Mitigating CSRF attacks on OAuth 2.0 Systems

Wanpeng Li
School of Computing, Mathematics and Digital Technology
Manchester Metropolitan University
W.Li@mmu.ac.uk

Chris J Mitchell
Information Security Group
Royal Holloway,University of London
C.Mitchell@rhul.ac.uk

Thomas Chen
Department of Electrical and Electronic Engineering
City, University of London
Tom.Chen.1@city.ac.uk

***Abstract*—Many millions of users routinely use Google, Facebook and Microsoft to log in to websites supporting OAuth 2.0 and/or OpenID Connect. The security of OAuth 2.0 and OpenID Connect is therefore of critical importance. Unfortunately, as previous studies have shown, real-world implementations of both schemes are often vulnerable to attack, and in particular to cross-site request forgery (CSRF) attacks. In this paper we propose a new and practical technique which can be used to mitigate CSRF attacks against both OAuth 2.0 and OpenID Connect.**

***Index Terms*—OAuth 2.0, OpenID Connect, CSRF**

I. Introduction

Since OAuth 2.0 appeared in 2012 [13], it has been widely used for single sign-on (SSO). Websites use OAuth 2.0 to simplify user password management and save users from re-entering attributes that are instead held by identity providers (IdPs). There is a correspondingly rich infrastructure of IdPs providing identity services using OAuth 2.0, e.g. as shown by the fact that some Relying Parties (RPs), e.g. USATODAY (https://login.usatoday.com/USAT-GUP/authenticate/?), support as many as six different IdPs.

The theoretical security of OAuth 2.0 has been analysed [1], [2], [4], [6], [11], [21], [26]. Research focusing on practical security and privacy properties of OAuth 2.0 implementations has also been conducted [7], [16], [17], [24], [27], [29], [32], revealing that many real-world implementations of OAuth 2.0 and OpenID Connect have serious vulnerabilities, often because IdP implementation advice is hard to follow.

In this paper we look at a class of cross-site request forgery (CSRF) attacks applying to many real-world implementations of both OAuth 2.0 and OpenID Connect (which is OAuth 2.0-based). We introduce a new class of mitigations which can prevent such attacks; such techniques are needed because, although existing mitigations are effective in principle, for a variety of reasons these are often not deployed in practice.

The remainder of this paper is as follows. §II provides background, and §III describes how RP support multiple IdPs. §IV gives our adversary model and also details possible CSRF attacks against OAuth 2.0 and OpenID Connect. In §V we propose a new way of mitigating CSRF attacks. §VI describes how CSRF attacks can be mitigated using a specific OAuth 2.0 client library. §VII describes possible limitations of our approach and also possible ways of avoiding these limitations. §VIII concludes the paper.

978-1-5386-7493-2/18/$31.00

II. Background

A. OAuth 2.0

OAuth 2.0 [13] allows an application to access resources (typically personal information) protected by a *resource server* on behalf of the *resource owner*, through the consumption of an *access token* issued by an *authorization server*. The OAuth 2.0 architecture involves the following four roles. The *Resource Owner* is typically an end user. The *Client* is an application running on a server, which makes requests on behalf of the resource owner. The *Authorization Server* generates access tokens for the client, after authenticating the resource owner and obtaining its authorization. The *Resource Server* stores protected resources and consumes access tokens provided by an authorization server.

OAuth 2.0 has four protocol flows, i.e. ways for RPs to obtain access tokens; we are only concerned here with the *Authorization Code Grant* and *Implicit Grant* flows. Protocol parameters given in bold below are mandatory [13].

B. OpenID Connect

OpenID Connect 1.0 [22] is an identity layer on top of OAuth 2.0, enabling RPs to verify user identities by relying on authentications performed by an *OpenID Provider (OP)*. To enable RPs to verify user identities, OpenID Connect adds a new token type to OAuth 2.0, the *id_token*. This complements the OAuth 2.0 access token and code. An *id_token* contains claims about the authentication of an end user by an OP, together with any other claims requested by the RP. OpenID Connect supports three authentication flows [22]: *Hybrid Flow*, *Authorization Code Flow* and *Implicit Flow*.

