A security analysis of the OAuth protocol

Feng Yang and Sathiamoorthy Manoharan
Department of Computer Science
University of Auckland
New Zealand

***Abstract*—The OAuth 2.0 authorization protocol standardises delegated authorization on the Web. Popular social networks such as Facebook, Google and Twitter implement their APIs based on the OAuth protocol to enhance user experience of social sign-on and social sharing. The intermediary authorization code can be potentially leaked during the transmission, which then may lead to its abuse. This paper uses an attacker model to study the security vulnerabilities of the OAuth 2.0 protocol. The experimental results show that common attacks such as replay attacks, impersonation attacks and forced-login CSRF attacks are capable of compromising the resources protected by the OAuth 2.0 protocol. The paper presents a systematic analysis of the potential root causes of the disclosed vulnerabilities.**

***Keywords*-Single sign-on, OAuth, security vulnerabilities.**

I. Introduction

OAuth (open standard for authorization) protocol provides a generic framework to let a resource owner authorize third-party to access the owner's resource held at a server without revelaing to the third-party the owner's credentials (such as username and password) [1], [2].

The protocol works roughly as follows. When required by a (trusted) third-party, the resource owner requests the server to provide a valet key to the third-party. The server authenticates the resource owner (e.g. using username and password) and once authenticated, passes a valet key. The third-party then uses the valet key to access the owner's resources held at the server. See Figure 1.

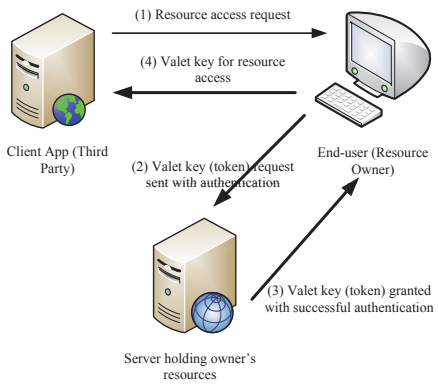

Fig. 1. A rough overview of the OAuth protocol

A leaked valet key can potentially be used by other parties to obtain illegal access to owner's resources. This paper investigates some of the mechanisms that may lead to such compromises. An implementation of OAuth protocol is used as the basis for understanding the protocol. Assuming that it is possible to eavesdrop on the network traffic carrying OAuth data, the paper examines through experiments the ease or otherwise of impersonation and replays. Some of the attacks are seen to be possible and the paper discusses how these attacks could be mitigated.

The rest of this paper is organized as follows. Section II introduces the OAuth protocol. Section III reviews related work. Section IV presents an experimental analysis of the OAuth protocol through attacker models. The final section concludes with a summary and a discussion on the mitigation of attacks.

II. Introduction to OAuth

Quite a number of websites provide APIs to access user information stored on the site. For example, Google provides APIs to add entries to a user's calendar. It will not be appropriate for client applications using such an API to request the user's Google credentials in order to add calendar entries. The OAuth protocol alleviates this issue by allowing the user to obtain from Google an access code that she can pass on to the client application; this access code is then honoured by Google when the application attempts to add to the user's calendar entry. This is known as *delegated authorization*. The user in this case grants access to the client application to add entries to her calendar on her behalf.

OAuth 2.0 specification defines how to handle delegated authorization in a variety of situations such as client-server web applications, mobile application, and desktop applications. It defines four client profiles: Web Server profile, User-Agent profile, Native Application profile, and Autonomous. The web server profile is widely implemented and deployed by several major service providers such as Google, Twitter, Facebook, and Yahoo!.

OAuth 2.0 defines several authorization flows to cope with the specific need of each of the client profiles. They are authorization code flows, implicit grant flows, resource owner password credentials flows and client credentials flow. Server-side web applications typically use authorization code grant flows while client-side web applications typically use implicit grant flows.

