Analysing the Security of Google's implementation of OpenID Connect

Wanpeng Li and Chris J Mitchell

Information Security Group, Royal Holloway, University of London
Wanpeng.Li.2013@live.rhul.ac.uk, C.Mitchell@rhul.ac.uk

Abstract. Many millions of users routinely use Google to log in to relying party (RP) websites supporting Google's OpenID Connect service. OpenID Connect builds an identity layer on top of the OAuth 2.0 protocol, which has itself been widely adopted to support identity management. OpenID Connect allows an RP to obtain authentication assurances regarding an end user. A number of authors have analysed OAuth 2.0 security, but whether OpenID Connect is secure in practice remains an open question. We report on a large-scale practical study of Google's implementation of OpenID Connect, involving forensic examination of 103 RP websites supporting it. Our study reveals widespread serious vulnerabilities of a number of types, many allowing an attacker to log in to an RP website as a victim user. These issues appear to be caused by a combination of Google's design of its OpenID Connect service and RP developers making design decisions sacrificing security for ease of implementation. We give practical recommendations for both RPs and OPs to help improve the security of real world OpenID Connect systems.

1 Introduction

In order to help alleviate the damage caused by identity attacks and simplify management of identities, a range of identity management systems, such as OAuth 2.0, Shibboleth, CardSpace and OpenID, have been put forward [1–3]. As a replacement for the well-established OpenID [3] scheme, OpenID Connect 1.0 [4] builds an identity layer on top of the OAuth 2.0 framework [2]. The OAuth 2.0 framework enables an RP to obtain profile information about the end user, but does not provide any means for the RP to obtain information about the authentication of the end user. In OpenID Connect, in addition to obtaining profile information about the end-user, RPs can obtain assurances about the end user's identity from an OpenID Provider (OP), which itself authenticates the user.

OpenID Connect involves interactions between four core parties:

1. the End User (U), who accesses on-line services of the RP;
2. the User Agent (UA), typically a web browser, that is employed by an end user to transmit requests to, and receive responses from, web servers;
3. the OpenID Provider (OP), e.g. Google, which provides methods to authenticate an end user and generates assertions regarding the authentication event and the attributes of the end user;

4. the Relying Party (RP), e.g. Wikihow, which provides protected on-line services and consumes the identity assertion generated by the OP in order to decide whether or not to grant access to the end user.

In summary, the end user employs a UA to access resources provided by the RP, which relies on the OP to provide authentic information about the user. Even though OpenID Connect was only finalised at the start of 2014, there are already more than half a billion OpenID Connect-based user accounts provided by Google [5], PayPal [6] and Microsoft [7]. This large user base has led very large numbers of RPs to integrate their services with OpenID Connect.

The security of OAuth 2.0, the foundation for OpenID Connect, has been analysed using formal methods [8–10]. Research focusing on implementations of OAuth 2.0 has also been conducted [11–15]. However, as a newly standardised protocol, it is not yet clear how secure practical implementations of OpenID Connect really are. Given the large scale use of Google's service, clarifying this issue is vitally important. To help answer the question, the operation of all one thousand sites from the GTMetrix Top 1000 Sites [16] providing services in English was examined. Of these sites, 103 were found to support the use of the Google's OpenID Connect service at the time of our survey (early 2015). All 103 of these websites were further examined for potential vulnerabilities, with results as reported below. All RPs and the Google OP site were treated as black boxes, and the HTTP traffic sent between RP and OP via the browser was carefully analysed. For every identified vulnerability, we implemented and tested an exploit to evaluate the possible attack surface.

Our study reveals serious vulnerabilities of a number of types, occurring in many of the examined sites; they either allow an attacker to log in to the RP as the victim user or enable compromise of potentially sensitive user information. Google has customised its implementation of OpenID Connect by combining SDKs, web APIs and sample code, and so the OpenID Connect specification only acts as a loose guide to what RPs actually implement. Further examination suggests that the identified vulnerabilities are mainly caused by Google's implementation of its *Hybrid Server-side Flow*, and by RP developers making design decisions sacrificing security for simplicity of implementation. Some of the attacks use cross-site scripting (XSS) [17–20] and cross site request forgeries (CSRFs) [21–26], well-established and widely exploited attack techniques.

OpenID Connect is used to protect millions of user accounts and sensitive user information stored at RPs and the Google OP server. Moreover, as of April 20th 2015, Google shut down its OpenID 2.0 [27] service; as a result a huge number of RPs have had to upgrade their Google sign-in service to use OpenID Connect. It is therefore vitally important that the issues we have identified are addressed urgently, and that Google considers issuing updated advice to all RPs using its service. In this connection we have notified all the RPs in whose OpenID Connect service we have identified the most serious vulnerabilities, as well as Google itself. To summarise, we make the following contributions:

– We report on the first field study of the security properties of Google's implementation of OpenID Connect.

- We examined the security of all 103 of the RPs supporting the Google OpenID Connect service from the GTMetrix list of the Top 1000 Sites.
- We discovered a number of vulnerabilities which allow an attack to log in to the RP as a victim user, we reported our findings to the most serious affected websites and Google, and helped these RPs fix the identified problems.
- We propose practical improvements which can be adopted by OpenID Connect RPs and OPs that address the identified problems.

The paper is organised as follows. In §2 we review OpenID Connect. We describe our adversary model in §3. §4 describes the experiments we performed. Possible reasons for the identified vulnerabilities are discussed in §5. In §6 we propose mitigations for these vulnerabilities, we review related work in §7, and §8 concludes the paper.

2 OpenID Connect

As already noted, OpenID Connect 1.0 [4] builds an identity layer on the OAuth 2.0 protocol. The added functionality enables RPs to verify an end user identity by relying on an authentication process performed by an OpenID Provider (OP).

2.1 OpenID Connect Tokens

In order to enable an RP to verify the identity of an end user, OpenID Connect adds a new type of token to OAuth 2.0, namely the *id_token*. This complements the *access_token* and *code*, which are already part of OAuth 2.0. These three types of token are all issued by an OP, and have the following functions.

- A *code* is an opaque value which is bound to an identifier and a URL of the RP. Its main purpose in OpenID Connect is as a means of giving an RP authorisation to retrieve other tokens from the OP. In order to help minimise threats arising from its possible exposure, it has a limited validity period and is typically set to expire shortly after issue to the RP [2].
- An *access_token* is a credential used to authorise access to protected resources stored at a third party (e.g. the OP). Its value is an opaque string representing an authorization issued to the RP. It encodes the right for the RP to access data held by a specified third party with a specific scope and duration, granted by the end user and enforced by the RP and the OP.
- An *id_token* contains claims about the authentication of an end user by an OP together with any other claims requested by the RP. Claims that can be inserted into such a token include: the identity of the OP that issued it, the user's unique identifier at this OP, the identity of the intended recipient, the time at which it was issued, and its expiry time. It takes the form of a JSON Web Token [28] and is digitally signed by the OP.

