The Achilles' Heel of OAuth: A Multi-Platform Study of OAuth-based Authentication

Hui Wang*, Yuanyuan Zhang, Juanru Li, Dawu Gu
Lab of Cryptology and Computer Security
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai, China

ABSTRACT

Websites and mobile applications today increasingly utilize OAuth for authorization and authentication. Major companies such as Facebook, Google and Twitter all provide OAuth services. The usage of OAuth for authorization is well documented and has been studied by many researchers. However, little work has been done to specify or analyze the usage of OAuth for authentication. Given that many developers have employed OAuth for authentication on multiple platforms, we believe it is imperative to conduct a study to understand how developers customize OAuth for authentication on different platforms.

In this paper, we analyze how popular applications on the Web, Android and iOS platform authenticate users with OAuth. Our approach is to dissect the traffic from an attacker's perspective to recover the authentication mechanisms employed by the apps and identify exploitable vulnerabilities. The results show that OAuth-based authentication mechanisms employed by these applications lack sufficient verification and suffer from many vulnerabilities. Closer examination reveals that developers have different tendencies to authenticate users with OAuth on different platforms, and 32.9%, 47.1% and 41.6% of the analyzed mechanisms on the three platforms are vulnerable. We then categorize the root causes of these vulnerabilities and make practical recommendations for developers to help design and implement robust authentication mechanisms with OAuth.

Keywords

OAuth; Single-Sign-On; Authentication

1. INTRODUCTION

OAuth [13] is an open standard for authorization, and it is quickly becoming the de facto standard for handling authentication between apps and websites. Today, many prestigious companies (e.g., Google, Facebook and Twitter) are using OAuth to manage authentication with their APIs. They provide OAuth SDKs for different platforms, to help developers integrate their services. When OAuth is used as an Single-Sign-On (SSO) scheme, it allows a user to identify herself to a third-party application (e.g., ESPN), which we call a relying party (RP), using an identity that is managed by an identity provider (IdP), without exposing her passwords (and other credentials). However, the problem is that OAuth was initially designed for delegating authorization in the web, there exists no specification providing guidance for developers to authenticate users with OAuth on different platforms. Even though major IdPs provide services based on OAuth to help user authentication, these schemes are not authentication protocols themselves. Given the fact that failing to authenticate users with OAuth securely can lead to serious consequences (e.g., user impersonation), and OAuth has been deployed commercially for authentication by many websites/applications, it is very imperative to conduct a study to understand how RPs authenticate users with OAuth on different platforms.

Single-Sign-On (SSO) schemes have been studied by several others [5, 8, 21]. Prior research investigated web SSO protocols such as Open ID, SAML SSO, Facebook Connect, etc. Unlike OAuth, these protocols were specifically designed for authentication rather than authorization. The vulnerabilities uncovered in these protocols were different from the ones in OAuth authentication. Meanwhile, how widely these protocols are deployed is unclear, some of them are even outdated. For example, Facebook Connect has already been deprecated and replaced by OAuth 2.0, Google ID is no longer supported in 2016. For those studies related to OAuth, Sun et al. concentrated on classical web attacks involving stealing the user's access token from an RP [18]. Wang et al. introduced a systematic process for identifying critical assumptions in SDKs, which led to the identification of some vulnerabilities [22]. The focus of these researches was on the process of token exchange, they concerned the transmission of the tokens, the binding between the token and the RP, or the binding between the token and the user, but not the process of identity information exchange. Similar to password authentication, which utilizes the username/password to authenticate a user, the returned identity information is the key to identify a user in OAuth authentication. However, it is unclear how real-world RPs customize the OAuth protocol to authenticate users.

OAuth authentication mechanisms are built on top of the existing authorization protocol, but indeed they are not protocols, they are APIs with proprietary implementations and public interfaces. In a protocol, the actions different parties supposed to take are clearly specified, as well as the roles of

ACSAC '16, December 05-09, 2016, Los Angeles, CA, USA

DOI: http://dx.doi.org/10.1145/2991079.2991105

different parameters in each step. While in OAuth authentication, which parameters are suitable to be used for authentication and how the authenticators should be transmitted are unspecified, let alone how RP servers are supposed to verify the binding between the authenticators and the current user. It is completely up to the RP how to use the outputs of OAuth for authentication. Through a successful exploit of an uncovered flaw in these mechanisms, an attacker can impersonate a victim on an RP application, or entice a victim to login to an RP as the attacker in order to steal the victim's privacy (e.g., trick the victim to linking his credit card to the attacker's account).

To ensure the security of OAuth authentication, we propose an attacker-guided analysis methodology to further the understanding of (1) how RPs decide to customize OAuth for authentication on different platforms, (2) the security and privacy issues of these customized OAuth-based authentication mechanisms, and their prevalence on different platforms, (3) the root causes and consequences of RPs' decisions. First, we utilize an "in-browser"technique to get the view of the entire OAuth-based authentication transaction of an RP from a malicious RP attacker's perspective, including the HTTPS traffic. We pinpoint the critical parameters selected by the RP for authentication in this step, as well as the communication channel used for authenticator transmission. What's more, we perform a differential fuzzing test to check whether the RP server verifies the binding between the authenticator and the current user. Second, we set up a man-in-the-middle proxy to inspect the authentication traffic from an active network attacker's perspective. Based upon the knowledge obtained in the first step, the mechanisms transmitting authenticators in an insecure channel will be observed. Security of an OAuth-based authentication mechanism is evaluated in three aspects: a) whether the authenticators can be used to verified their binding relationship with the current user; b) whether the authenticators are transmitted under well-protection; c) whether the RP server verifies the binding between the authenticators and the current user.

Our study was conducted on 241 popular OAuth-capable applications, including 79/85/77 apps on the Web, Android and iOS platform. The focus was to dissect the OAuth-based authentication mechanisms deployed by RPs on different platforms, and assess the security of these mechanisms. As far as we know, this is the first time an investigation on OAuth-based authentication mechanism is conducted on different platforms. Our study reveals that real-world OAuth-based authentication mechanisms are extremely diverse, and RPs tend to employ different mechanisms on different platforms. The diversity of the real-world OAuth-based authentication mechanisms reflects the real issues with OAuth authentication: (1) The usage of OAuth for authentication is poorly defined and unspecified, (2) RPs customize OAuth for user authentication without sufficient verification. Our study shows that 32.9%, 47.1% and 41.6% of OAuth-based authentication mechanisms implemented in the tested apps on Web, Android, iOS platforms suffer from vulnerabilities we uncovered.