C. OAuth 2.0 used for SSO

When using OAuth 2.0 for SSO, the resource server and authorization server jointly form the IdP, the client is the RP, and the resource owner is the user. OAuth 2.0 and OpenID Connect SSO systems build on user agent (UA) redirections, where a user (U) wishes to access services protected by the RP which consumes the access token generated by the IdP.

1) RP Registration: The RP must register with the IdP, during which the IdP gathers RP information, including the RP's ***redirect_uri***, the URI to which the UA is redirected after the IdP has generated the authorization response. The IdP issues the RP with a ***client_id*** and a *client_secret*, which can be used by the IdP to authenticate the RP.

2) Authorization Code Grant: OAuth 2.0 Authorization Code Grant is very similar to OpenID Connect Authorization Code; we thus only give the description of Authorization Code Grant. This flow relies on certain information having been established during registration (see §II-C1). The protocol proceeds as follows.

1) U → RP: The user clicks a login button on the RP, which causes the UA to send an HTTP request to the RP.
2) RP → UA: The RP sends an authorization request to the UA, including: ***response_type=code***, requesting Authorization Code Grant; *state*, an opaque value used by the RP to maintain state between request and callback (step 6 below); and *scope*, the scope of the requested permission; ***client_id*** and *redirect_uri*.
3) UA → IdP: The UA redirects the request to the IdP.
4) IdP → UA: The IdP first compares the value of *redirect_uri* received in step 3 with the registered value; if the comparison fails, the process terminates. If the user has already been authenticated by the IdP, then the next step is skipped. If not, the IdP returns a login form which is used to collect user authentication data.
5) U → UA → IdP: The user completes the login form and grants permission for the RP.
6) IdP → UA → RP: The IdP generates an authorization response and redirects the UA back to the RP. The authorization response contains ***code***, the code generated by the IdP; and *state*, the value sent in step 2.
7) RP → IdP: The RP sends an access token request to the IdP token endpoint directly (i.e. not via the UA). The request includes ***grant_type=authorization_code***, ***client_id***, *client_secret*, ***code***, and the ***redirect_uri***.
8) IdP → RP: The IdP checks ***client_id***, *client_secret* (if present), ***code*** and ***redirect_uri*** and, if the checks succeed, responds to the RP with *access_token*.
9) RP ↔ IdP: The RP passes *access_token* to the IdP via a defined API to request the user attributes.

3) Implicit Grant: OAuth 2.0 Implicit Grant is very similar to OpenID Connect Implicit and Hybrid, and so we only describe Implicit Grant. We specify below only those steps where Implicit Grant differs from Authorization Code Grant.

2. RP → UA: The RP sends an OAuth 2.0 authorization request back to the UA including: ***response_type=token***, indicating that Implicit Grant is requested; ***client_id***; *redirect_uri*; *state* and *scope*.
6. IdP → UA → RP: The IdP generates an access token and redirects the UA back to the RP using the *redirect_uri*. The access token is appended to *redirect_uri* as a URI fragment (i.e. a suffix to the URI following a # symbol).

As URI fragments are not sent in HTTP requests, the access token is not immediately transferred when the UA is redirected to the RP. Instead, the RP returns a web page (typically HTML with an embedded script) which accesses the full redirection URI including the UA-retained fragment and extracts the access token from the fragment. The RP can now use the token to retrieve data from the IdP.

III. Supporting multiple IdPs

As described in §I, RPs can support multiple IdPs, since users will have varying IdP trust relationships — e.g., one user may prefer Facebook, whereas another may prefer Google. We next describe two ways in which this is achieved in practice.

A. Using redirect URIs

An RP can support multiple IdPs by registering a different *redirect_uri* with each IdP, and setting up a sign-in endpoint for each. It can then use the endpoint on which it receives an authorization response to learn which IdP sent it. E.g., AddThis (http://www.addthis.com/) has registered:

- https://www.addthis.com/darkseid/account/register-facebook-return as its Facebook *redirect_uri* and
- https://www.addthis.com/darkseid/account/register-google-return as its Google *redirect_uri*.

