
   session(u, c, s,
   kac, kau, kcu,
   uid, pwd, cid, suri)
end role

goal
  authentication_on u_c_cid_suri
  authentication_on s_u_auth
  authentication_on s_c_cid
  secrecy_of sec_tok
end goal

environment()
```

References

1. Adam Barth, Collin Jackson, and John C Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88. ACM, 2008.

2. David Basin, Sebastian Mödersheim, and Luca Vigano. *An on-the-fly model-checker for security protocol analysis*. Springer, 2003.

3. Iliano Cervesato. The dolev-yao intruder is the most powerful attacker. In *16th Annual Symposium on Logic in Computer ScienceLICS*, volume 1. Citeseer, 2001.

4. Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

5. Luca Compagna. Sat-based model-checking of security protocols. *Phd, Universita di Genova, Italy, and University of Edinburgh, UK*, 2005.

6. Danny Dolev and Andrew C Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.

7. J. Bradley et. al. OpenID Connect Core 1.0. `http://openid.net/specs/openid-connect-core-1_0.html`, 2014. [Online; accessed 18-April-2014].

8. Nash et. al. OpenID Security Issues. `https://sites.google.com/site/openidreview/issues`, 2009. [Online; accessed 18-April-2014].

9. Dinei Florencio and Cormac Herley. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web*, pages 657–666. ACM, 2007.

10. OpenID Foundation. What is an OpenID? `http://openid.net`. [Online; accessed 18-April-2014].

11. OpenID Foundation. OpenID Authentication 2.0 - Final. `http://openid.net/specs/openid-authentication-2_0.html`, 2007. [Online; accessed 18-April-2014].

12. Ulrike Hugl. Approaching the value of privacy: Review of theoretical privacy concepts and aspects of privacy management. 2010.

13. Paul B Jackson, Bill J Ellis, and Kathleen Sharp. Using smt solvers to verify high-integrity programs. In *Proceedings of the second workshop on Automated formal methods*, pages 60–68. ACM, 2007.

14. Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification*, pages 585–591. Springer, 2011.

15. Alexander Lindholm. *Security evaluation of the OpenID protocol*. Skolan för datavetenskap och kommunikation, Kungliga Tekniska högskolan, 2009.

16. Luca. Authentication vs Weak Authentication. `http://marc.info/?l=avispa-users&m=121372895911115`, 2007. [Online; accessed 18-April-2014].

17. Marino Miculan and Caterina Urban. Formal analysis of facebook connect single sign-on authentication protocol. In *SOFSEM*, volume 11, pages 22–28, 2011.

18. Lawrence O'Gorman. Comparing passwords, tokens, and biometrics for user authentication. *Proceedings of the IEEE*, 91(12):2021–2040, 2003.

19. Hyun-Kyung Oh and Seung-Hun Jin. The security limitations of sso in openid. In *Advanced Communication Technology, 2008. ICACT 2008. 10th International Conference on*, volume 3, pages 1608–1611. IEEE, 2008.

20. Michael Olson. Research study: Consumer perceptions of online registration and social sign-in@Janrain Blog, February 2011.

21. San-Tsai Sun, Yazan Boshmaf, Kirstie Hawkey, and Konstantin Beznosov. A billion keys, but few locks: the crisis of web single sign-on. In *Proceedings of the 2010 workshop on New security paradigms*, pages 61–72. ACM, 2010.

22. San-Tsai Sun, Kirstie Hawkey, and Konstantin Beznosov. Systematically breaking and fixing openid security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures. *Computers & Security*, 31(4):465–483, 2012.

23. The AVISPA Team. HLPLS Tutorial . `http://www.avispa-project.org/package/tutorial.pdf`, 2006. [Online; accessed 18-April-2014].

24. Allen Tom. What's broken in OpenID 2.0? `http://lists.openid.net/pipermail/openid-general/2007-May/002386.html`, 2007. [Online; accessed 18-April-2014].

25. Eugene Tsyrklevich and Vlad Tsyrklevich. Single sign-on for the internet: a security story. *BalckHat USA*, 2007.

26. Bart van Delft and Martijn Oostdijk. A security analysis of openid. In *Policies and Research in Identity Management*, pages 73–84. Springer, 2010.

27. Christian Weiske. OpenID Libraries. `http://wiki.openid.net/w/page/12995176/Libraries`, 2013. [Online; accessed 18-April-2014].