The contributions of this paper can be summarized as follows:

- We show that OAuth-based authentication mechanisms of many websites and applications lack sufficient verification, and are vulnerable to many attacks including user impersonation, client impersonation, enticement attack and token hijacking.
- We present the first study of OAuth-based authentication mechanisms on multi-platforms, including Web, Android, iOS. Our results show that RPs have different tendencies to authenticate users with OAuth on different platforms, and vulnerabilities exist prevalently on these platforms.
- We categorize the root causes of our discovered vulnerabilities, and develop some practical recommendations for RPs to help them authenticate users with OAuth in a secure manner.

2. OAUTH AUTHENTICATION DEMYSTIFIED

In this section, we describe a typical example to illustrate what can go wrong in OAuth authentication. Furthermore, we pinpoint the key portions in OAuth authentication that are security critical, most vulnerabilities in OAuth-based authentication mechanisms relate to theses portions.

2.1 Illustrative Example

When OAuth is used for authentication, the RP needs to request an access token from the IdP when a user elects to use SSO to login, and then exchange the token for user's identity information to identify him. The OAuth authentication logic is illustrated in Figure 1. The key part of OAuth authentication is in Stage 2, i.e., the process of identity information exchange. However, there exists no specification providing the details of the authenticator exchange process. In other words, it is unclear which authenticators are suitable for authentication, how the authenticators should be transmitted, how an RP server is supposed to verify the authenticity of the authenticators. Without stating these clearly, RPs may customize OAuth for authentication in an insecure way.

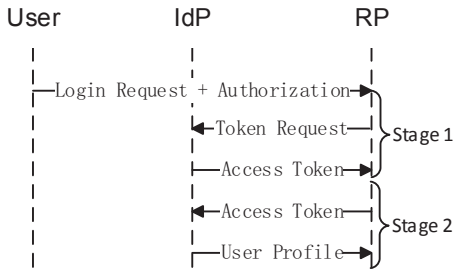


Figure 1: OAuth Authentication Logic

A typical misuse is shown in Figure 2. In this case, the RP utilizes his client to exchange the information between the RP server and the IdP server. And he uses the user profile (e.g., user id) and access token as the authenticator. There are some exploitable vulnerabilities in this mechanism that allow attackers to masquerade as the user on the RP: (1) It's obviously insecure if the authenticators are transmitted in an unencrypted channel, as has been discussed by many others [20, 22]. (2) If the authenticators are protected with HTTPS, a malicious RP attacker (Section 3.2) can intercept his login traffic to the benign RP in his own browser, and

substitute the authenticators with a victim's to hijack the victim's account on the RP. Because a malicious RP can gather the IdP-managed user id and token of a user on his own website, as long as he supports the IdP's SSO service. (3) An attacker can replay the authentication traffic to impersonate the victim on the RP.

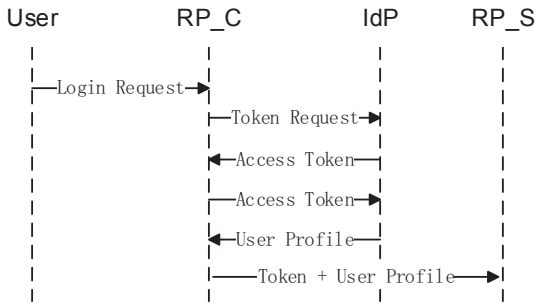


Figure 2: Typical Misuse

The problem is related to the authenticator, no mechanism is provided to protect the integrity and the effectiveness of the authenticators, thus attackers can tamper with or replay the traffic to hijack users' accounts. Such a misuse is fairly common when RPs implement OAuth authentication on their mobile applications. In our observation, 28.4% of the OAuth-capable apps in our dataset on the Android and iOS platform are vulnerable due to this mistake.

2.2 Crux of OAuth Authentication

2.2.1 Authenticator Selection

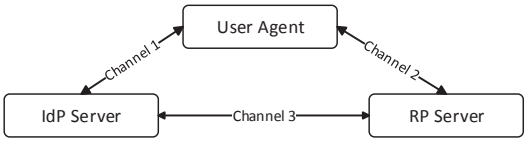
In OAuth authentication, the authenticators should be able to tell an RP the identity information of a user and his presence with the application.

However, there are some common misunderstandings in authenticator selection. Typically, many RPs and researchers thought that an access token can be used as the authenticator. Since the IdP authenticates the user before issuing an access token, it is tempting to think that the token can represent the user's identity. However, the access token is designed to be opaque to the RP. OAuth defines no specific token format, no common set of scopes for the access token, RPs are unable to derive user profile from the access token. For example, if an RP utilizes the token as the authenticator, generally he can exchange the token for user's identity information on the server-side, and bind the IdP-managed user id to the current session with the user-agent. However, an attacker can substitute the victim's token with his own, and the RP will bind the attacker's identity to the current session with the victim's user-agent, i.e., the attacker forces the victim to login to the attacker's account on RP, then he can obtain the victim's personal information.

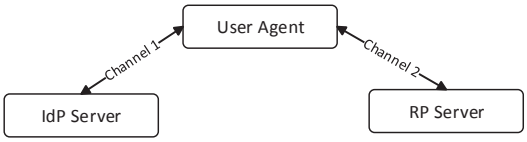
As a consequence, in the context of OAuth authentication, tokens can only be used to exchange for user's identity information, and use the obtained information to authenticate users.