If AddThis receives an authorization response at the endpoint *https://www.addthis.com/darkseid/account/register-facebook-return?code=[code_generate_by_Facebook]*, (in step 7 of §II-C2), it assumes that this response was generated by Facebook, and thus sends the authorization *code* to the Facebook server (step 8 of §II-C2) to request an *access_token*.

B. Explicit User Intention Tracking

Alternatively, an RP keeps a record of the IdP each user wishes to use (e.g. it could save the identity of the user's selected IdP to a cookie). In this case, when a authorization response is received by the RP, the RP can retrieve the identity of the IdP from the cookie and then send the *code* to this IdP. This method is typically used by RPs that allow for dynamic registration, where using the same URI is an obvious implementation choice [11].

IV. CSRF Attacks against OAuth 2.0 Systems

A. Adversary Model

We suppose the adversary is a **web attacker**, i.e. it can share malicious links or post comments which contain malicious content (e.g. stylesheets or images) on a benign website, and/or can exploit vulnerabilities in an RP website. The malicious content might trigger the web browser to send an HTTP/HTTPS request to an RP and IdP using the GET or POST methods, or execute attacker-crafted JavaScript.

B. CSRF attacks

A CSRF attack [3], [5], [9], [15], [19], [23], [31] operates in the context of an ongoing interaction between a target UA (used by a target user) and a target website. A malicious website causes the target UA to initiate a request of the attacker's choice to the target website. This can cause the target site to execute actions without user involvement. E.g., if the target user is currently logged into the target site, the target UA will send cookies containing an authentication token generated by the target site for the target user, along with the attacker-supplied request, to the target site. The target site will then process the malicious request as though it was initiated by the target user. According to a 2017 OWASP report [20],

CSRF vulnerabilities are present in 5% of web applications, meaning that such attacks are a real practical danger.

C. CSRF Attacks Against the Redirect URI

CSRF attacks against the OAuth 2.0 *redirect_uri* [18] can allow an attacker to get authorization to access OAuth-protected resources without user consent. Such attacks are possible for both Authorization Code Grant and Implicit Grant. An attacker first acquires a *code* or *access_token* for its own resources. The attacker then aborts the redirect flow back to the RP on the attacker's own device, and then, by some means, tricks the victim into executing the redirect back to the RP. The RP receives the redirect, fetches the attributes from the IdP, and associates the victim's RP session with the attacker's resources accessible using the tokens. The victim user then accesses resources on behalf of the attacker. The impact of such an attack depends on the type of resource accessed. For example, the user might upload private data to the RP, thinking it is uploading information to its own profile, and this data will subsequently be available to the attacker. Alternatively, as described by Li and Mitchell [16], an attacker can use a CSRF attack to control a victim user's RP account without knowing the user's username and password.

D. Existing CSRF Defences

Barth et al. [3] describe four website mitigations for CSRF attacks. One involves the use of a secret validation token, sent in each HTTP request; this token can be used to determine whether the request came from an authorized source. The "validation token" should be hard to guess for an attacker who does not already have access to the user's account. If a request does not contain a validation token, or the token does not match the expected value, the server should reject the request. The other three techniques all involve distinguishing between same-site and cross-site requests and so, as we discuss below, they do not help in the context of OAuth 2.0.

According to the OAuth threat model [18], two possible mitigations for CSRF attacks on OAuth 2.0 RPs are:

- a *state* parameter (i.e. a secret validation token, as above) should be used to link the authorization request to the redirect URI used to deliver the *code* or *access_token*;
- RP developers and end users should be educated not to follow untrusted URLs.

The other three CSRF defences described by Barth et al. [3] are not recommended. This is because the OAuth 2.0 response to the redirect URI is a cross-site request, since the request is generated by the IdP and is redirected to the RP by the browser; thus these three CSRF defences will not work in this case.