Both an *access_token* [29] and an *id_token* [30] can be verified by making a call to the web API of the issuing OP.

2.2 Authentication Flows

OpenID Connect builds on user agent HTTP redirections. We suppose an end user wants to access RP services, which consumes OP-generated tokens. The RP generates an authorization request on behalf of the end user and sends it to the OP via the UA (typically a web browser). The OP provides ways to authenticate the end user, asks the end user to allow the RP to access the user attributes, and generates an authorization response which includes tokens of two types: *access_tokens* and *id_tokens*, where the latter contain claims about user authentication. The RP can use a received *access_token* to access end user's attributes using the OP-provided API, and after receiving an *id_token* the RP learns about the user authentication, as summarised in Fig. 1.

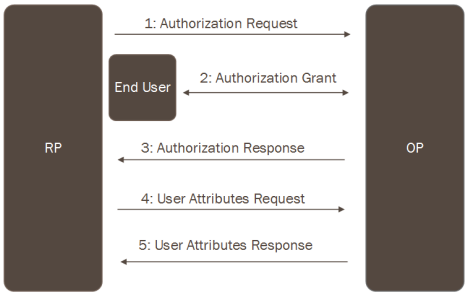

Fig. 1. OpenID Connect Protocol Overview

OpenID Connect [4] supports four authentication flows [5], i.e. ways in which the system can operate, namely *Hybrid Server-side Flow* (or *Hybrid Flow*) [31], *Authorization Code Flow*, *Client-side Flow* (or *Implicit Flow*), and *Pure Server-side Flow*. We describe the first two, since they are most relevant here.

An RP must register with the OP before using Google OpenID Connect. During registration, the OP gathers security-critical information about the RP, including either the RP's redirect URI or its *origin*. The redirect URI is used in the *Authorization Code Flow*, and the user agent is redirected to it after step 5 of §2.2. The *origin* is used in the *Hybrid Server-side Flow* and *Client-side Flow*, and points to the RP's domain name. The OP issues the RP with a unique identifier (*client_id*) and a secret (*client_secret*), used to authenticate the RP when using the *Authorization Code Flow* or *Hybrid Server-side Flow*.

Hybrid Server-side Flow Google's OpenID Connect uses **postMessage** [32–35] to enable cross domain communication between an RP and Google's OP. Normally, scripts on different pages can only access each other if the web pages that caused them to execute are at locations sharing the same protocol, port number and host. The **postMessage** method gives a way to securely pass messages across domains — see, for example, in Son and Shmatikov [35]. In the

Hybrid Server-side Flow (see Fig. 2) and *Client-side Flow*, an RP JavaScript Client (RPJC) runs on the UA and listens for the postMessage event.

We now describe the *Hybrid Server-side Flow*, which is summarised in Figure 2 where the numbers correspond to the numbered steps below.

1. U → UA → RPJC: The user clicks the Google button on the RP website, causing the UA to trigger the RPJC to generate an authorization request.
2. RPJC → UA → OP: The RP generates an OpenID Connect authorization request and sends it to the OP via the UA. This request includes *client_id*, an identifier the RP registered with the OP previously; *response_type=code token id_token*, requesting that a *code*, an *access_token* and an *id_token* be returned directly from Google; *redirect_uri=postmessage*, indicating **postMessage** is being used; *state*, used by the RP JavaScript Client to maintain state between the request and the callback (step 5 below); *origin*, a URL without a path appended; and the *scope* of the requested permission.
3. OP → UA: If the OP has already authenticated the user then this step and the next are skipped. If not, the OP returns a login form to collect authentication information (e.g. user account and password).
4. U → UA → OP: The user completes the login form and grants permission for the RP to access the attributes stored by the OP.
5. OP → UA: After receiving the permission grant, the OP generates an HTML document containing the authorization response and returns it to the UA. The authorization response contains the *code*, *access_token* and *id_token* generated by the OP; and *state* as sent in step 2.
6. UA → RPJC → RP: The UA executes the JavaScript inside the HTML document it received in the previous step. The JavaScript sends the authorization response using **postMessage** to the RPJC which is running on the UA and listening for the **postMessage** event. After the RPJC receives the authorization response it extracts the *code* and sends it back to the RP.
7. RP → OP: The RP produces an *access_token* request and sends it to the OP token endpoint directly (i.e. not via the UA). The request includes *grant_type=authorization_code*, indicating that the RP wants to use the *code* to retrieve an *access_token* from the OP; the *code* generated in step 5; *redirect_uri=postmessage*, indicating that **postMessage** has been used to get the *code*; and *client_secret*, the secret shared by the RP and OP.
8. OP → RP: The OP checks the *code*, *client_secret* and *redirect_uri* and, if correct, responds to the RP with *access_token* and *id_token*, the latter of which is the same as the *id_token* sent in step 5.
9. RP → OP: The RP verifies the *id_token*. If valid, the RP knows the user has been authenticated. If necessary it can make a web API call to retrieve user attributes from the OP, using the *access_token* as authorisation.

Authorization Code Flow One advantage of this flow is that no tokens are available to the UA or any malicious application able to access the UA. If either of the tokens are compromised they could be used to access sensitive user data

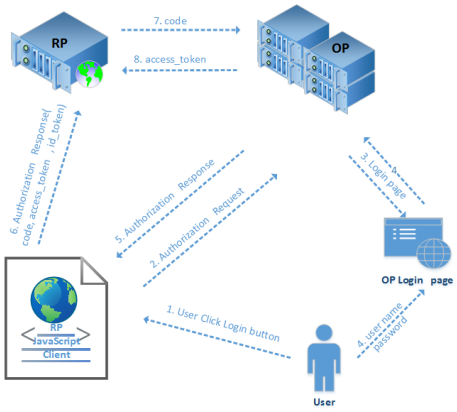

Fig. 2. Google's Hybrid Server-side Flow

and/or successfully masquerade as the user. The OP must authenticate the RP before it issues the tokens, and hence use of the *Authorization Code Flow* requires that an RP shares a secret with the OP. The flow involves the OP returning an authorization *code*, typically a short-lived opaque string, to the RP, which uses it to obtain the *id_token* and *access_token* directly from the OP's *access_token* endpoint, i.e. not via the UA. The main steps are as follows.