2.2.2 Authenticator Transmission

Another key point in OAuth authentication is authenticator transmission. The authenticators can be transmitted in two ways: (1) **Direct Communication**, the IdP server can transmit authenticators directly to the RP server though channel 3, as sketched in figure 3(a). Authenticators transmitted in this way is secure, as the traffic is invisible to attackers. (2) **Indirect Communication**, as sketched in figure 3(b). As the authenticators are transmitted though channel 1 and channel 2. IdPs can adopt transport security measures to protect channel 1, but not channel 2. If channel 2 is not protected by the RP, attackers can intercept and tamper with the traffic, to masquerade as the user or exchange the authenticators (e.g., access token) for user's protected resources. Meanwhile, If no mechanism (e.g., a signature) is provided to protect the integrity of the authenticators, attackers can tamper with the authenticators during transmission, to hijack a victim's account or entice him to login to an attacker's account.



(a) Direct Communication



(b) Indirect Communication

Figure 3: Server-Server Communication

2.2.3 Server-side Verification

On the server side, the RP is responsible to verify the binding between the authenticators and the current user, as well as the integrity of the authenticators, so as to prevent the impersonation attack. In the example mentioned in Section 2.2.1, if the IdP utilizes a signature to protect the token, and the key to generate the signature is the client secret (a secret shared between the IdP and the RP), the RP can validate the signature to verify the token, however, if he fails the validation, the attacker can still threaten user's privacy.

3. APPROACH

Our approach consists of three empirical studies that examine the OAuth-based authentication mechanisms on different platforms:

- We first focus on understanding how RPs decide to authenticate users with OAuth on different platforms. In this step, we observe the authentication flows implemented by RPs to pinpoint the authenticators selected by RPs, as well as the communication channels. These information plays an important role in the security analysis of RPs' authentication mechanisms.

- Based upon the obtained knowledge, we conducted our study on a representative sample of the most popular applications on the Web, Android, iOS platform, to analyze the security of RPs' OAuth-based authentication mechanisms and the effectiveness of the OAuth SSO protocols provided by different IdPs.

- In addition, we manually analyzed 16 popular relying parties (e.g., IMDB, Feedly), each of these RPs distributes their apps for multiple platforms. For example, Feedly distributes its client application for Web, Android, and iOS. We investigated how RPs implement their OAuth-based authentication mechanisms on different platforms for their client applications, and how the vulnerabilities in an app on a certain platform may affect its siblings on other platforms.

3.1 Data Collection

We looked through the list of Alexa's top 250 most-visited websites, and observed 79 RPs supporting SSO services provided by 27 different IdPs. We took a closer look at the relative popularity of the IdPs, and note that 98.6% of the RPs are served by the top 6 IdPs whose SSO services are integrated by at least 10% of the RPs. Our observation on the Web platform conforms to that on the Android and iOS platform. Table 1 illustrates the top-6 list of global IdPs, and the number of RPs integrating their SSO services on different platforms.

We examined the implementations of the six high-profile IdPs, all of which use OAuth or OAuth-based protocol (e.g., Google OpenID Connect) as their primary protocol. Hence, we decide to investigate the security and privacy issues of the RPs' authentication mechanisms built on top of the OAuth SSO schemes provided by these IdPs.

We note that half of the six IdPs are American sites, while the other half are Chinese sites. It can be explained by the fact that RPs prefer to select IdPs with similar geographic or cultural focus [19]. Ergo, we selected top 100 apps from Google Play and top 100 apps from Chinese mainstream markets as the samples of RP applications on the Android platform, and top 100 apps in the US on iOS store [3], as well as top 100 apps in China [2], as the samples of RP applications on the iOS platform.

Meanwhile, among the RP application samples, we selected 16 popular RPs who distribute their applications on all of the three platforms to help conduct our third study. For each RP, at least one of their apps on a certain platform is included in our aforementioned samples.

3.2 Adversary Model

Our study assumes that the user's computer, browser and mobile devices are not compromised, the IdP is benign, and the communication between the RP and IdP is secured. We do not trust the relying party who can be fully controlled by the adversary. In addition, the client apps installed on a mobile device are not trusted either, any information hard-coded in such a client app can be obtained by the adversary though reverse engineering techniques. Meanwhile, users may download apps from third-party app markets rather than the official app market, such a behavior introduces the possibility of installing repackaged apps or malware.

In our adversary model, the goal of an adversary is to (1) impersonate the victim on the RP application, (2) entice the victim to login to the adversary's account on the RP website. There are two different adversary types considered in this work, which vary on their attack capabilities:

- **A malicious RP attacker** can login to other benign RPs as a normal user, and intercept, modify, replay the traffic of his own. He can also act as a normal RP to gather users' information on his website.

IdP	I/A Rank	Web	Android	iOS	Total
Facebook	1/3	46	30	32	108
QQ	2/8	21	53	28	102
Weibo	3/18	23	41	26	90
WeChat	4/4634	14	44	28	86
Google	5/1	27	20	24	71
Twitter	6/10	8	6	10	32

Table 1: Top-6 list of global IdPs and the number of RPs integrating their SSO services on different platforms. Legends: I: IdP; A: Alexa.

- **An active network attacker** can intercept, modify and replay any unencrypted network traffic between the user-agent and the RP. He can also launch common SSL attacks (e.g., SSL stripping, rogue SSL certificate) to decrypt the HTTPS traffic while the user-agent or RP client implements SSL problematically.

3.3 Methodology

By virtue of the thorny challenges faced when conducting a security analysis of real-world OAuth-based authentication mechanisms, including the lack of access to the source code on the server-side of the RPs and IdPs, the complexity of SDKs and the implicit assumptions to use the SDKs, undocumented design features of the OAuth SSO schemes, etc. We employ a security analysis methodology similar to those used by many others [18, 21], that is, we treat IdPs and RPs as black boxes, and focused on the traffic going through the user-agent during an OAuth authentication transaction. But different in two important aspects:

- First, we investigate the RP's OAuth-based authentication mechanism from a malicious RP attacker's perspective, who can login to other RPs as a normal user and analyze his authentication traffic including the HTTPS-protected traffic going through his own browser, to identify the exploitable vulnerabilities, and then use the pre-gathered user profile and OAuth credentials to hack user's account on other RPs' websites. The methodologies employed by the prior studies [18, 21] can only analyze the unencrypted traffic, and identify the flaws exploitable by an active network attacker, but not other weaknesses we unveiled.