978-1-4799-1501-9/13/$31.00

A. Authorization Code Flow and Implicit Grant Flow

In the authorization code flow, the resource owner is redirected back to the client application with an authorization code. To exchange the authorization code for an access token, the client application must provide the client credentials in the access token request. In the implicit grant flow, on the other hand, when the resource owner grants access to the application, instead of returning an authorization code, an access token is immediately redirected back to the client application by the authorization server. Since the client application is hosted in the resource owner machine, likely running in the web browser, returning an authorization code is redundant.

B. Authorization Request

OAuth 2.0 recommends that the authorization endpoint should be protected by the use of TLS while there is no requirement and no recommendation for the protection of the authorization endpoint in WRAP. The vulnerability of not authenticating clients with TLS when using implicit grant is well documented in the OAuth 2.0 specification. Should an implementation choose not to use TLS, the authorization endpoint will be vulnerable to phishing attacks.

C. Authorization Response and Callback Endpoint

In the authorization flow, if the resource owner grants the access request, the authorization server generates an authorization code and adds it in the redirection URI as a query parameter. The redirection passes via the resource owners user-agent to the client. If TLS is not uses, the authorization code can be eavesdropped [3].

D. Access Token Request

In OAuth 2.0, authorization server must implement a TLS mechanism to protect requests to the token endpoints. Figure 2 shows a client's HTTP access token request to Google authorization server. This includes client credential and an authorization code in the POST data.

grant_type:	authorization_code
redirect_uri:	http://127.0.0.1:3000/oauth2callback/
client_id:	678942420820.apps.googleusercontent.com
client_secret:	hWh0fWA79J42z_m19PB7iINU
type:	web_server
code:	4/Y5PZXm . . . GMuFzsEfQI

Fig. 2. Post Data from Access Token Request

E. Access Tokens

According to OAuth 2.0 specification, the access token represents the grant's scope, expiration and other attributes issued by the authorization server. It is important for the legitimate client to keep the access token safe from unintended parties.

OAuth working group published two specifications OAuth 2.0 Message Authentication Code (MAC) Token [4] and "OAuth 2.0: Bearer Token Usage" [5] to describe the usage of the access token. These two drafts define two kinds of access tokens: Bearer Token and MAC Access Token respectively. Bearer tokens are considered less secure compared to MAC access tokens because anyone simply possessing of the token can have access to the protected resources. No signature of the API request is needed in the API endpoint. Although it has this weakness, most major service providers implementing OAuth 2.0 protocol use bearer tokens as a form of access tokens due to its simplicity. There is always a tradeoff between security and usability. To protect against token disclosure, especially bearer tokens, the specification requires that the TLS mechanism must be used in the transmission of access token requests and protected resource requests [5].

To minimize the risk as a consequence of stolen token, the protocol introduces a short-lived access token and a long-live refresh token. If the duration authorized to an access token is expired, the client application can use the refresh token to issue another access token request. What happens if the refresh token is stolen? The answer is any party possessing of the refresh token will have a long duration of authorized access to the protected resources. Therefore, the optional refresh token is only granted to the client applications with high security standard.

III. Related Work

Francisco and Karen classified OpenID and OAuth as a double-redirection protocol [6], [7]. First redirection is the application redirects the browser to a third party authentication or authorization endpoint. Second redirection is the third party endpoint redirects the browser to a callback endpoint of the application. They analyze the security of the double-redirection protocol and find that the user is vulnerable to phishing attacks if the third partys authentication or authorization endpoint is not protected by TLS. Moreover, they point out that implementing TLS in the callback endpoint of the application can provide security protection against man-in-the-middle and passive attacks.