1. U → RP: The user clicks a button on the RP website, as displayed by the UA, causing the UA to send an HTTP or HTTPS request to the RP.
2. RP → UA → OP: The RP generates and sends an OpenID Connect authorization request to the OP via the UA, including *client_id*, previously registered by the RP with the OP; *response_type=code*, indicating use of *Authorization Code Flow*; *redirect_uri*, to which the OP will redirect the UA after granting access; *state*, used by the RP to maintain state between request and callback (step 5 below); and the *scope* of the requested permission.
3. OP → UA: If the OP has already authenticated the user then this step and the next are skipped. If not, the OP returns a login form to collect user authentication data.
4. U → UA → OP: The user completes the login form and grants permission for the RP to access the attributes stored by the OP.
5. OP → UA: After using the information provided in the login form to authenticate the user, the OP generates an authorization response and sends it back to the UA. The authorization response contains *code*, the authorization code generated by the OP; and *state*, the value sent in step 2.
6. UA → RP: The UA redirects the response received in Step 5 to the RP.
7. RP → OP: The RP produces an *access_token* request and sends it to the OP token endpoint directly (i.e. not via the UA). The request includes *grant_type=code*, indicating the RP wants to use the *code* to retrieve an *access_token*; the *code* sent in step 5; the *redirect_uri*; and *client_secret*, the secret shared by the RP and OP.

8. OP → RP: The OP checks the *code*, *client_secret* and *redirect_uri* and if all are correct responds to the RP with an *access_token* and *id_token*.
9. RP → OP: The RP verifies the *id_token*. If valid, the RP now knows that the user has been authenticated. If necessary it can also make a call to the OP's web API, using the *access_token* for authorisation, to retrieve user attributes.

3 Adversary Model

In our assessment of the security of Google's OpenID Connect service, and of RPs using the service, we consider two adversary scenarios.

- **A Web Attacker** can share malicious links or post comments containing malicious content (e.g. stylesheets or images) on a benign website; and/or exploit vulnerabilities in an RP website. Malicious content forged by a web attack might trigger the UA to send HTTP(S) requests to an RP and OP using GET or POST methods, or execute attacker JavaScripts. For example, a web attacker could operate an RP website to collect *access_tokens*.
- **A Passive Network Attacker** can intercept unencrypted data sent between an RP and a UA (e.g. by monitoring an open Wi-Fi network).

Conducting a security analysis of commercially deployed OpenID Connect SSO systems requires various challenges to be addressed. These include lack of detailed specifications for the SSO systems, undocumented RP and OP source code, and the complexity of APIs and/or SDK libraries in deployed SSO systems. The methodology we used is similar to that of Wang et al. [14] and Sun and Beznosov [13], i.e. we treated the RPs and OP as black boxes and analysed the BRMs produced during authorization. Since we used a black-box approach, there may be vulnerabilities and implementation flaws we did not uncover.

4 A Security Study

We used Fiddler[1] to capture BRMs sent between RPs and the OP; we also developed a Python program to parse the BRMs to simplify analysis and avoid mistakes arising from manual inspections. All experiments were performed using accounts set up specially for the purpose, i.e. at no time was any user's account accessed without permission. Of the 103 RPs supporting Google OpenID Connect that we examined, 69 (67%) adopt the *Authorization Code Flow*, 33 (32%) use the *Hybrid Server-side Flow*, and just 1 adopted the *Client-side Flow*.

4.1 Studying the security of the Hybrid Server-side Flow

As described in §2.2, Google's OpenID Connect API uses **postMessage** to deliver the authorization response from the OP to an RP. When the RPJC running on the user's browser receives the authorization response from the OP, it extracts the *code* from the authorization response and then submits the *code* back to the RP's OpenID Connect sign-in endpoint.

[1] `http://www.telerik.com/fiddler`

Authentication by Google ID As stated above, the RPJC running on the UA submits the *code* it receives from the Google OP back to the RP's Google sign-in endpoint (see step 6 in §2.2). The RP is meant to use the *code* to retrieve the *access_token* and *id_token* from the OP. However, we observed that 18% of the RPs using the Hybrid Server-side Flow (i.e. 6 of 33) simply submit the user's Google ID to the RP's Google sign-in endpoint; of these, two submit the user's Google ID without appending a *code*, and one submits the user's Google ID with an *access_token*. This led us to suspect that such RPs might be basing their verification of user identity solely on the Google ID, and not using the *code* as intended. If so, then a web attacker knowing a user's Google ID could use it to log in to the user's RP account. We tested this, and found that 9% of the RPs using the Hybrid Server-side Flow (i.e. 3 of 33) have this vulnerability.

Learning a user's Google ID can be relatively simple, as a user's Google+ post URL reveals it. An attacker can use the Google+ *search for people* function to find a victim user to attack, and can then visit the chosen user's Google+ page to learn the ID. For example, `https://plus.google.com/u/0/115722834054889887046/posts` is the Google+ post URL for a Gmail account, for which the ID is `115722834054889887046`.

We reported our findings to the three affected websites, and recommendations were also provided to enable the RP developers to fix the problem (see also 6.3).

Using the Wrong Token An *access_token* is a bearer token, so anyone can use it to get access to the associated user attributes stored by Google. By contrast, the *id_token* is designed for use in providing assurances about user authentication. However, in practice, some RPs use an *access_token* to obtain user authentication assurances without verifying it (i.e. making a web API call to the OP token information endpoint [29]). In such a case, any party with a user's *access_token* can impersonate that user to the RP simply by submitting it. This is a particular threat for a malicious RP, which can routinely obtain *access_tokens* from the Google OP. In other words, any RP using Google OpenID Connect can log in as a victim user to any RPs using an *access_token* to authenticate the user without verifying it. Unfortunately, we found that 58% of RPs using the *Hybrid Server-side Flow* (i.e. 19 of 33) submit an *access_token* back to their Google sign-in endpoint (see step 6 in §2.2) and 45% (i.e. 15 of these 19) use the *access_token* to authenticate the user; of these 15 RPs, only two RPs verify the *access_token* before using it to retrieve user attributes. As a result, 39% of the RPs (i.e. 13 of 33) we examined are vulnerable to this impersonation attack.