- Second, We adopt a differential analysis method to pinpoint the authenticators utilized by RPs for user authentication. Pinpointing the authenticators help us (1) understand how RPs decide to authenticate users with OAuth on different platforms, (2) investigate fundamental causes of the security and privacy issues introduced by RP's customized OAuth-based SSO mechanisms, and propose practical remedy options that are applicable to the RPs.

Our methodology includes three steps: (1) understanding how RPs decide to authenticate users with OAuth on different platforms, (2) analyzing the security of the RPs' OAuth-based authentication mechanisms and the effectiveness of the OAuth SSO protocols provided by different IdPs, and (3) investigating how vulnerabilities in an RP application on a certain platform may affect its siblings on other platforms.

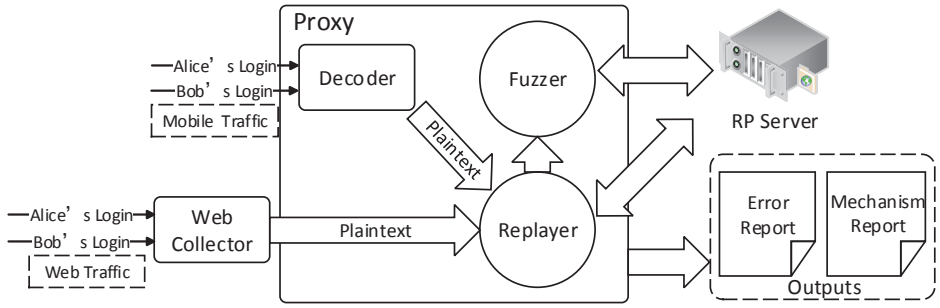


Figure 4: An Overview of How We Analyze the Authentication Mechanisms

To conduct the analysis, we register two test accounts Alice and Bob for each IdP to collect the authentication traffic. The security analyses we execute are within the ethical and legal boundary.

In the first step, we implement a web authentication flow collector as a Firefox add-on. When using OAuth-based authentication mechanisms to login to an RP website, we can use the browser add-on to save all the observed traffic (including HTTP and HTTPS traffic) in pcap format for latter analysis. This add-on enables us to inspect the communication between the RP and the user-agent from an RP's perspective. As for the mobile platform, we pre-install a self-signed SSL certificate on an mobile device, and utilize Mitmproxy [4] as our proxy to decrypt and dump the authentication traffic.

The dumped pcap files serve as the input to the replayer, which is implemented as an Mitmproxy inline script to parse the captured traffic and replay/compare the authentication requests. The replayer examines whether an authentication request can be replayed. If the examination fails, the request is added to an *error report* for manual analysis, otherwise it compares Alice's and Bob's request to identify the parameters with different values, then Alice's request along with the identified parameters in Bob's request are sent to a fuzzer, which is also implemented as an Mitmproxy inline script to modify the request and analyze the responses. The fuzzer replaces one parameter in Alice's request a time with the corresponding parameter in Bob's request, and replays the modified request with other parameters unchanged to the RP. This operation is executed iteratively until all the pinpointed parameters have been checked. Then the fuzzer analyzes RP's response to identify the key parameters used for authentication, and check whether the RP server verifies the binding between the authenticators and the current user. The analysis results of the fuzzer, the transmission protocol (e.g, HTTP, HTTPs) as well as the RP name and the platform type are recorded in a *mechanism report* file, to help subsequent analysis in step (2). Figure 4 illustrates the process of the analysis in step (1).

In the second step, we analyze the authentication traffic from an active network attacker's perspective. We disable the Firefox add-on, remove the forged SSL certificate from the mobile devices, and clean the browser's cache, as well as the IdP and RP clients installed in the mobile device. Then we use Alice's account to login to the RPs. The authentication traffic is captured using the aforementioned proxy. The proxy is configured to forge an SSL certificate to attack the client's SSL implementation by default, as the situation of SSL implementation in mobile apps is far less rosy [12, 17]. A problematic SSL deployment in an RP app can be detected, and its HTTPS traffic will be decrypted by the proxy and treated as the plaintext traffic. We attempt to perform static analysis on the RP's client application if the authentication flow involves client-side logic. A Python program is developed to parse the dumped traffic and identify the exploitable vulnerabilities based on the *mechanism report* generated in the first step. Meanwhile, we manually analyze the *mechanism report*, *error report* and the results obtained in this step jointly to evaluate the selected authenticators and communication channels, as well as RP server's verification on the authenticators. This step answers how the adversary can circumvent the authentication, and whether IdPs' OAuth SSO schemes increase the attack surface.

In the third step, based on the prior knowledge obtained in step (1) and (2), we manually analyze the selected 16 RPs' OAuth-based SSO schemes on different platforms. We assess the apps of each RP on different platforms in turn. First, we use Alice's account to perform login on each app, and capture the authentication traffic. Then we compare the authentication requests/responses generated from different platforms, to examine whether the same app distributed by a certain RP in different platforms employs a consistent authentication mechanism. If a difference exists, we further explore the root causes of the difference by analyzing their client applications, and investigate whether such a difference may compromise the RP's SSO system.

4. MULTI-PLATFORM ANALYSIS AND RESULTS

Using the aforementioned approach, we conducted a multi-platform analysis on the real-world OAuth-based authentication mechanisms. Our study covers OAuth SSO services provided by notable IdPs (e.g., Facebook, QQ, Weibo, WeChat, Google and Twitter), and commercially deployed OAuth-based authentication mechanisms of prestigious websites/applications (e.g., Feedly, PhotoGrid and Youku Video). Overall, we analyzed three sets of apps, including (1) **Web app set**: Alexa's top 250 websites, (2) **Android app set**: top 100 apps on Google Play and top 100 apps in Chinese mainstream markets, (3) **iOS app set**: top 100 apps in the United States and top 100 apps in China on iOS Store. We found that 241 out of the 650 applications implement OAuth-based SSO schemes. Table 3 illustrates the usage of OAuth authentication schemes on different platforms. The relatively lower OAuth-based authentication usage on the

Name	Vulnerability	Attacker type	Consequence	Platform
V_1	Unprotected Authentication	RP Attacker/ Network Attacker	Impersonation/Enticement/ Information Leakage	Android/iOS
V_2	Use publicly accessible information as authenticator (with HTTPS-protected)	RP Attacker	Impersonation/Enticement	Web/Android/iOS
V_3	Client-side protocol logic	Network Attacker	Impersonation/Enticement/ Information Leakage	Android/iOS
V_4	Unprotected Token Refresh	RP Attacker/ Network Attacker	Information Leakage	Web/Android/iOS
V_5	Wrong Token (with HTTPS-protected)	RP Attacker	Impersonation/Enticement	Web/Android/iOS

Table 2: Vulnerabilities uncovered in our study.

web platform can be explained by the use of subdomains for hosting the same service. For example, 39 of the top 250 websites are Google's subdomains hosting the Google Search service for different countries, which share the same login mechanism.

