# Breaking and Fixing Mobile App Authentication with OAuth2.0-based Protocols

Ronghai Yang(✉), Wing Cheong Lau, and Shangcheng Shi

Department of Information Engineering, The Chinese University of Hong Kong, Hong Kong, China
{yr013,wclau,ss016}@ie.cuhk.edu.hk

**Abstract.** Although the OAuth2.0 protocol was originally designed to serve the authorization need for websites, mainstream identity providers like Google and Facebook have made significant changes on this protocol to support authentication for mobile apps. Prior research mainly focuses on how the features of mobile operating systems can affect the OAuth security. However, little has been done to analyze whether these significant modifications of the protocol call-flow can be well understood and implemented by app developers. Towards this end, we report a field-study on the Android OAuth2.0-based single-sign-on systems. In particular, we perform an in-depth static code analysis on three identity provider apps including Facebook, Google and Sina as well as their official SDKs to understand their OAuth-related transactions. We then dynamically test 600 top-ranked US and Chinese Android apps. Apart from various types of existing vulnerabilities, we also discover three previously unknown security flaws among these first-tier identity providers and a large number of popular 3rd-party apps. For example, 41% apps under study are susceptible to a newly discovered profile attack, which unlike prior works, enables remote account hijacking without any need to trick or interact with the victim. The prevalence of vulnerabilities further motivates us to propose/implement an alternative, fool-proof OAuth SDK for one of the affected IdPs to automatically prevent from these vulnerabilities. To facilitate the adoption of our proposed fixes, our solution requires minimal code changes by the 3rd-party-developers of the affected mobile apps.

**Keywords:** OAuth2.0 · OpenID Connect · Mobile app authentication

## 1 Introduction

The OAuth2.0 protocol was originally designed to serve the authorization need for 3rd-party websites. However, many major Identity Providers (IdPs) such as Facebook, Google and Sina, have recently adapted the OAuth2.0-based protocols to support Single-Sign-On (SSO) services for 3rd-party mobile apps (which take the role of Relying Party under the context of OAuth2.0). When OAuth2.0 is used as a SSO scheme, a user can log into the mobile Relying Party (RP), *e.g.*,


D. Gollmann et al. (Eds.): ACNS 2017, LNCS 10355, pp. 313–335, 2017.
DOI: 10.1007/978-3-319-61204-1_16

IMDB, via the IdP without sharing the identity credential with the RP. In this paper, we will focus on OAuth2.0 and OpenID Connect (which is built on top of OAuth2.0), since they are the *de facto* SSO standards.

To support SSO services with 3rd-party mobile apps, the security of the adapted OAuth2.0 protocol has been evaluated by the literature. Chen *et al.* [12] first point out that some operating-system-provided components (*e.g.*, Intent/WebView for Android) are required to implement OAuth2.0 for mobile platforms. As further shown by Chen, the features of these components, if not well understood by mobile app developers, can be leveraged to compromise mobile SSO systems. Following this work, Ye *et al.* [34] apply model checking method to theoretically evaluate this modified protocol and Wang *et al.* [28,29] summarize those known vulnerabilities among 15 Chinese IdPs over different platforms.

Prior studies mainly focus on how the differences of mobile systems (*i.e.*, vulnerabilities in the system-provided components) can compromise mobile SSO systems. However, the security implications resulting from the major protocol changes, when adapting the OAuth2.0-based protocols to mobile platforms, are often left out. For example, the standard OAuth2.0 implicit flow has shown to be insecure for authentication and thus a revised version, *i.e.*, a variant of OpenID Connect (OIDC), is recommended. Nevertheless, they do not consider that the SSO results, even in the revised version, are passed through the user device. Consequently, these security-critical results are subject to tampering and further enable an adversary to infer the program logic of the RP server.