We tested the above attack using Burp Suite[2] by submitting an *access_token* obtained from a randomly chosen RP using the *Hybrid Server-side Flow* to the target RP's Google sign-in endpoint. If the attack succeeds, we are able to log in to the target RP as the victim user. As noted above, as many as 39% of the RPs using the *Hybrid Server-side Flow* are vulnerable to this attack. Some of the vulnerable RPs (i.e. 3 out of 13) require additional evidence of the user to be submitted with the *access_token*, in the form of the Google ID or the user's

[2] `http://portswigger.net/burp/`

email address. However, an attacker with an *access_token* can readily use it to get the user's Google ID or email address from Google, and so such additional steps do not prevent the attack.

Intercepting an *access_token* As stated above, 58% of RPs using the *Hybrid Server-side Flow* require the submission of an *access_token* back to their Google sign-in endpoint (see step 6 in §2.2). If the RPJC running on the UA sends an *access_token* back to its Google sign-in endpoint without SSL protection, a passive network attacker is able to intercept it (see §3). According to the OAuth 2.0 specification [36], an *access_token* should never be sent unencrypted between the user browser and the RP. However, we found that 12% of RPs using the *Hybrid Server-side Flow* (i.e. 4 out of 33) send the *access_token* unprotected. A sniffer written in Python was implemented to test this.

We also observed that one additional site, namely TheFreeDictionary[3] does use SSL to protect the transfer of the *code* to its Google sign-in endpoint. However, the *access_token* is subsequently stored in a cookie, and when the cookie is sent from the browser back to TheFreeDictionary the link is not SSL-protected. That is, the *access_token* is observable by a passive eavesdropper.

Privacy Issues When a user chooses to use OpenID Connect to log in to an RP website, the user attributes (e.g. email address, name) that the RP retrieves from the OP should never be revealed to parties other than the RP. SSL connections should be established to protect user information transmitted between the browser and the RP or OP.

However, as explored below, user information leakage might happen if:

- the RPJC running on the user's browser sends user information, the *id_token* or the *access_token* back to its Google sign-in endpoint without SSL protection (see step 6 in §2.2);
- the RP Google sign-in endpoint sends the user information directly to the user's browser without SSL protection; or
- the RP uses SSL to protect the link to the Google sign-in endpoint, but changes to http when sending user information back to the UA.

As described in §4.1, a passive eavesdropper can intercept the *access_token* for 12% of the RPs that use the Hybrid Server-side Flow (i.e. 4 of 33), and can then use it to retrieve potentially sensitive user information, e.g. including Google ID and email address. As stated in §2.1, the *id_token* is a JSON web token in which the user email address and Google ID are encoded in cleartext; so anyone obtaining the token can immediately obtain the information within. One of the four RPs referred to above sends an *id_token* in addition to the *access_token* to its Google sign-in endpoint, and thus a passive web attacker can retrieve the token's user information without requesting it from Google. We also found that one RP did not enable SSL to protect its Google sign-in endpoint, and returned

[3] `http://www.thefreedictionary.com`

user information directly to the UA. Another RP sends user information back to its Google sign-in endpoint without SSL protection. Yet another RP uses SSL to protect the link to the Google sign-in endpoint, but changes to HTTP when sending user information back to the UA. As a result, user privacy cannot be guaranteed for 21% of the RPs we examined (i.e. 7 out of 33). As noted above, a sniffer in Python was implemented to demonstrate the feasibility of the attack.

Session Swapping As discussed earlier, the RPJC running on the UA sends the user's OpenID tokens (i.e. a *code*, an *access_token*, an *id_token*, and/or the user's Google ID) back to its Google sign-in endpoint (see step 6 in §2.2). The OpenID Specification [4] recommends a *state* value should be appended when the RPJC sends the tokens back, and that this *state* value should be bound to the session. If the RPJC fails to send *state*, an attacker can execute a session swapping attack [21, 13, 37] as follows.

1. The attacker logs in to the RP website using his/her own account (step 4 in §2.2), and intercepts the Google-generated tokens (step 5 in §2.2).
2. The attacker constructs a request to the RP's Google sign-in endpoint, including the attacker's own tokens.
3. The attacker inserts the request in an HTML document (e.g. in the **src** attribute of a **img** or **iframe** tag) made available via an HTTP server.
4. The victim user is now, by some means, induced to visit the website offering the attacker's page. The HTML can be constructed in such a way (described in detail below) that the victim's UA will automatically use GET or POST to send the attacker-constructed request to the RP; as a result the user session on the RP website will be bound to the attacker's account.

We observed that 42% of the RPJCs using *Hybrid Server-side Flow* (i.e. 14 of 33) use POST to submit the tokens back to the RP's server without an accompanying *state*. Use of a static **img** or **iframe** tag to perform an attack of the above type does not work against these RPs, as the browser will automatically use GET to retrieve the img and iframe data. In order to use POST to submit those tokens, we created a special HTML page to conduct the attack. We used JavaScript to create an iframe with a unique name in the browser. We then constructed a form inside the iframe whose action points to the RP's Google sign-in endpoint, put the attacker's tokens into the form input, and configured the HTML to submit the form whenever the HTML is loaded by a browser.

To deploy the attack, the constructed HTML page is made available via a web server. If a victim's UA visits the page, the JavaScript inside the HTML automatically submits the attacker's tokens to the RP using POST; as a result the victim user's session at the RP is bound to the attacker's, i.e. a session-swapping attack has been performed. An attacker could use such an attack to collect sensitive user information, e.g. if the victim user updates his credit card information on the RP website, this information will be written to the attacker's account.

Sadly, we found that 73% of RPs using *Hybrid Server-side Flow* (i.e. 24 of 33) are vulnerable. Of these 24 RPs, eight (i.e. 24% of this category) submit a

code to their Google sign-in endpoint; as *code* is a one-time value, the attacker must update it within the attack HTML every time the page is retrieved by a victim. For the other 48% of vulnerable RPs (i.e. 16 of 33), an *access_token* or the user's Google ID is submitted back to the Google sign-in endpoint, in which case the attacker does not need to update the attack page HTML as frequently.

4.2 Studying the security of the Authorization Code Flow

We first observe that Google's OAuth 2.0 *Authorization Code Flow* implementation [38] has similar steps to those in 2.2. The token endpoint provided as part of Google's implementation of OAuth 2.0 (as checked on April 22, 2015) returns an *id_token* to the RP. That is, without knowing details of the RP's internals, we cannot tell whether an RP is using OpenID Connect or OAuth 2.0. We therefore cover all cases where Google returns a *code* to the RP's Google sign-in endpoint under our discussion of the OpenID Connect *Authorization Code Flow*, even though some of the RPs may actually be using OAuth 2.0. However, this makes no difference to our security analysis.