Our study shows that the real-world OAuth-based authentication mechanisms are riddled with security flaws, which may enable an attacker to impersonate a victim on an RP application, or to entice a victim to login to the attacker's account on an RP website. These flaws are diverse and distributed on different platforms, as sketched in Table 2. We found that 32.9%, 47.1% and 41.6% of three sets of apps suffer from the vulnerabilities we uncovered. The results of our investigation are elaborated in the rest of this section.

Scheme	Web		Android		iOS	
	N	%	N	%	N	%
OAuth-based	79	31.6	85	42.5	77	38.5
Login-required	201	80.4	156	78	147	73.5
Total	250	100	200	100	200	100

Table 3: Usage of OAuth authentication on different platforms. N refers to number.

4.1 Real-world OAuth SSO Mechanisms

As many websites and applications use OAuth for authentication, major IdPs provide OAuth SSO SDKs for different platforms to help developers integrate their services. In this section, we introduce how real world RPs customize IdPs' OAuth services for authentication on different platforms, and how they manage identities from multiple IdPs.

4.1.1 Mechanisms on Different Platforms

RPs customize their OAuth-based authentication mechanisms upon their obtained resources, their understandings of the roles of OAuth credentials, their security requirements, as well as their assumptions for different platforms. Table 4 illustrates the mechanisms used on different platforms.

Tendency. We can observe that the authenticators and transmission protocols employed by RPs vary significantly between the Web and mobile platform. The mechanism deployed on the Android and iOS platform are similar. On the mobile platform, RP developers are inclined to select the publicly accessible user profile (e.g., user id, email) as the authenticator. Meanwhile, they prefer to include client-side protocol logic with the mobile apps and generate cryptographic parameters to secure the authentication. Many mobile apps neglect the protection of authenticators during transmission. While on the Web platform, RPs are relatively prudent, they prefer to use the authorization code or access token as the authenticator, many of them also rely on the server-side logic to complete the authentication. Most RPs transmit authenticators with HTTPS protected.

Authenticator selection. Overall, we observed six typical authenticators used by RPs, as listed in Table 4. Apart from the publicly accessible information and access token, authorization code were used by a large number of RPs as the authenticator, which is an one-time value and has a short lifetime. Id token is a specific token used in Google OpenID Connect, it is signed by the client secret and contains the user profile, email address and other protected information, which are encoded in clear-text using Base64. It fits the requirements of authentication well. Not all of the mechanisms depicted in Table 4 are secure to authenticate users. We will explain the details in Section 4.2.

Authenticator transmission. Most RPs employed indirect communication to transmit authenticators and deployed no measures to protect the integrity of authenticators. Some mobile app developers performed cryptographic operations on the client side to process the authenticators to avoid disclosure, such a behavior is not suggested, as discussed in Section 4.2.2. A few RPs utilized a signature to protect the authenticators, there are two situations: (1) the signature is generated in the client app of the RP, this is similar to A_5, (2) the signature is generated on an authorization server of the RP, and the authorization server returns the signature to the client, this situation is similar to A_7, as the user has been authenticated by the authorization server. Using direct communication (A_7) to transmit authenticators is secure and suggested, as the traffic between the IdP server and the RP server is invisible to attackers.

	Mechanism	Web		Android		iOS	
		\	SSL	\	SSL	\	SSL
A_1	user id/email	×	×	√	×	√	×
A_2	auth code	√	√	√	√	√	√
A_3	access token	×	√	√	√	√	√
A_4	uid+access token	×	√	√	√	√	√
A_5	encrypted data	×	-	√	-	√	-
A_6	id token	√	√	×	√	×	√
A_7	server-side logic	√		×		×	

Table 4: Authenticators used on different platforms. Legends: \: no SSL. -: no such case.

Server-side verification. Most RPs provided no server-side verification, because the authenticators cannot be used to verify the binding between them and the current session. We observed 2 apps failed to validate the signatures used to protect the authenticators, and 5 apps failed to verify the binding between the user id and the access token, i.e., an attacker in possession of a valid access token can hijack

victims' accounts by simply altering the user id.

4.1.2 Multi-Identity Management

78.4% (189/241) of the OAuth-capable apps in our dataset support SSO schemes provided by more than one IdPs. For example, *Foxnews.com* supports to be logged in via user's Facebook or Google account. Therefore, how to manage the identities from multiple IdPs appears to be a great challenge. In general, there are three solutions adopted by RPs in our observation: (1) Create a new account and combine identities from multiple IdPs to this account, (2) Create an account for the identity from each IdP, and use an auto-generated string (e.g., *jedi013579*) as the new username, (3) Create an account for the identity from each IdP, and use the username obtained from the IdP as the new username, or ask the user to specify the username/password for RP.

All of the three solutions have some shortcomings. In solution (1), an attacker can compromise RP's entire SSO system by compromising a certain IdP's SSO service, no matter how secure other IdPs' SSO services are. The more IdPs' SSO services supported, the bigger the attack surface. In solution (2), auto-generated usernames make it easier for an enticement attack, it is difficult for a victim to distinguish his own username from the attacker's, especially on mobile platforms where a long username may be displayed incompletely due to a restricted screen size. In solution (3), requesting username/password violates the design goal of OAuth, as many users prefer to use same passwords for different websites/ apps [1], this request may expose user's password for the IdP.