Both recommended mitigations need to be implemented by RPs; however, in practice, RPs do not always correctly implement CSRF countermeasures. A 2015 study by Shernan et al. [25] found that 25% of websites in the Alexa Top 10,000 domains using Facebook's OAuth 2.0 service appear vulnerable to CSRF attacks. Further, a 2016 study by Yang et al. [30] revealed that 61% of 405 websites using OAuth 2.0 (chosen from the 500 top-ranked US and Chinese websites) did not implement CSRF countermeasures; even worse, of those RPs which support the *state* parameter, 55% remain vulnerable to CSRF attacks due to misuse of the *state* parameter. They also found four scenarios where the *state* parameter can be misused by RP developers. Thus, if CSRF attacks are to be prevented in practice, new, simple-to-implement countermeasures would be extremely valuable, motivating the work described below.

V. A NEW APPROACH

Since RP developers often fail to add a state parameter to authorization requests, large numbers of real-world OAuth 2.0 implementations are vulnerable to CSRF attacks; moreover traditional Referer header, Origin header and Custom header countermeasures [3] are infeasible in the OAuth 2.0 framework. We propose instead to combine the Referer header and the fact that RPs register different URIs for different IdPs (see §III-A) to provide a novel means of mitigating CSRF attacks. We first describe how a Referer header can be used to mitigate CSRF attacks against the *redirect_uri* in both the Authorization Code Grant Flow of OAuth 2.0 and the (very similar) Authorization Code Flow of OpenID Connect.

A. Protecting the Authorization Code (Grant) Flow

An authorization response is typically only generated after a user clicks on a IdP-rendered grant button. The HTTP message (see, e.g., Listing 1) of such an authorization response contains a Referer header which points to the IdP domain.

```
// privacy related values have been suppressed using ***
GET /AIdP-callback?code=[code_generated_by_AIdP]
Host: RP.com
User-Agent: ***
Accept: ***
Accept-Language: en-US,en;q=0.5
Referer: https://AIdP.com/
Cookie: ***
Connection: close
```

Listing 1. HTTP message of a normal OAuth 2.0 Authorization Response

In practice, major IdPs, such as Google, Facebook and Microsoft, implement an 'automatic authorization granting' feature [17]. That is, when the user has logged in to his/her OAuth 2.0 IdP account, the IdP generates an authorization response without explicit user consent. The HTTP message (see, for example, Listing 2) of such an authorization response contains a Referer header which points to the RP domain.

In the proposed mitigation, when the RP receives an authorization response it first retrieves the identity of the IdP from *redirect_uri*, and then checks that the domain in the Referer header is either the RP Domain or the IdP domain. If so, then the RP knows it is a genuine authorization response coming from the IdP; otherwise, the RP should discard this HTTP message and send an error page to the user.

```
GET /AIdP-callback?code=[code_generated_by_AIdP]
Host: RP.com
User-Agent: ***
Accept: ***
Accept-Language: en-US,en;q=0.5
Referer: https://RP.com/
Cookie: ***
Connection: close
```

Listing 2. HTTP message of an automatic authorization granting OAuth 2.0 Authorization Response

For example, suppose a web attacker puts the link https://rp.com/AIdP-callback?code=[code_BelongsToAttacker_generated_by_AIdP] on attacker.com to try to attack the *redirect_uri* that the RP registered with the target IdP (which we call AIdP). The HTTP message of the attack request will contain a Referer header which points to attacker.com (see Listing 3). The RP can detect this is an attack by examining the domain in the Referer header.

```
GET /AIdP-callback?code=[code_BelongsToAttacker_generated_by_AIdP]
Host: RP.com
User-Agent: ***
Accept: ***
Accept-Language: en-US,en;q=0.5
Referer: https://attacker.com/
Cookie: ***
Connection: close
```

Listing 3. HTTP message of a CSRF attack against *redirect_uri*

B. Protecting the Implicit Grant Flow

We next describe how a Referer header can be used to mitigate CSRF attacks against the *redirect_uri* in both the Implicit Grant Flow of OAuth 2.0 and the Implicit Flow and Hybrid Flow of OpenID Connect.