Around 67% of the RPs we examined (i.e. 69 of 103) use *Authorization Code Flow*. Unlike the *Hybrid Server-side Flow*, Google's implementation of *Authorization Code Flow* uses HTTP status code redirect techniques (using code 302) to deliver the authorization response to the RP's Google sign-in endpoint.

Intercepting an *access_token* In the *Authorization Code Flow*, a *code* is returned by Google to the RP's Google sign-in endpoint (see step 6 in §2.2). No tokens are transmitted during the authorization procedure. After the RP receives the *code*, it can use it to retrieve an *access_token* from Google (steps 7/8 in 2.2); it can then use the *access_token* to retrieve user attributes from Google (step 9 in 2.2). The RP then logs the user in to its website.

If an RP does not use SSL to protect communications with its Google sign-in endpoint, a passive web attacker may be able to intercept the *code*. A passive web attacker cannot use the *code* to retrieve an *access_token* from Google, as it will not know the RP's *client_secret* (shared by the RP and Google). However, we observed that, of the RPs using the *Authorization Code Flow*, 6% of their Google sign-in endpoints (i.e. 4 out of 69) return an *access_token* to the user's browser instead of binding the user to the RP's session. As these RPs do not use SSL to protect the transfer of the *access_token*, a passive web attacker is able to obtain the user's *access_token* returned from the RP's Google sign-in endpoint.

Stealing an *access_token* via Cross-site Scripting Google's 'automatic authorization granting' feature [13] generates an authorization response automatically if a user has a session with Google and previously granted permission for the RP concerned. Using this feature, an attacker might be able to steal a user *access_token* by exploiting an XSS vulnerability in the RP or UA.

To test the feasibility of such an attack, an exploit written in JavaScript was implemented. The exploit takes advantage of a recently revealed vulnerability in

Android's built-in browser [39] which allows an attacker to conduct a universal XSS attack [20, 17–19]. The exploit uses a browser **window.open** event to send a forged authorization request to Google's authorization server, within which *response_type=code* (see step 2 in §2.2) is changed to *response_type=code token id_token*. If the user is logged in to his or her Google account and has previously granted permission for this RP, Google automatically generates an authorization response without the involvement of the user; this response is appended as a URI fragment (#) to the redirect URI (see step 5 in §2.2) and is sent back to the RP (see step 6 in §2.2). As the RP Google sign-in endpoint does not expect an URI fragment, a predefined error page will be generated by the RP (e.g. a '404 not found' or 'Failed connection' error). The exploiting JavaScript can now extract the authorization response from the URL of the error page and send it to its opener window, where the **window.open** event is triggered. The opener window then sends the *access_token* to the attacker's server.

Unfortunately, we found that all the RPs using *Authorization Code Flow* are vulnerable to this attack. The vulnerability affects all Android versions up to 4.4, which as of April 6, 2015 still accounted for 53.2% of Android devices[4].

Privacy Issues Unlike the *Hybrid Server-side Flow*, only a *code* is submitted back to the RP's Google sign-in endpoint (see step 6 in §2.2). No user information (e.g. a Google ID or *id_token*) is transmitted during authorisation. However, user information might still leak if the RP Google sign-in endpoint sends the user data directly to the UA without SSL.

We found that 16% of RPs using the *Authorization Code Flow* (i.e. 11 of 69) return user information to the browser directly without SSL protection. Thus a passive web attacker is able to intercept potentially sensitive user information, e.g. if the user is using an open Wi-Fi network (see §3).

Session Swapping If an RP using *Authorization Code Flow* does not enable anti-CSRF measures (e.g. by appending a *state* bound to the browser session to the tokens) to protect its Google sign-in endpoint, a web attacker can launch a session swapping attack, as described in 4.1 for *Hybrid Server-side Flow*.

Unlike the session swapping attack in 4.1, in the *Authorization Code Flow* only the GET method is used to submit the *code* back to the RP's Google sign-in endpoint. This means that the attacker can simply insert the forged request in the **src** attribute of a **img** or **iframe** tag of an HTML document. When the victim user visits the malicious HTML, the browser will automatically send the request to the RP's Google sign-in endpoint using the GET method.

We found that 35% of the RPs using the *Authorization Code Flow* (i.e. 24 of 69) are vulnerable to this attack. However, as *code* is a one time value, the attacker must update it every time the attack page is visited by a victim. Thus such an attack is not as harmful as session swapping in the *Hybrid Server-side*

[4] `https://developer.android.com/about/dashboards/index.html?utm_source=suzunone`

Flow, where an *access_token* which can be used multiple times is submitted back to the RP's Google sign-in endpoint.

Forcing a Login Using a CSRF attack A CSRF login attack operates in the context of an ongoing interaction between a target UA (running on behalf of a target user) and a target RP. A malicious website somehow causes the target UA to initiate an OpenID Connect authorization request to the OP. Because of Google's 'automatic authorization granting' feature, receiving such a request can cause the Google OP to generate an authorization response, which is delivered to the RP without involvement by the user. If the target user is logged in to Google, the UA will send cookies containing the target user's Google OP-generated tokens, along with the attacker-supplied authorization request, to the OP. The OP will process the malicious authorization request as if initiated by the target user, and will generate and send an authorization response to the RP. The target UA could be made to send the spurious request in various ways; for example, a visited malicious site could use the HTML **img** tag's **src** attribute to specify the URL of a malicious request, causing the UA to silently use a GET method to send the request.

We found that 35% of the RPs using *Authorization Code Flow* (i.e. 24 of 69) are vulnerable to such an attack. One consequence is that an attacker can cause a victim user to log in to the RP, as long as the user has previously logged in to Google. This could damage the user experience of the RP website, as the victim user might dislike such a potentially annoying 'automatic login' feature.

5 Security Concerns over Google's implementation of OpenID Connect

In the *Hybrid Server-side Flow*, any authorization request generated by an RPJC using Google's OpenID Connect API will always include *response_type=code token id_token*; as a result, the authorization response returned by Google to the RPJC always contains a *code*, *access_token* and *id_token*. Unfortunately, this feature is the source of many security threats to the system. First, as the *access_token* and *id_token* are directly transferred to the UA, this means that these tokens are potentially revealed to the user agent and any applications which might be able to access the user agent. Second, it gives RP developers a choice — that is, they can choose which token will be submitted back to the RP server by the RPJC. We found that 67% (i.e. 22 out of 33) of RPs using the *Hybrid Server-side Flow* design their RPJC to submit an *access_token* or a user's Google ID back to the RP's Google sign-in endpoint, and this leads to most of the attacks described in §4.1.