Towards this end, the goal of this research is to further the understanding of (1) how the changes of OAuth[1] protocol flow, if not well implemented, can lead to nontrivial security flaws, (2) the overall security quality of mobile SSO systems by checking whether existing vulnerabilities have been fixed or not. Our work consists of two pillars: (1) We perform a standard static code analysis of three first-tier Android IdP apps (Facebook, Google and Sina) and their corresponding SDKs widely used by the RP to understand their client-side program logic. (2) We develop a tool to dynamically test Top-600 US and Chinese Android apps to see how well the RP/IdP servers perform the OAuth transactions. From these studies, we have made the following technical contributions in this paper:

- We have identified the security-critical changes of the OAuth protocol call-flow.
- We have examined the implementations of 3 first-tier IdPs and 600 top-ranked US/Chinese Android Apps. In addition to different types of existing vulnerabilities, we have discovered three previously unknown vulnerabilities resulting from the incorrect implementations of the protocol call-flow modification.
- We have designed and implemented a foolproof solution which prevents future 3rd party app developers from committing the same mistakes. To ease the transition, our solution only requires minimal changes in the 3rd-party developed codes of a vulnerable mobile app.

[1] We use OAuth to denote OAuth2.0 and OpenID Connect, if not specified otherwise.

## 2 Background

In the OAuth ecosystem, four parties are involved to support SSO for 3rd-party mobile apps, namely, the backend server of the 3rd-party mobile app (RP server, for short), the backend server of the IdP (IdP server), the 3rd-party client-side mobile app (RP app) and the client-side mobile app of the IdP (IdP app). For ease of presentation, in the rest of this paper, we use notations in the parentheses to denote these four parties and use OAuth to denote OAuth2.0 as well as OIDC, if not specified otherwise.

The ultimate goal of SSO is for the IdP server to issue an identity proof, *e.g.*, the access token for OAuth2.0 and the id_token for OIDC, to the RP server. With this identity proof, the RP server can determine the user's identity and then log the user in.

### 2.1 The Implicit Flow of OAuth 2.0 for Mobile Platforms

OAuth2.0 [18] defines four types of authorization flows, out of which the implicit flow and authorization code flow[2] are widely used by the mobile platform and the website, respectively. Thanks to the demystification by Chen *et al.* [12], the standard implicit flow has proven to be insecure for authentication under mobile platforms. After revising the standard protocol, one believed-to-be-secure realization is illustrated in Fig. 1, although neither the RFC nor the IdP provides a complete call-flow diagram.

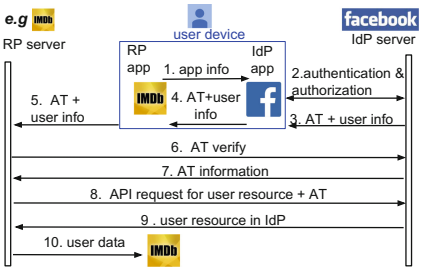



**Fig. 1.** The implicit flow of OAuth2.0 for mobile platforms

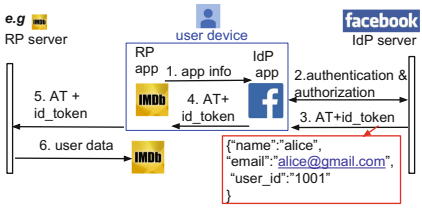



**Fig. 2.** The implicit flow of OpenID Connect

1. A user attempts to log into the RP with the IdP. The RP app sends its app information (*e.g.*, requested permissions) to the IdP app via a secure channel provided by the mobile operating system, for example the *Intent* channel in Android.

[2] In fact, the authorization code flow can also be securely used for mobile apps, but with the cost of worse performance.

2. Thanks to the secure channel, the IdP app can verify whether the RP app information is correct or not. If so, the IdP app then sends such information, as the authorization request, to the IdP server.
3. The IdP server compares information of the authorization request and that pre-registered by the RP developer. If it matches, the IdP server would believe the validity of the authorization request and thus issue an access token (AT) together with optional user information (*e.g.*, uid) to its own client-side app.
4. The IdP app returns the access token to the RP app via the secure channel maintained by the operating system.
5. The RP app sends the access token and user information to the RP server.
6. The RP server should call the security-focused SSO-API provided by the IdP to verify the access token.
7. If the access token is valid, the IdP server should respond the RP server with authorization information including which RP this access token is issued to.
8. Only if the access token belongs to this RP can the RP server accept it. Thereafter, the RP server can retrieve the user data with this verified access token.
9. The IdP server returns the user data associated with the access token.
10. The RP server can then identify the user and return his sensitive data to RP app.