Uruena et al. present a paper studying the privacy risks for the users of the OpenID Single Sign-On mechanism [8]. They find out that OpenID Authentication Protocol relying URL-parameter encoding as a means of information transmission between the Relying Party and the OpenID Provider can potentially compromise privacy of the application users. Even though the parameters of the authentication grant response are signed to protect the integrity, anyone that has access to the full URLs of the authentication request or response can see all these non-encrypted parameters such as OpenID Identifier. It is extremely easy to access these URLs due to the nature of the Hypertext Transfer Protocol (HTTP). Nowadays, it is unlikely for a website to only host their resource . So resources such images and links from third parties are often embedded in the page the user is currently visiting. All these external resources will trigger individual HTTP request to a third party with a Referer header set to the full URL of the web page where those resources are accessed from. Presenting the Referer field in HTTP request header can prevent CSRF attacks and

facilitate traffic analysis. Unfortunately, attackers can take advantage of the Referer field to reveal user confidential information passing in the delegated-based authorization and authentication protocol. Uruena and Busquiel proposed three possible solution to this vulnerability: Redesigning the OpenID Authentication Protocol, Disabling the HTTP Referer field and prohibiting external resources embedded in the web pages processing OpenID Authentication URLs.

Pai et al. formalize the OAuth protocol using a method called knowledge flow analysis [9] . They formalize the OAuth protocol as a set of formulas in the form of predicate logic. These formulas are then used to build a model in Alloy [10]. They show that the OAuth protocol has a security vulnerability in the client credentials flow of the OAuth protocol. Client applications using client credentials flow store client password in the software package. However, reverse engineering can reveal the stored password. Then this compromised secret client credentials could be used in malicious application to harvest sensitive information.

Miculan and Urban present a formal analysis of Facebook Connect protocol using the specification language HLPSL and AVISPA [11]. To translate the Facebook protocol into HLPSL, a detailed specification of the protocol is essential. Unfortunately, Facebook never releases an official specification about its propriety protocol. In order to formalize the Facebook protocol, they analysed all HTTP traffic between the browser, the Facebook authentication server and a client application during the process of the authentication. As a security protocols verification tool, AVISPA revealed two security vulnerabilities which are replay attack and impersonation attack. They provided a countermeasure to defend these attacks.

Bansal et al. modelled several profiles of the OAuth 2.0 protocol in applied pi-calculus and verified them using ProVerif [12], [13]. They introduced a library called WebSpi for modelling web applications and web-based attackers. Typically a complete web application involves three parties: User, User-agent, Server. To simulate these three components in the real world, WebSpi contains three processes: (1) a server-site (PHP-like) process providing services as a website, running on top of HttpServer, (2) a client-side (JavaScript-like) process acting as a user agent with all the basic functionalities of HttpClient, and (3) a user process interacting with the user agent and the latter communicate with the web application. They also introduced a customizable attacker model to launch different kinds of attacks. The Attacker issues a command by signalling the public channel admin. A specific attacker capability can be enabled from three categories by this API . The three categories are Managing principals, Network attackers and Website attackers. Based on the WebSpi model, they identify four circumstances where OAuth 2.0 deployments are vulnerable to Social CSRF attacks such as Automatic Login CSRF, Social Sharing CSRF, Social Login CSRF on stateless clients and Social Login CSRF through AS Login CSRF. Furthermore, they revealed that OAuth 2.0 are vulnerable to token redirection attacks and resource theft as a consequence of access token redirection.

Chari et al. present an analysis on the authorization code mode of OAuth 2.0 using the Universal Composability Security Framework [14], [15]. They implement the server-side authorization flows based on the OAuth 2.0 protocol Internet draft proposal [1]. On top of the implementation, they enhance the security feature by adding a mandatory requirement that the communication channel between the client application and the authentication server must be encrypted by SSL-like functionality. This does not require the usage of session identifier in advance, which can eliminate the possibility of session fixation and session swapping attacks. The refinement is proven to be the complementary of the countermeasures on the security threats introduced by exposing authorization code in the communication channel.

The OAuth threat model provides implementers with a comprehensive threat model analysis and respective counter measures in a systematic way [3].

IV. Experimental Analysis

To evaluate the security of the implementation of the OAuth 2.0 protocol from different parties, including client applications and their associated authorization servers, we implement two client applications. One redirects the login request to a authorization server implemented locally. The other communicates with Google's authorization server. Figure 3 illustrates the architecture for evaluating the OAuth protocol.