5.1 Giving RPs the Ability to Customise the Hybrid-Server-side Flow

According to the OpenID Connect specification [4], a *code* must be returned by the OP to the RP's Google sign-in endpoint (see step 6 in the *Hybrid Server-*

side Flow). However, as described above, in Google's implementation of the Hybrid Server-side Flow, a *code*, *access_token* and *id_token* are always returned by Google to the RPJC running on the user's browser. Unlike the *Authorization Code Flow*, where only a *code* is returned to the RP's Google sign-in endpoint (see step 6 in §2.2) and no RPJC exists, this gives RPs the ability to customise their *Hybrid Server-side Flow*. In fact our experiments have shown that as many as 67% of RPs (i.e. 22 out of 33) customise their implementation of the *Hybrid Server-side Flow* by submitting an *access_token* or a user's Google ID back to the RP's Google sign-in endpoint. Among these RPs, 73% (16 out of 22) are vulnerable to the first two attacks (namely **Authentication by Google ID** which allows an attacker to log in to the RP as any victim user and **Using the Wrong Token** which allows an attacker to impersonate the victim user using an *access_token* generated for another RP) described in §4.1. Moreover, as the *code*, *access_token* and *id_token* are returned by Google inside a HTML document, these values are also revealed to the user agent and hence to any applications (e.g. browser plug-ins), which might be able to access the user agent. If the plug-in or user agent has vulnerabilities which could allow an attacker to access these values, the attacker can steal the user's *access_token*; for example a malicious plug-in which has the right to read the content of HTML pages could obtain the *access_token*.

5.2 No CSRF Countermeasures in the Hybrid-Server-side Flow

In Google's implementation of the *Hybrid Server-side Flow*, the authorization request generated by the RPJC includes a *state* value which is designed to prevent CSRF attacks [21, 23–25]. However, we found that the *state* value extracted by the RPJC is actually a null value; this means that Google itself fails to deliver the *state* value to the RPJC, and hence the *state* value cannot be used to mitigate the threat of a CSRF attack. We also observed that one of the RPs using the *Authorization Code Flow* sends a null *state* value back to its Google sign-in endpoint. As the *state* value generated by the RPJC is not bound to the RP's session and cannot be extracted by the RPJC, another *state* value which is bound to the session needs to be implemented to protect the RP's Google sign-in endpoint against a CSRF attack.

In addition, checking the Google OpenID Connect sample code [40] reveals that Google has not included a *state* value in its example of an RPJC-generated AJAX request, used to send data back to the RP [31] (see step 6 in *Hybrid Server-side Flow*). The lack of a *state* parameter in the sample code and the complexity of implementing anti-CSRF measures helps to explain why 73% of the RPs using the *Hybrid Server-side Flow* are vulnerable to this attack.

5.3 Automatic Authorization Granting

The 'automatic authorization granting' feature of Google's OpenID Connect significantly enhances the user experience and system performance. Without this, users would have to click an "OK" button in a popup window whenever

they wished to log in to an RP, in order to grant authorisation. However, it can also be harmful, since it may allow an attacker to steal an *access_token* (see §4.2) and force a user log in to the RP (see §4.2).

We also found that, in the *Hybrid Server-side Flow*, iframes are used to manage the session [33] between the RPJC and the OP. Suppose a user, who has previously both granted permission for the RP and logged in to his or her Google account, visits the RP login page which contains an iframe pointing to the authorization request. Because of the 'automatic authorization granting' feature, the browser can use the GET method to retrieve the authorization response from Google without involvement by the user. The UA and any applications (e.g. plug-ins) which can access the UA are able to extract the authorization response, which might expose the *Hybrid Server-side Flow* to new attacks.

6 Recommendations

OpenID Connect has been deployed by many RPs and OPs, and increasing numbers of RPs supporting the Google service will likely implement it now Google has shut down its OpenID service. We found serious vulnerabilities in existing deployments of OpenID Connect, and there is a significant danger that these vulnerabilities will be replicated in the future. Below we make a number of recommendations designed to address the identified vulnerabilities. These recommendations primarily apply to RPs using the Google service and to the Google OP itself, but some may have broader applicability. These recommendations are intended both to try to address the problems that exist in current systems, and to help ensure that future systems are built more robustly.

6.1 Recommendations for RPs

When using OpenID Connect, especially the *Hybrid Server-side Flow*, RP developers are responsible for designing the RPJC action on receiving an authorization response from the Google OP. As a result, system security for the RP largely depends on its developers. We have the following recommendations for RPs.

- **Do not customise the Hybrid Server-side Flow:** One reason OpenID Connect is vulnerable to the attacks in §4.1 is that some RPs customise the *Hybrid Server-side Flow*. In particular, instead of submitting a *code* back to its Google sign-in endpoint, the RPJC running in the UA submits an *access_token* or Google ID, which is then used by the RP to authenticate the user. Such a customised *Hybrid Server-side Flow* might improve user experience and RP website efficiency, but this is at the cost of opening serious vulnerabilities. RPs must implement the OpenID Connect *Hybrid Server-side Flow* strictly conforming to the OpenID Connect Specification.
- **Deploy countermeasures against CSRF attacks:** One reason the OpenID Connect systems we investigated are vulnerable to CSRF and session swapping attacks is that the RPs have not implemented any of the

well-known countermeasures. In order to prevent CSRF attacks, Google recommends that RPs include the *state* parameter in the OpenID Connect authorization request and response, and RPs should follow this recommendation.

- **Do not use a constant or predictable *state* value:** Some RPs include a fixed *state* value in the OpenID Connect authorization request. If the *state* value is fixed, it cannot be uniquely bound to the browser session, thereby allowing an attacker to successfully forge a response, since the RP cannot distinguish between a legitimate response produced by a valid user and a forged response produced by an attacker. Hence, in such a case, the inclusion of the *state* value does not protect against CSRF attacks. Thus RPs must generate a non-guessable *state* value which should be bound to the browser session so that the *state* value can used to verify the validity of the response.

6.2 Recommendations for OPs

In an OpenID Connect SSO system, the OP designs the process and provides the API for RPs. An RP supporting a particular OP must therefore comply with the requirements of that OP, and so OPs play a critical role in the system. We have the following recommendations for OPs (and in particular for Google).