4.2 Security and Privacy Issues

Our investigation shows that OAuth-based authentication mechanisms of many RPs suffer form various vulnerabilities. Table 5 illustrates the percentage of vulnerabilities on different platforms. In this section, we categorize the root causes of the vulnerabilities in Table 2. The results show concretely how RP misused the results of an OAuth-based SSO for its decision making, and how their customization of the published protocols totally crippled the security of OAuth authentication.

Platform	V_1	V_2	V_3	V_4	V_5
Web	0	9.6%	0	2.7%	23.3%
Android	31.7%	34.1%	22.4%	2.4%	16.5%
iOS	27.6%	21%	18.4%	3.9%	19.7%

Table 5: Percentage of vulnerabilities on different platforms.

4.2.1 Blind Faith in HTTPS (V_2 & V_5)

Transmitting authenticators in a plaintext is obviously insecure, especially when the publicly accessible information (e.g., user id, email address) is used for authentication (V_1). We observe that 19.9% of the RPs implemented OAuth authentication in an unprotected manner. However, HTTPS is not a panacea for OAuth authentication either. In our observation, when HTTPS is applied to protect the communication, 0.8% of the RPs based their verification of user identity solely on the user id or email address (A_1), 9.1% of the RP apps submitted the user id along with an access token to their backend servers, but 22.7% of them failed to verify the binding between the user id and the access token (A_4). 5.8% of the RPs simply used an access token for authentication (A_3). These RPs relied completely on HTTPS for security of OAuth authentication.

Even though the mechanisms seem immune to the network attackers, they are vulnerable to an RP attacker. For example, RP A is controlled by an adversary, and he wants to hack a victim's account on RP B, we assume that the victim is also a user on RP A. The adversary can log into RP B as a normal user in his own browser, and observe the authentication process to identify exploitable vulnerabilities, if RP B bases the user authentication on user id/email or access token, the adversary can use the pre-gathered victim's information to tamper with the authenticators and hack the victim's account on RP B.

4.2.2 Client-side Protocol Logic (V_3)

In our observation, using client generated parameters (A_5) as the authenticators is a unique phenomenon on the mobile platform. 13.7% of the RP apps in our dataset adopted such a mechanism to authenticate users.

These RPs realized the need to protect confidential data, and included client-side protocol logic to process the credentials before sending them back to the server. However, instead of using common high-level protocols (e.g., SSL), they customized the OAuth SSO services with cryptographic mechanisms, such as encryption (e.g., DES), hashing (e.g., MD5, SHA1), and signing (e.g., HMAC), to ensure the security of OAuth authentication.

Unfortunately, such a behavior can completely break the security of OAuth authentication and increase the attack surface. This is because an adversary can completely control an app, and analyze how these encrypted parameters and signatures are generated, and correspondingly generate forged messages from the client side. The feasibility has been shown in Zuo et al.'s study in [24]. Meanwhile, as stated in [7], developers often violate principles in security engineering and misuse cryptography in this process, which may induce numerous vulnerabilities. Hence, including security sensitive protocol logic with the mobile application is not recommended.

4.2.3 Offline Access (V_1 & V_4)

Offline access is a unique access service provided by Google. It enables an RP application to access a Google API when the user is not present. Examples of this include backup services and applications that make Blogger posts exactly at 8am on Monday morning. In our analysis, 36.6% of the examined RP apps incorporating Google's SSO service requested offline access permission.

Offline access for Google APIs is achieved through a refresh token, which is issued when a user first visits the app and grants offline access. Our observation shows that RPs are not clear about how to use the refresh token securely. If an RP wants to refresh tokens for long-term access, he should submit the refresh token along with the client secret to exchange for a new access token. As the previous HTTPS session related to OAuth authentication has already been ended, some RPs transmit the refresh token in HTTP, which exposes the client secret to attackers. Meanwhile, even though RPs are educated not to hard code the client secret in their client applications when implementing the authorization code grant flow [13], some of them hard code the client secret in their mobile applications to

refresh tokens. Given that 33.2% of the examined apps utilized the authorization code as the authenticator, and the id token used for authentication is signed by the client secret, an attacker with knowledge of the client secret can intercept authorization codes to hijack users' accounts, or forge id tokens to entice a victim to login to the attacker's account.

4.2.4 Wrong Token (V_5)

Another common confusion amongst RP developers is the roles of different tokens. The "OAuth Threat Model" [14] introduces two types of access token: *Bearer token*, which can be used by any client who has received the token, and *MAC token*, which can only be used by a specific client, and prevent replay attacks when the communications are eavesdropped. When an access token is used for authentication, the RP is supposed to check that the token used to retrieve the user's identity information is granted to the same RP, otherwise an adversary can use tokens issued to a malicious RP to impersonate the victim on a benign RP application (as discussed in the prior section). Hence, if RPs want to use an access token for authentication, they should choose MAC token rather than bearer token.

However, all the access tokens used as an authenticator observed in our analysis were bearer tokens. There is a possibility that IdPs offer *bearer token*s as the only option for the sake of simplicity.

Google OpenID Connect also offers two types of token: *access token*, which is a *bearer token*, and *id token*, in which the user profile and email address are encoded in cleartext using Base64. Google suggests to use the *access token* for authorization and *id token* for authentication. Nevertheless, 20.4% of the apps incorporating Google OpenID Connect utilized the *access token* to authenticate users. The *id token* is supposed to be transmitted under protection, otherwise an adversary can decode it to obtain user profile and other protected resources even without invoking the Google APIs.

4.3 Case Study

To investigate how RPs deploy their OAuth-based authentication mechanisms on different platforms for the same app, we manually analyzed 16 representative RPs. 43.75% (7/16) of them deployed consistent authentication mechanisms on different platforms, three of the seven RPs' mechanisms were vulnerable. Eight of the remaining 56.25% RPs suffered from at least one vulnerability on one kind of the examined platforms. As apps for different platforms share the same back-end database and other resources, one vulnerability on a certain platform can compromise the security of the entire SSO system. In the following, we present two cases to explain the typical vulnerabilities.

Case #1: *Feedly leaks credentials during the authentication and token refresh process*

Feedly is one of the most popular news aggregator application for various web browsers and mobile devices running iOS and Android, with millions of active users.