For example, suppose a web attacker creates the link https://rp.com/AIdP-callback#access_token=[accsstoken_BelongsToAttacker_generated_by_AIdP] on attacker.com to launch a CSRF attack against the *redirect_uri* that RP registered with the target IdP, AIdP. As in §II-C3, the *access_token* in the Implicit Grant Flow is not immediately transferred when the UA is redirected to the RP. Thus the HTTP request message looks similar to the CSRF HTTP request in Listing 4. The only difference between a normal HTTP message and a CSRF HTTP message in the OAuth 2.0 Implicit Grant is the Referer header. The RP can detect a CSRF attack by checking the domain of the Referer header is either the IdP identity it retrieves from the *redirect_uri* or its own domain; it can then respond with different HTML depending on the HTTP messages it has received (see lines 18 and 41 in Listing 4).

```
// A normal HTTP request message to the RP redirect_uri in OAuth 2.0
     Implicit Grant
GET /AIdP-callback
Host: RP.com
User-Agent: ***
Accept: ***
Accept-Language: en-US,en;q=0.5
Referer: https://AIdP.com
Cookie: ***
// The HTTP response message
HTTP/1.1 200 OK
Date: ***
Server: ***
Last-Modified: ***
Content-Length: ***
Content-Type: text/html
Connection: Closed

<html>
<body>
<h1>This HTML can be used to extract the access_token!</h1>
......
</body>
</html>
// The HTTP request message of a CSRF attack on the RP redirect_uri
     in OAuth 2.0 Implicit Grant
GET /AIdP-callback
Host: RP.com
User-Agent: ***
Accept: ***
Accept-Language: en-US,en;q=0.5
Referer: https://attacker.com
Cookie: ***
// The HTTP response message
HTTP/1.1 200 OK
Date: ***
Server: ***
Last-Modified: ***
Content-Length: ***
Content-Type: text/html
Connection: Closed

<html> <body> <h1>A CSRF attack is detected on the AIdP signin
endpoint!</h1> </body> </html>
```

Listing 4. Preventing CSRF attacks on OAuth 2.0 Implicit Grant

C. Supporting multiple IdPs

We have seen how Referer can be used to mitigate CSRF attacks against RPs using *redirect_uri* to track user login intentions. We now describe how the Referer header can be used to protect RPs using explicit user intention tracking. The user log-in intention is stored by the RP as part of the session state (Jsession=12345 in the example below). When the RP receives an authorization response (e.g. that given in Listing 5), it retrieves the IdP's identity from the session state and checks whether the domain of the Referer header is either the IdP identity or its own domain. If so then the RP knows that this is a genuine authorization response; otherwise, it should respond to the user with an error page.

```
GET /oauth2-callback?code=[code_BelongsToAttacker_generated_by_AIdP]
Host: RP.com
User-Agent: ***
Accept: ***
Accept-Language: en-US,en;q=0.5
Referer: https://AIdP.com/
Cookie: Jsession=12345
Connection: close
```

Listing 5. HTTP message of a CSRF attack against *redirect_uri*

VI. RPs using specific IdP client libraries

Many IdPs, such as Facebook[1] and Google[2], implement their own OAuth 2.0 client libraries. RPs can use these libraries to simplify integration of the Facebook and Google OAuth 2.0 services with their websites. These libraries use postMessage [8] to deliver OAuth 2.0 responses to the RP client. The RP client must then use XMLHttpRequest to send the OAuth 2.0 response back to the RP OAuth 2.0 callback endpoint, e.g. https://www.rp.com/AIdP-callback.

The RP OAuth 2.0 callback endpoint might be different from the *redirect_uri* the RP registered with the IdP, e.g. https://www.rp.com; for example, Google requires RPs to register an *origin* value if they want to use the Google OAuth 2.0 client libraries. Because the request to the RP's OAuth 2.0 callback endpoint is initiated from the RP client using XMLHttpRequest, the Referer header in the HTTP message of the request always points to the RP domain.