- **Remove the *token* from the authorization request in the Hybrid Server Flow:** In the *Hybrid Server-side Flow*, the *token* in the authorization request causes Google to return an *access_token* to the RPJC. This allows RPJCs to submit an *access_token* back to their Google sign-in endpoints, as was the case for 58% of the RPs using the *Hybrid Server-side Flow* that we investigated. This practice gives rise to a range of possible impersonation attacks. Sending the *access_token* also creates further risks, since if the RP does not enable SSL to protect its Google sign-in endpoint, a passive network attacker could steal it. This would not only enable a malicious RP to impersonate a user to those RPs which submit an *access_token* to the Google sign-in endpoint, but also allow the possibility of other misuses of this token, e.g. to compromise sensitive user data.
- **Add a *state* value to the sample code:** OPs typically provide sample code to help RP developers make their website interact appropriately with the OP. As we discovered, Google does not include a *state* value in its sample code for the *Hybrid Server-side Flow*. It seems reasonable to speculate that this is the main reason why 73% of the RP-OP interactions we analysed (see §4.1) are vulnerable to session swapping attacks. However, for cases where a *state* value is included in Google's sample code, this number fell to 35% (see §4.2).
- **Allow the RP to specify the *state* value in the Hybrid Server Flow:** The *state* value in the authorization request of the *Hybrid Server-side Flow* is automatically handled by the Google OpenID Connect API. However, the RPJC cannot extract the *state* as it is null. As the *state* value is not bound to the browser session, it does not protect the RP against CSRF attacks.

It would probably be better to let the RP handle the *state* rather than the Google API. Google should also check the source code of its postmessage.js script to ensure that *state* can be extracted by the RPJC.

6.3 Notifying affected parties

Given their seriousness, we reported the **Authentication by Google ID** issues directly to the affected parties in Feb. 2015 and also gave advice to help fix the problems. As of 16/11/15, one had fixed the problem, one ignored our warning, and the third terminated support for Google SSO. On 17/4/15 we notified Google of all the issues described here. Google acknowledged the problem in §5.2 and notified their OpenID Connect group. However, as of 16/11/15 we are not aware of any other steps taken by Google.

7 Related Work

OAuth 2.0 has been analysed using formal methods. Pai et al. [9] confirmed a security issue described in the OAuth 2.0 Thread Model [8] using the Alloy Framework [41]. Chari et al. analysed OAuth 2.0 in the Universal Composability Security framework [42] and showed that OAuth 2.0 is secure if all the communications links are SSL-protected. Frostig and Slack [10] discovered a cross site request forgery attack in the Implicit Grant flow of OAuth 2.0, using the Murphi framework [43]. Bansal et al. [44] analysed the security of OAuth 2.0 using the WebSpi [45] and ProVerif models [46]. However, all this work is based on abstract models, and so delicate implementation details are ignored.

Meanwhile, the security properties of real-world OAuth 2.0 implementations have also been examined. Wang et al. [14] examined deployed SSO systems, focussing on a logic flaw present in many such systems, including OpenID. In parallel, Sun and Beznosov [13] also studied deployed systems of OAuth 2.0. Li and Mitchell [12] examined the security of deployed OAuth 2.0 systems providing services in Chinese. In parallel, Zhou and Evans [15] conducted a large scale study of the security of Facebook's OAuth 2.0 implementation. Chen et al. [11], and Shehab and Mohsen [47] have looked at the security of OAuth 2.0 implementations on mobile platforms. However, unlike OAuth, very little research has been conducted on OpenID Connect security, except for the recent work of Mladenov et al. [48] who looked at the security of the OpenID Connect Discovery and Dynamic Registration extensions.

8 Concluding Remarks

We have reported on the first field study of the security properties of Google's implementation of OpenID Connect. We examined the security of all 103 of the RPs that implement support for the Google OpenID Connect service from the GTMetrix list of the Top 1000 Sites. Our study reveals widespread serious

vulnerabilities of a number of types, many allowing an attacker to log in to an RP website as a victim user. We give practical recommendations for both RPs and OPs to help improve the security of real world OpenID Connect systems.

References

1. Chappell, D.: Introducing windows cardspace. (2006) http://msdn.microsoft.com/en-us/library/aa480189.aspx.
2. Hardt, D.: The OAuth 2.0 authorization framework. (2012) http://tools.ietf.org/html/rfc6749.
3. Recordon, D., Fitzpatrick, B.: OpenID Authentication 2.0 — Final. (2007) http://openid.net/specs/openid-authentication-2_0.html.
4. Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., Chuck, M.: OpenID Connect Core 1.0. (2014) http://openid.net/specs/openid-connect-core-1_0.html.
5. Google Inc.: Google OpenID Connect 1.0. (2015) https://developers.google.com/accounts/docs/OpenIDConnect.
6. PayPal Holdings, Inc.: PayPal OpenID Connect 1.0. (2014) https://developer.paypal.com/docs/integration/direct/identity/log-in-with-paypal/.
7. Microsoft Inc.: Microsoft OpenID Connect. (2014) https://msdn.microsoft.com/en-us/library/azure/dn645541.aspx.
8. Lodderstedt, T., McGloin, M., Hunt, P.: OAuth 2.0 Threat Model and Security Considerations. (2013) http://tools.ietf.org/html/rfc6749.
9. Pai, S., Sharma, Y., Kumar, S., Pai, R.M., Singh, S.: Formal verification of OAuth 2.0 using Alloy framework. In: Proceedings of the International Conference on Communication Systems and Network Technologies (CSNT), 2011, IEEE (2011) 655–659
10. Slack, Q., Frostig, R.: Murphi Analysis of OAuth 2.0 Implicit Grant Flow. (2011) http://www.stanford.edu/class/cs259/WWW11/.
11. Chen, E.Y., Pei, Y., Chen, S., Tian, Y., Kotcher, R., Tague, P.: Oauth demystified for mobile application developers. In Ahn, G., Yung, M., Li, N., eds.: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014, ACM (2014) 892–903
12. Li, W., Mitchell, C.J.: Security issues in OAuth 2.0 SSO implementations. In Chow, S.S.M., Camenisch, J., Hui, L.C.K., Yiu, S., eds.: Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings. Volume 8783 of Lecture Notes in Computer Science., Springer (2014) 529–541
13. Sun, S.T., Beznosov, K.: The devil is in the (implementation) details: An empirical analysis of OAuth SSO systems. In Yu, T., Danezis, G., Gligor, V.D., eds.: the ACM Conference on Computer and Communications Security, CCS '12, Raleigh, NC, USA, October 16-18, 2012, ACM (2012) 378–390
14. Wang, R., Chen, S., Wang, X.: Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In: IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA, IEEE Computer Society (2012) 365–379
15. Zhou, Y., Evans, D.: SSOScan: Automated testing of web applications for Single Sign-On vulnerabilities. In Fu, K., Jung, J., eds.: Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014., USENIX Association (2014) 495–510