### 2.2 The OpenID Connect Protocol

Since OAuth2.0 was originally designed to support authorization, to adapt it for authentication, it involves multiple high-latency round trips, *i.e.*, Step 6–Step 9 of Fig. 1. To support authentication more efficiently, IdPs like Google and Facebook have developed the OpenID Connect (OIDC) protocol [25] and its variants, on top of OAuth2.0. In addition to the authorization code flow and implicit flow, OIDC further supports another type of authorization flow, *i.e.*, hybrid flow. Regardless, *ALL* the real-world OIDC-enabled apps under study only implement the implicit flow. As such, we will focus the implicit flow in the rest of this paper.

Regarding the implicit flow, OIDC is backward compatible with OAuth2.0. The only difference is that, apart from the access token, IdPs also introduce a new parameter, *i.e.*, *id_token*, which is digitally signed by the IdP server. As illustrated in Fig. 2, the *id_token*, which consists of the user profile, is then sent to the RP server along with the original access token. Since the signature cannot be tampered/forged by an attacker, the RP server can now directly identify the user by extracting the user profile from the *id_token* without the trouble to retrieve the user profile from the IdP server.

### 2.3 Threat Model

The goal of an adversary is to break the mobile app authentication, *i.e.*, log into the RP app as the victim. Here, we trust the mobile operating system and

assume it cannot be compromised. We assume the adversary can install the RP app and IdP app in her own device so as to communicate with the RP server and IdP server. The adversary, Eve, can act as a normal user and monitor/tamper the network traffic through her own device. In addition, we consider that an attacker can trick a user into installing a malicious app on her device. This app does not have any permission considered to be dangerous.

## 3 Major Protocol Changes that Affect Mobile OAuth Security

Although the above OAuth protocol seems simple, some security-related changes (when adapting the protocol from the website) are often overlooked by mobile app developers.

### 3.1 Untrusted Identity Proof

Websites typically employ the authorization code flow to support SSO services. In this case, the identity proof is transmitted via a secure HTTPS channel between the RP server and IdP server. On the contrary, mobile developers advocate the implicit flow, which passes the identity proof through the user device. Since the mobile device is untrustworthy (the attacker has full control of her own device), the identity proof is subject to tampering. Therefore, such an identity proof can only be accepted by the RP server for two reasons:

1. The RP server makes a direct server-to-server call to the IdP server to verify the identity proof (*i.e.*, Step 6–Step 9 in Fig. 1);
2. The identity proof is signed by the IdP server (*i.e.*, *id_token* in Fig. 2).

In other words, the RP server should establish a direct trust relationship with the IdP server to correctly process the identity proof.

Furthermore, there are different types of the so-called identity proof including the access token, the id_token, the user profile (returned at Step 9 of Fig. 1) and even the authorization code (if the authorization code flow is used). Since this notion is neither covered by the protocol specification nor research studies, RP developers need to determine which identity proof to use in which way, based on their own understanding of the protocol.

### 3.2 Heavy Client-Side Logic

Another major difference is about the user-agent. In the web-based SSO services, the user operates on the end-user's browser. By contrast, in the mobile SSO systems, the user interacts with the RP app and IdP app (the browser is split into two parts). Since mobile apps are more powerful, the RP app and IdP app are often responsible for more message exchange which instead is managed by the backend server in the case of website. This seems reasonable at the first sight. But some messages can only be processed at the server side, such as the user

resource retrieved at Step 9 of Fig. 1. Otherwise, it can lead to the user-profile attack, as presented in Sect. 5.

Meanwhile, both the IdP app and RP app store much more data on the client side. For example, the IdP app keeps the user's identity information and the RP app stores the authorization information (*e.g.*, the RP's name), which is displayed for the user to check/grant the permissions. Note that these security-critical data need to be retrieved from the server during SSO transactions. However, with such information in the client side, it may be tempting for an app developer to retrieve it from the phone directly. This can lead to the so-called user-profile attack and inconsistent RP app identity as presented in Sect. 5.

## 4 Our Approach

To evaluate the security implications resulting from the above protocol changes, we first perform dynamic testing on every RP/IdP app. The testing helps to understand the program logic on the server side. Secondly, to better understand the security practices on the client side, we conduct an in-depth static code analysis on the IdP apps (*i.e.*, Facebook, Google and Sina) and their corresponding SDKs (used by RP apps). Given the limited number of IdP apps and SDKs, we can afford for the manual code examination.