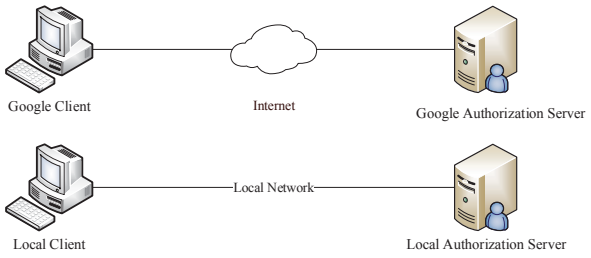

Fig. 3. The Architecture for evaluating OAuth

A user needs to perform three clicks and provide two text inputs in the authorization flow in order to login into the client application.

- First click: they need to click on the external link which redirects them to the authorization endpoint.
- Type username in the authentication page.
- Type password in the authentication page.
- Second click: Click on the submit button to send the authentication request.
- Third click: They need to make a decision if they would like to grant access permissions to the client application.

A. Local Authorization Server

To simplify the design, the authorization server implemented locally serves protected resource requests as well as authorization requests. So the authorization endpoint and

token endpoint are defined in the authorization server. When the user visits the client site and initiates the protocol, the client redirects the user to the authorization endpoint to grant access. The authorization server will ensure the user is in an authenticated state. If not, the user must perform the login action. After the user authenticates, a security validation should be checked by the authorization server to ensure the request is redirected by a legitimate client. This is done by checking the received client id and redirection URI matches the registered URI in the authorization server's persistent data store. If the validation is successful, an authorization code is generated and stored in a temperate storage. Then the authorization code is redirected to the requesting client via the user's web browser. In the meantime, there is a process running in the authorization server to handle access token requests. Like authorization code grant, a validation check needs to perform to ensure it is a legitimate access token request. As discussed previously, an access token request must contain the client credential, redirection URI and a valid authorization code. In addition to making sure the received client credentials and redirection URI match the registered ones, the process still need to validate the submitted authorization code is originally assigned to the requesting client. Failing to do so may result in phishing attack vulnerability.

B. Client Applications

In this implementation, Client applications basically have three functionalities: user redirection, access token requests, and protected resource requests. As discussed previously, when the client application redirects the user to the authorization server, the request must contain the client id and redirection URI. And the scope parameter is optional and is used to limit the access of authorization grant. If the scope parameter is not provided, minimum or default permission is given to the access grant. When the client application exchanges an authorization code with an access token, it must provide its client credentials and redirection URI.

C. The Attacker Model

To evaluate the security of the OAuth 2.0 implementation, we build an attacker model, which automatically simulate common network attacks on the OAuth 2.0 protocol. The attacker model consists of intercepting modules. Each module represents a single attack. Four attack modules are implemented in this work. They are monitoring attack module, replay attack module, phishing attack module, and impersonation attack module. The monitoring attack module is for analysis purposes only and it simply monitors the HTTP traffic.

The attacker model is built on the following attack assumptions:

- It is assumed that the attacker has full access to the network among the three involving parties: the resource owner's user-agent, the client application and the authorization server. Three communication channels (between the resource owner's user-agent and the client application, between the client application and the authorization server, and between the resource owner's user-agent and the authorization server) are eavesdropped by the attacker model.
- It is assumed that the attacker model is equipped with unlimited resources to launch an attack

1) Replay Attack Module: In authorization code flows of the OAuth 2.0 protocol, an authorization code is a representation of the resource owner's access grant to the client. Once it is consumed by the client application, it should be void. The authorization code should be single use. If not, it may result in replay attack vulnerability. The malicious attacker may capture an authorization code redirection request in the communication between the resource owner's user-agent and the client application. Then the attacker resends the request to the client application in order to login in with the account associated with the authorization code.