Feedly uses Google as an identity provider for authentication, and the authentication mechanism is Google OpenID Connect: Feedly requests an authorization code from Google first, and then utilizes the code to exchange for an *id token* to authenticate the user. However, when processing the actual traffic on the web platform, we observed that Feedly's authentication mechanism was based on Google OAuth 2.0, which is designed for authorization. Feedly utilized a bearer token as the authenticator and transmitted it in a plaintext. But even worse, Feedly refreshed the access token in the subsequent communication and leaked the client secret. On the Android platform, we observed that Feedly protected the authentication process with HTTPS. Nevertheless, Feedly for Android failed to protect the client secret, it hard coded the secret in the "refreshToken"function, and the secret was the same as that observed on the Web platform.

In this case, Feedly appeared to be completely ignorant of the role of a bearer token, and neglected the protection of the client secret when refreshing tokens. Because of this flaw, a malicious RP was able to use his pre-gathered access tokens to hijack victims' accounts on Feedly. He could even use the leaked client id/secret to impersonate Feedly, so as to interact with Google and collect user's privacy information. The flaw related to the refresh token is quite unique among the ones we have discovered. As the token refresh process occurs occasionally, how real-world RPs refresh an access token was rarely discussed in the prior studies. This operation does not have much technical subtlety, but it exists in such a significant application. Other RPs should learn lessons from this flaw whether they make the same mistake or not.

Case #2: *Mango TV's wrong authentication mechanism in Android compromised the security of its entire SSO system*

The second case involves Mango TV or imgo.com, which is a top online video and streaming service platform in China, with over one million monthly active users. Mango TV distributes its client application on different platforms, including Web, Android, iOS.

When analyzing Mango TV's authentication schemes based on Weibo's SSO service, we found that it deployed different mechanisms for apps on different platforms.

On the web platform, Mango TV deployed a resource server and an authorization server for authentication. Its client submitted the obtained authorization code to the authorization server. The authorization server requested user's identity information and generated a ticket parameter on the server side, and then redirect the ticket and a signature to the resource server, to complete the authentication. As the tokens and user profile were obtained on the server side, and the ticket used for authentication was protected by a signature, the authentication process was free from known attacks.

On the mobile platform, including the Android and iOS platform, the Mango TV app requested an access token from Weibo first, and utilized the access token along with the user id to authenticate a user. Although the authorization process was protected by HTTPS, the identity information used for authentication was transmitted in HTTP, such a behavior made the previous protection of the access token meaningless. We were able to build a exploit by tampering with the user id and access token to hijack a victim's account on Mango TV easily.

What confused us was that Mango TV's authentication mechanism on the Web platform showed that the developers knew how to authenticate users securely, but they still deployed different mechanisms on different platforms, thus put the entire system in danger.

5. DISCUSSION

We analyzed the status quo of OAuth-based authentica-

tion mechanisms deployed by real-world RPs in the prior section, and uncovered several critical vulnerabilities that may threaten the security of OAuth authentication. In this section, we discuss the advantages and disadvantages of three possible defenses that can be used by RPs to prevent these vulnerabilities, and develop some practical recommendations for IdPs and RPs to secure OAuth authentication.

5.1 Pros and Cons of Possible Mitigations

5.1.1 Certification of Symbolic Transactions

The Certification of Symbolic Transaction (CST) [9] proposal tries to verify a protocol-independent safety property jointly defined over all parties, to avoid the burden of individually specifying every party's property for every protocol. It treats a multiparty transaction as a runtime process for creating a proof obligation for a static program verifier. The certifier logically examines whether the sequence of computations on all parties in this transaction collectively ensures the global predicate. This approach prevents any manipulation on the communications between involved parties. However, it requires modifications to existing clients and servers. When applied to an SSO system, CST requires the client to hold a piece of BrowserSecret, which can be leaked on the mobile platform, thus compromises the security of the entire system.

5.1.2 Secure Channel

Using an existing in-browser communication channel to establish a dedicated, bi-directional, authenticated, secure channel is a channel established is an intuitive idea to protect the communication between an RP and an IdP [8]. It leverages public key cryptography to help an RP and an IdP authenticate each other and share a common secret, i.e., the session key. All further communication between the RP and the IdP is then encrypted with the session key and is kept secret from eavesdroppers. This solution is able to prevent network attackers. However, it cannot defend against a malicious RP inside the browser. Meanwhile, this approach fundamentally relies on the in-browser communication channel, it is not designed to protect the communications on mobile (e.g., Android, iOS) platforms.

5.1.3 InteGuard

InteGuard [23] offers security protection to vulnerable web API integrations. It operates a proxy in front of the service integrator's web site, and perform security checks on the set of invariant relations among the HTTP messages the integrator receives during a transaction. These invariants link multiple HTTP sessions to a transaction and capture their security-critical relations. This proposal may block most of the existing man-in-the-middle attacks, since attackers cannot impersonate the client. However, its policy checking step can introduce some additional states for an SSO transaction, which makes the system more vulnerable to a distributed denial of service (DDoS) attack. RPs should assess whether this is an acceptable trade-off.

5.2 Recommendations

Some mitigations are robust against our discovered vulnerabilities, but none of them are available for RPs to use yet. Hence, we develop some practical recommendations to help RPs address the problems existing in current mechanisms and design more robust authentication mechanisms based on OAuth.

- **Enforce authenticator integrity protection**: One reason the OAuth-based authentication mechanisms are vulnerable to the impersonation attack is that the integrity of authenticators is compromised by attackers. RPs should include a value (e.g., a signature) that binds the authenticator to the current user (i.e., the current session), to protect the integrity of authenticators. Even if an attacker can tamper with the authenticator, he is unable to generate the hash, thus fails to impersonate the user.

- **Deploy countermeasures against replay attacks**: RPs should include a timestamp or use a MAC token to prevent the replay attack. The authorization code is restricted to one-time use. Similar restrictions should be applied to the authenticators. Otherwise attackers can replay the authentication traffic to masquerade as the user.

- **Do not include client-side protocol logic**: When implementing authentication mechanisms with OAuth in mobile applications, RPs should not include client-side protocol logic. As the mobile apps can be completely controlled and analyzed by attackers, it is feasible to forge cryptographically consistent messages from the client side, as shown in [24]. Meanwhile, Cai et al. [7] showed that developers often violate principles in security engineering and misuse cryptography when including security sensitive logic with the client-side application. RPs are suggested to use server-side logic to obtain authenticators.