16. GTmetrix: GTmetrix Top 1000 Sites. (2015) http://gtmetrix.com/top1000.html.
17. Nadji, Y., Saxena, P., Song, D.: Document structure integrity: A robust basis for cross-site scripting defense. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009, The Internet Society (2009)
18. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Krügel, C., Vigna, G.: Cross site scripting prevention with dynamic data tainting and static analysis. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007, The Internet Society (2007)
19. Wassermann, G., Su, Z.: Static detection of cross-site scripting vulnerabilities. In Schäfer, W., Dwyer, M.B., Gruhn, V., eds.: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, ACM (2008) 171–180
20. Kirda, E., Krügel, C., Vigna, G., Jovanovic, N.: Noxes: a client-side solution for mitigating cross-site scripting attacks. In Haddad, H., ed.: Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006, ACM (2006) 330–337
21. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. In Ning, P., Syverson, P.F., Jha, S., eds.: Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008, ACM (2008) 75–88
22. De Ryck, P., Desmet, L., Joosen, W., Piessens, F.: Automatic and precise client-side protection against CSRF attacks. In Atluri, V., Díaz, C., eds.: Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings. Volume 6879 of Lecture Notes in Computer Science., Springer (2011) 100–116
23. Jovanovic, N., Kirda, E., Kruegel, C.: Preventing cross site request forgery attacks. In: Second International Conference on Security and Privacy in Communication Networks and the Workshops, SecureComm 2006, Baltimore, MD, Aug. 28 2006 - September 1, 2006, IEEE (2006) 1–10
24. Mao, Z., Li, N., Molloy, I.: Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In Dingledine, R., Golle, P., eds.: Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers. Volume 5628 of Lecture Notes in Computer Science., Springer (2009) 238–255
25. Zeller, W., Felten, E.W.: Cross-site request forgeries: Exploitation and prevention. Bericht, Princeton University (2008)
26. Shernan, E., Carter, H., Tian, D., Traynor, P., Butler, K.R.B.: More guidelines than rules: CSRF vulnerabilities from noncompliant oauth 2.0 implementations. In Almgren, M., Gulisano, V., Maggi, F., eds.: Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings. Volume 9148 of Lecture Notes in Computer Science., Springer (2015) 239–260
27. Google Inc.: Google OpenID 2.0. (2015) https://developers.google.com/accounts/docs/OpenID.
28. Jones, M., Sakimura, N., Bradley, J.: JSON Web Token (JWT). (2014) http://tools.ietf.org/html/draft-ietf-oauth-json-web-token-21.
29. Google Inc.: Google OAuth 2.0 Client-side. (2015) https://developers.google.com/identity/protocols/OAuth2UserAgent?hl=es.

30. Bray, T.: Verify ID Tokens. (2015) https://www.tbray.org/ongoing/When/201x/2013/04/04/ID-Tokens.
31. Google Inc.: Google OpenID Connect Server-side Flow. (2015) https://developers.google.com/+/web/signin/server-side-flow.
32. W3C: HTML5 Web Messaging. (2012) http://www.w3.org/TR/2012/WD-webmessaging-20120313/.
33. de Medeiros, B., Agarwal, N., Sakimura, N., Bradley, J., Jones, M.B.: OpenID Connect Session Management. (2014) http://openid.net/specs/openid-connect-session-1_0.html.
34. Barth, A., Jackson, C., Mitchell, J.C.: Securing frame communication in browsers. Commun. ACM **52** (2009) 83–91
35. Son, S., Shmatikov, V.: The postman always rings twice: Attacking and defending postmessage in HTML5 websites. In: 20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013, The Internet Society (2013)
36. Jones, M., Hardt, D., eds.: The OAuth 2.0 Authorization Framework: Bearer Token Usage. (2012) https://tools.ietf.org/html/rfc6750.
37. van Delft, B., Oostdijk, M.: A security analysis of OpenID. In de Leeuw, E., Fischer-Hübner, S., Fritsch, L., eds.: Policies and Research in Identity Management - Second IFIP WG 11.6 Working Conference, IDMAN 2010, Oslo, Norway, November 18-19, 2010. Proceedings. Volume 343 of IFIP Advances in Information and Communication Technology., Springer (2010) 73–84
38. Google Inc.: OAuth 2.0 Authorization Code Flow. (2015) https://developers.google.com/identity/protocols/OAuth2WebServer.
39. Baloch, R.: Android Browser Same Origin Policy Bypass. (2014) http://www.rafayhackingarticles.net/2014/08/android-browser-same-origin-policy.html.
40. Google Inc.: Google OpenID Connect Hybrid Server-side Flow. (2014) https://developers.google.com/+/web/signin/.
41. Jackson, D.: Alloy 4.1. (2010) http://alloy.mit.edu/community/.
42. Chari, S., Jutla, C.S., Roy, A.: Universally composable security analysis of OAuth v2.0. IACR Cryptology ePrint Archive **2011** (2011) 526
43. Dill, D.L.: The mur*phi* verification system. In Alur, R., Henzinger, T.A., eds.: Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings. Volume 1102 of Lecture Notes in Computer Science., Springer (1996) 390–393
44. Bansal, C., Bhargavan, K., Delignat-Lavaud, A., Maffeis, S.: Discovering concrete attacks on website authorization by formal analysis. Journal of Computer Security **22** (2014) 601–657
45. Bansal, C., Bhargavan, K., Maffeis, S.: WebSpi and web application models. (2011) http://prosecco.gforge.inria.fr/webspi/CSF/.
46. Blanchet, B., Smyth, B.: (ProVerif: Cryptographic protocol verifier in the formal model) http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.
47. Shehab, M., Mohsen, F.: Securing OAuth implementations in smart phones. In Bertino, E., Sandhu, R.S., Park, J., eds.: Fourth ACM Conference on Data and Application Security and Privacy, CODASPY'14, San Antonio, TX, USA - March 03 - 05, 2014, ACM (2014) 167–170
48. Mladenov, V., Mainka, C., Krautwald, J., Feldmann, F., Schwenk, J.: On the security of modern Single Sign-On protocols: OpenID Connect 1.0. CoRR **abs/1508.04324** (2015)