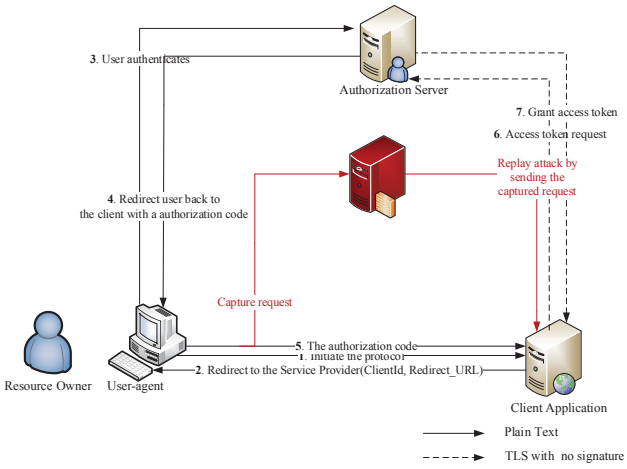

Fig. 4. The Attack Model: Replay Attack

2) Phishing Attack Module: Phishing is a common way in which the attacker masquerades as a trustworthy entity such as a website to gain confidential information such as usernames, passwords, access code, and other security information. As we know, it is extremely easy to obtain a client id from a service provider. For example, as long as you have an account with Google, you can register a client application to consume Google web services in any time. So this means not all the client applications allowed to use web services from a well-known service provider respect user privacy. Some of them may just masquerade as a legitimate website in order to gain authorization codes. In order to launch a phishing attack, the attack can poison DNS cache records on the user's machine. As a result, whenever the victim tries to visit some legitimate sites, he or she is redirected to a malicious client site that he or she is not intended to visit. In this attack module, the resource owner is redirected to a malicious client application rather than the legitimate client

site he or she intended to. When the victim initiates the flow, he is redirected to the authorization server with the request containing the malicious client's client id and redirect URI. Once the malicious application receives the authorization code, it constructs a request with the authorization code and sends it to the callback endpoint of the legitimate client.

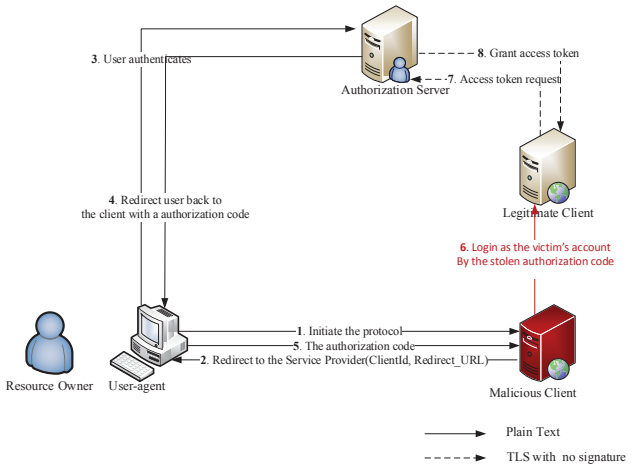

Fig. 5. The Attack Model: Phishing Attack

3) Impersonation Attack Module: This attack module takes advantage of the security vulnerability that the callback endpoint of the client application is not required to use TLS mechanism to protect the confidentiality of the communication. Firstly, the authorization code is eavesdropped by the attacker. Second, the attacker block the original request to keep the stolen authorization code in a fresh state as we know the authorization code could become invalid after it has been used. Finally, the attacker establish a session with the client application by initiating the authorization flow and send a forged request with the stolen authorization code.