An RP using these client libraries can detect a CSRF attack by checking that the domain in the Referer header of the HTTP message is as expected (i.e. RP.com in Listing 6).

```
// HTTP message of the request to the RP's OAuth 2.0 callback
     endpoint
GET /AIdP-callback?code=[code_generated_by_AIdP]
Host: RP.com
```

[1]https://developers.facebook.com/docs/facebook-login/web

[2]https://developers.google.com/identity/sign-in/web/devconsole-project

```
User-Agent: ***
Accept: ***
Accept-Language: en-US,en;q=0.5
Referer: https://RP.com
Cookie: ***
Connection: close
```

Listing 6. Defending RPs using specific IdP client libraries

VII. Limitations of our approach

A possible limitation of our Referer header approach is that a UA might suppress the Referer header in a (non-secure) HTTP request if the referring page is transferred via a secure protocol (e.g. HTTPS) [12]. This means that an RP which uses HTTP to register its *redirect_uri* with an IdP cannot use the approach described in §V to defend against CSRF attacks against its *redirect_uri*, since as part of suppression the Referer header will be removed by the web browser when it redirects the authorization response to the RP (we assume here IdPs use HTTPS at their OAuth 2.0 authorization endpoint).

In real-world cases this limitation often does not arise, since many widely used IdPs, including Amazon and Microsoft, require the RP to register its *redirect_uri* using the HTTPS protocol. This means that the attack mitigation described above will work successfully for RPs supporting Amazon and Microsoft login. It would clearly be beneficial if other IdPs could change their registration process to require RPs to register their *redirect_uri* using HTTPS, enabling all RPs to use our approach to mitigate CSRF attacks.

There is also a possible danger that, even if HTTPS is used for *redirect_uri* registration, a UA will, for some other reason, suppress the Referer header. This problem can be avoided using the Referrer Policy [28]; that is, the IdP can set its Referrer Policy to require UAs to not suppress the Referer header in HTTP requests that originate from HTTPS domains, preventing the UA from omitting this header by default.

VIII. Conclusions

We have proposed a new class of mitigations for CSRF attacks against *redirect_uri* in OAuth 2.0 and OpenID Connect. Our approach is practical and simple to implement, and requires no changes to the IdP service; i.e. it can be adopted by an RP independently of what any other party does. RPs can adopt this approach to provide an additional layer of protection against CSRF attacks for their OAuth 2.0 and/or OpenID Connect services. Of course, adoption would likely be increased if this measure was recommended by major IdPs and/or included in relevant specifications.

References

[1] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. *Journal of Computer Security*, 22(4):601–657, 2014.

[2] C. Bansal, K. Bhargavan, and S. Maffeis. WebSpi and web application models. 2011. http://prosecco.gforge.inria.fr/webspi/CSF/.

[3] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proc. ACM CCS 2008*, pages 75–88. ACM, 2008.

[4] B. Blanchet and B. Smyth. ProVerif: Cryptographic protocol verifier in the formal model. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.

[5] J. Burns. Cross site reference forgery: An introduction to a common web application weakness. *Security Partners*, 2005. http://dl.packetstormsecurity.net/papers/web/XSRF_Paper.pdf.

[6] S. Chari, C. S. Jutla, and A. Roy. Universally composable security analysis of OAuth v2.0. *IACR Cryptology ePrint Archive*, 2011:526, 2011.

[7] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague. OAuth demystified for mobile application developers. In *Proc. ACM SIGSAC 2014*, pages 892–903. ACM, 2014.

[8] B. de Medeiros, N. Agarwal, N. Sakimura, J. Bradley, and M. B. Jones. OpenID Connect Session Management. 2014. http://openid.net/specs/openid-connect-session-1_0.html.