Furthermore, we recommend that RPs can adopt our analysis methodology (as publicized in this paper) to detect vulnerabilities of OAuth-based authentication on different platforms. In the meantime, we suggest IdPs to provide standard authentication mechanisms (e.g., OpenID Connect) for RPs. The purpose of their SSO services is to help RPs authenticate users securely with OAuth, but the prevalence of RPs' insecure OAuth-based authentication mechanisms indicates the failure of the SSO service integration. In our observation, understanding the complexity of the OAuth protocol and customizing the home-brewed protocols are the common source of problems for RP developers, the IdPs are responsible to offer guidance to RPs, to help them authenticate users in a secure way with OAuth.

6. RELATED WORK

Extensive research has been conducted to analyze security of OAuth in recent years. Sun et al. [18] conducted an empirical analysis of real-world OAuth implementations, and discussed the impact of classical web attacks on OAuth. Chen et al. [10] analyzed the major differences of OAuth implementation between the web and mobile platform, they revealed several vulnerabilities in OAuth-capable apps on mobile devices caused by developer's misinterpretation of the OAuth protocol. Wang at al. [20] proposed a vulnerability assessment framework of OAuth implementations on the Android platform. In [11, 15, 16, 22], further attacks on OAuth implementations were discovered and reported.

Instead of focusing on the authorization process or the exchange of the OAuth credentials, our work aims to provide deeper insights into the OAuth authentication process.

The issues with SSO have been considered by many researchers. Wang et al. [21] studied the security quality of popular web SSO systems and identified several implementation flaws related to token verification. They also introduced a systematic process for identifying critical assumptions in SDKs, and identified some exploits in apps constructed by importing these SDKs [22]. Bai et al. [5] proposed a framework to extract the authentication protocol specifications automatically, and found security flaws in several SSO systems. They further explained the risk of backing up OAuth authenticators on Android [6]. Unlike previous studies, the focus of our work is not on the generic SSO protocols, nor individual attacks, but rather how RPs authenticate users with OAuth on different platforms, and why some interpretations are correct while others are not.

7. CONCLUSION

In this paper, we report a security study of real-world OAuth-based authentication mechanisms employed by websites and mobile applications. The study shows that these mechanisms lack sufficient verification and are vulnerable to many attacks including user impersonation, client impersonation, enticement attack, etc. We also reveal that RPs have different tendencies to authenticate users with OAuth on different platforms, and 32.9%, 47.1% and 41.6% of the examined apps on the Web, Android and iOS platform suffer from various vulnerabilities. We then categorize the root causes of these vulnerabilities and develop some practical recommendations for RPs to secure their OAuth-based authentication mechanisms.

8. REFERENCES

[1] 55% of Net Users Use the Same Password for Most, If Not All, Websites. https://nakedsecurity.sophos.com/2013/04/23/users-same-password-most-websites/.

[2] iOS Top App Charts (China). https://www.appannie.com/apps/ios/top-chart/china/overall/.

[3] iOS Top App Charts (US). https://www.appannie.com/apps/ios/top-chart/united-states/overall/.

[4] MitmProxy. https://mitmproxy.org.

[5] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In *NDSS*, 2013.

[6] G. Bai, J. Sun, J. Wu, Q. Ye, L. Li, J. S. Dong, and S. Guo. All Your Sessions are Belong to us: Investigating Authenticator Leakage through Backup Channels on Android. In *ICECCS*, 2015.

[7] F. Cai, H. Chen, Y. Wu, and Y. Zhang. AppCracker: Widespread Vulnerabilities in User and Session Authentication in Mobile Apps. In *MOST*, 2014.

[8] Y. Cao, Y. Shoshitaishvili, K. Borgolte, C. Kruegel, G. Vigna, and Y. Chen. Protecting Web-Based Single Sign-On Protocols against Relying Party Impersonation Attacks through a Dedicated Bi-Directional Authenticated Secure Channel. In *RAID*, 2014.

[9] E. Y. Chen, S. Chen, S. Qadeer, and R. Wang. Securing Multiparty Online Services via Certification of Symbolic Transactions. In *S&P*, 2015.

[10] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague. OAuth Demystified for Mobile Application Developers. In *CCS*, 2014.

[11] H. E and L. A. How We Hacked Facebook with OAuth2 and Chrome bugs. http://homakov.blogspot.ru/2013/02/hacking-facebook-with-oauth2-and-chrome.html.

[12] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben. Why Eve and Mallory Love Android: An Analysis of Android SSL (In) Security. In *CCS*, 2012.

[13] IETF. The OAuth 2.0 Authorization Framework (RFC 6749), 2013.

[14] T. Lodderstedt, M. McGloin, and P. Hunt. OAuth 2.0 Threat OAuth Model and Security Considerations. 2013.

[15] R. Paul. Compromising Twitter's OAuth Security System. *Technical report, Ars Technica*, 2010.

[16] E. Shernan, H. Carter, D. Tian, P. Traynor, and K. R. B. Butler. More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations. In *DIMVA*, 2015.

[17] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. SMV-hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *NDSS*, 2014.

[18] S.-T. Sun and K. Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *CCS*, 2012.

[19] A. Vapen, N. Carlsson, A. Mahanti, and N. Shahmehri. Third-party Identity Management Usage on the Web. In *PAM*, 2014.

[20] H. Wang, Y. Zhang, J. Li, H. Liu, W. Yang, B. Li, and D. Gu. Vulnerability Assessment of OAuth Implementations in Android Applications. In *ACSAC*, 2015.

[21] 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 *S&P*. IEEE, 2012.

[22] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *USENIX Security Symposium*, 2013.

[23] L. Xing, Y. Chen, X. Wang, and S. Chen. InteGuard: Toward Automatic Protection of Third-Party Web Service Integrations. In *NDSS*, 2013.

[24] C. Zuo, W. Wang, R. Wang, and Z. Lin. Automatic Forgery of Cryptographically Consistent Messages to Identify Security Vulnerabilities in Mobile Services. In *NDSS*, 2016.