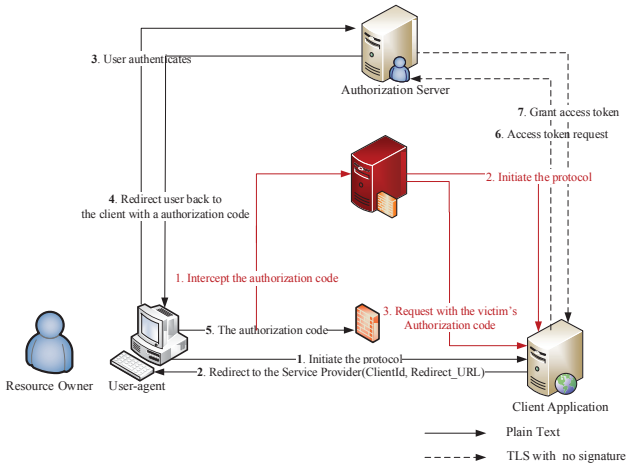

Fig. 6. The Attack Model: Impersonation Attack

D. Observations

We find that the local authorization server is vulnerable to replay attacks. We observed that authorization code redirection is sent by the web browser to the client application, followed by a forged request with the same authorization code. We can see that both requests gain access to the restricted area of the site. This means the local authorization server does not have any authorization code restriction limiting the duration of the authorization code. In contrast, Google authorization server passed the replay attack vulnerability test, which indicates that it enforce one-time use of the authorization code. We also manually tested this vulnerability on Facebook and found out that it allows multiple-time uses. While multiple-time uses of an authorization code gives developers flexibility in implementation, it may result in replay attacks vulnerability. We recommend that the authorization server should enforce one-time use of authorization codes unless proper security measure is put in place to prevent malicious reuse of the authorization code, as specified in the OAuth 2.0 specification.

The Google client application hosted in www.livejournal.com and found that two tests were failed on this website. We observed that the livejournal site failed the impersonation attack vulnerability test. After anaylzing the HTTP traffic from the generated HAR files, we found out that the livejournal site set a cookie in the user's web browser when the user visits a page from the site. When the user is redirected back from the authorization server, the callback endpoint considers the redirection invalid if the cookie is not contained in the request. From the packet captures, it seemed that the site tries to prevent the impersonation attacks by storing a cookie in the user's web browser.

In this case, only having the authorization code is not sufficient enough to impersonate the victim's account. It must cooperate with a valid cookie from livejournal.com. Since the callback endpoint of the livejournal website is not protected by TLS, the HTTP traffic data is not encrypted. So it is extremely easy for the attacker to obtain the cookie as well as the authorization code. Furthermore, the failed test testImpersonationAttackByAuthCodeAndForgedCookie shows that the original cookie is not necessary for the callback endpoint request from the redirection. It can be simply replaced by any other cookie that is obtained by any other connection to the page. This means the livejournal site user's account is likely to be compromised as long as the authorization code is leaked to the attacker. The approach where the livejournal site sets a cookie in advance when the user first visits the page does not thwart the attacks.

How does the Google authorization server perform in the test? The passing GoogleOAuthVulnerabilityTest indicates that appropriate countermeasures are carried out by Goolge to ensure access token requests are legitimate. For example, one-time use of authorization code and the client's authenticity are enforced by Google. To enhance user experience, Google introduces automatic authorization grant features in the authorization flow of OAuth 2.0. This means, if the user has

previously granted permissions to a client application not long ago but currently not in a authenticated session state with the client application, Google will not ask for the user to grants permissions to the client if it receives the authorization request from the client again. Instead, it will redirect the user back to the client application with a new generated authorization code. As both redirections happen instantly, the user does not realize the existence of the redirection. This reduces the number of mouse clicks users need to perform in order to login into the client application. However, it may result in forced login Cross-Site Request Forgery (CSRF) if the client application is vulnerable to CSRF attacks. For example, it does not check the Referer header field or generate random tokens for its pages. An attacker could take advantage of this feature to force the victim login into the client application without the knowledge of the victim. Then further attacks can be launched to compromise the user' data.

However, the impart of the attack could be minimized if the authorization server whitelists the redirection URI instead of matching the domain . As we know, after the authorization request is completed, the authorization server will redirect the user to the registered redirection URI. If only some specific secure URIs are permitted, it would be more difficult for an attacker to launch sequential attacks.

Furthermore, the forced login CSRF attacks is not likely to happen in the situation where the signature of the authorization request is required and the validation of the signature is enforced by the authorization server. This signature requirement was initially there in the protocol but was removed for the reason of simplicity. As a result, it is very difficult for the authorization server to check the validity of the authorization request due to lack of authenticity of the client application in the request.