[9] P. De Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against CSRF attacks. In *Proc. ESORICS 2011*, volume 6879 of *LNCS*, pages 100–116. Springer, 2011.

[10] D. L Dill. The mur*phi* verification system. In *Proc. CAV '96*, volume 1102 of *LNCS*, pages 390–393. Springer, 1996.

[11] D. Fett, R. Küsters, and G. Schmitz. A comprehensive formal security analysis of OAuth 2.0. In *Proc. ACM SIGSAC 2016*, pages 1204–1215. ACM, 2016.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext transfer protocol–HTTP/1.1, 1999. https://tools.ietf.org/html/rfc2616.

[13] D. Hardt (editor). RFC 6749: The OAuth 2.0 authorization framework. October 2012. http://tools.ietf.org/html/rfc6749.

[14] D. Jackson. Alloy 4.1. 2010. http://alloy.mit.edu/community/.

[15] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *Proc. SecureComm 2006*, pages 1–10. IEEE, 2006.

[16] W. Li and C. J. Mitchell. Security issues in OAuth 2.0 SSO implementations. In *Proc. ISC 2014*, volume 8783 of *LNCS*, pages 529–541. Springer, 2014.

[17] Wanpeng Li and Chris J. Mitchell. Analysing the security of Google's implementation of OpenID Connect. In *Proc. DIMVA 2016*, volume 9721 of *LNCS*, pages 357–376. Springer, 2016.

[18] T. Lodderstedt, M. McGloin, and P. Hunt. RFC 6819: OAuth 2.0 threat model and security considerations. 2013. http://tools.ietf.org/html/rfc6819.

[19] Z. Mao, N. Li, and I. Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *Proc. FC 2009*, volume 5628 of *LNCS*, pages 238–255. Springer, 2009.

[20] OWASP Foundation. Owasp top ten project. 2017. https://www.owasp.org/index.php/Top_10-2017_Top_10.

[21] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh. Formal verification of OAuth 2.0 using Alloy framework. In *Proc. (CSNT) 2011*, pages 655–659. IEEE, 2011.

[22] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and M. Chuck. Openid connect core 1.0. 2014. http://openid.net/specs/openid-connect-core-1_0.html.

[23] H. Shahriar and M. Zulkernine. Client-side detection of cross-site request forgery attacks. In *Proc. ISSRE 2010*, pages 358–367. IEEE Computer Society, 2010.

[24] M. Shehab and F. Mohsen. Securing OAuth implementations in smart phones. In *Proc. CODASPY'14*, pages 167–170. ACM, 2014.

[25] E. Shernan, H. Carter, D. Tian, P. Traynor, and K. R. B. Butler. More guidelines than rules: CSRF vulnerabilities from noncompliant OAuth 2.0 implementations. *Proc. DIMVA 2015*, volume 9148 of *LNCS*, pages 239–260. Springer, 2015.

[26] Q. Slack and R. Frostig. Murphi analysis of OAuth 2.0 implicit grant flow. 2011. http://www.stanford.edu/class/cs259/WWW11/.

[27] S.-T. Sun and K. Beznosov. The devil is in the (implementation) details: An empirical analysis of OAuth SSO systems. In *Proc. ACM CCS '12*, pages 378–390. ACM, 2012.

[28] W3C. Referrer Policy. 2017. https://www.w3.org/TR/referrer-policy/.

[29] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *Proc. SP 2012*, pages 365–379. IEEE Computer Society, 2012.

[30] R. Yang, G. Li, W. C. Lau, K. Zhang, and P. Hu. Model-based security testing: An empirical study on oauth 2.0 implementations. In *Proc. AsiaCCS 2016*, pages 651–662. ACM, 2016.

[31] W. Zeller and E. W. Felten. Cross-site request forgeries: Exploitation and prevention. *Bericht, Princeton University*, 2008.

[32] Y. Zhou and D. Evans. SSOScan: Automated testing of web applications for Single Sign-On vulnerabilities. In *Proc. USENIX Security Symposium 2014*, pages 495–510. USENIX Association, 2014.