The pursues of simplicity often sacrifies the security of the system. There is a trade-off between simplicity and security. We recommend that automatic authorization granting feature should only be turned on in a situation where the data security is not so important. Alternately, it should be only enabled for the client applications that meet the security requirement.

V. Summary and Conclusion

The initial goal of OAuth 1.0 protocol is to enables users to grant third-party applications access to their protected resources without revealing their login credentials to the third party. OAuth 2.0 removed several security features for simplicity, which makes the implementation simple and easy. Unfortunately, simplicity often comes with less security. The removal of signature requirements, for example, is a serious security concern. These relaxations in security makes the deployment of network attacks easier.

We presented a systematic root cause analysis of security threats in different phases of the OAuth protocol. The attacker model found out that there are several common network attacks that could be possibly carried out by attackers to impersonate users and access their protected resources, such as replay attacks, network eavesdropping, forced-login CSRF attacks, and impersonation attacks. We further investigated the root causes of these vulnerabilities and found out they are possible because (1) No requirement or no recommendation of TLS protection on callback endpoints; (2) Allowing multiple uses of authorization codes; (3) The removal of the signature requirement of authorization requests disables the authorization server to validate the authenticity of the client application; (4) No vetting process is enforced to ensure the security of the client application before enabling the automatic authorization granting feature; (5) Flexible redirection URIs validation mechanism is not adopted; and (6) No authenticity of the authorization code.

In this work, our analysis on the security threats of OAuth protocol was focused on the communication (1) between the user-agent and the authorization server, and (2) between the user-agent and the client application. Future work remains to analyse the security of the access token transmision between the client application and the authorization server.

References

[1] D. Hardt, "The OAuth 2.0 authorization framework," *The Internet Eng. Task Force RFC 6749*, October 2012.

[2] E. Hammer-Lahav, "The OAuth 1.0 protocol," *The Internet Eng. Task Force RFC 5849*, April 2010.

[3] T. Lodderstedt, M. McGloin, and P. Hunt, "OAuth 2.0 threat model and security considerations," *The Internet Eng. Task Force RFC 6819*, April 2013.

[4] J. Richer, W. Mills, and H. Tschofenig, "OAuth 2.0 message authentication code (MAC) tokens," November 2012. [Online]. Available: http://tools.ietf.org/html/draft-ietf-oauth-v2-http-mac-00

[5] M. Jones and D. Hardt, "OAuth 2.0: Bearer token usage," August 2012. [Online]. Available: http://tools.ietf.org/html/draft-ietf-oauth-v2-bearer-23

[6] K. P. L. Francisco Corella, "Security analysis of double redirection protocols," Pomcor, Tech. Rep., February 2011.

[7] OpenID.net, "OpenID authentication 2.0," *OpenID Foundation*, December 2007.

[8] M. Urueña, A. Muñoz, and D. Larrabeiti, "Analysis of privacy vulnerabilities in single sign-on mechanisms for multimedia websites," *Multimedia Tools and Applications*, pp. 1–18, 2012.

[9] S. Pai, Y. Sharma, S. Kumar, R. Pai, and S. Singh, "Formal verification of oauth 2.0 using alloy framework," in *Intl. Conf. om Communication Systems and Network Technologies*, 2011, pp. 655–659.

[10] D. Jackson, *Software Abstractions*. MIT Press, November 2011.

[11] C. U. Marino Miculan, "Formal analysis of facebook connect single sign-on authentication protocol," *In SofSem 2011, Proceedings of Student Research Forum*, 2011.

[12] C. Bansal, K. Bhargavan, and S. Maffeis, "Discovering concrete attacks on website authorization by formal analysis," in *Proceedings of the IEEE 25th Computer Security Foundations Symposium*, 2012, pp. 247–262.

[13] B. S. Bruno Blanchet, "ProVerif: Automatic cryptographic protocol verier, user manual and tutorial," http://www.proverif.ens.fr/manual.pdf.

[14] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," *Foundations of Computer Science*, pp. 136–14, 2001.

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