### 4.1 Dynamic Testing

We design a tool to automatically fuzz every OAuth-related message. As shown in Fig. 3, we first set up a man-in-the-middle (MITM) proxy so that we can observe and tamper the network traffic going into and leaving our phone. We then manually operate the phone as a normal user and mimic the SSO process to generate a series of OAuth requests. For every request, our tool replaces each parameter with that from a different RP and user. Finally, the tool sends the fuzzed request to the receiver and checks whether the response is normal or not. Note that since the network traffic is typically protected by HTTPS, we employ the SSL-enabled proxy like *mitmproxy* [3].

The analysis of the communication between the IdP app and IdP server is straightforward, since such interactions share the same format for the same IdP. Unfortunately, fuzzing the messages between the RP app and RP server, *i.e.*, Step 5 (ii) in Fig. 3, is more challenging than expected. Firstly, there are numerous interactions between the user mobile device and the RP server. It is therefore difficult to identify which request is used by the RP server to authenticate the user. Secondly, besides being protected by HTTPS, the message exchanges between the RP app and its backend server are often further encrypted or signed by the RP developer. Although it is possible to extract the cryptographic key from the Android app, such a practice may not be scalable. Therefore, it is usually easier to tamper the response from the IdP server to the IdP app, *i.e.*, Step 3 (ii) instead. After all, all OAuth-related information received by the RP server can only be derived from the IdP server[3].

[3] One exception is Google's Android account management, as presented in Sect. 5.1.

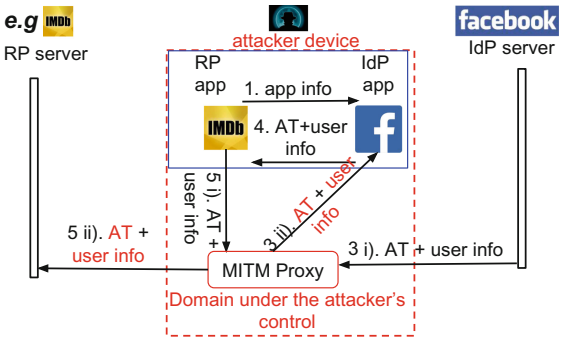



**Fig. 3.** The platform to analyze the implementations on the IdP/RP server side

### 4.2 Static Code Analysis

To understand the client-side logic, we decompile the binary code of the latest IdP app, and the official SDK (if it is compiled) widely used by the RP app. Although the IdP app and SDK usually are heavily obfuscated, the names of special activities and system APIs (*e.g.*, startActivity, getCallingPackage, *etc.*) are not changed. As such, we can identify the entry point of SSO services and then build a partial call flow graph. This provides an opportunity for us to only focus on the relatively small number of SSO-related security-critical activities. We then manually examine these activities to identify potential vulnerabilities. For example, we find that the SSO entry of Sina (*i.e.*, SSOActivity) by default verifies the received information, except when the information is from Sina itself. This practice is seemingly sound at first. However, it may be leveraged by a malicious RP app to bypass the security checking, if there is a "next Intent" [31] in Sina app. Therefore, during the code examination process, we will try to check the existence of the "next Intent". To confirm these vulnerabilities, we then build a toy RP app with the official SDK and launch the corresponding attacks on this toy app. By this way, we have identified the problem of inconsistent RP identity, as presented in Sect. 5.2.

## 5 Vulnerability Analysis

In addition to various existing vulnerabilities, the above method also helps to discover three unknown security flaws resulting from the inaccurate understanding/implementations of the protocol changes.

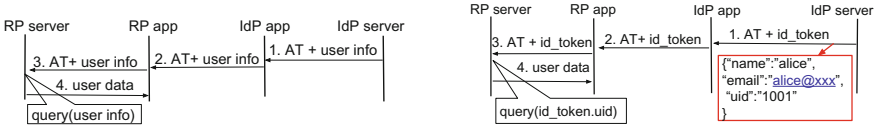
### 5.1 Profile Vulnerability

The identity proof is often incorrectly processed by the RP server and has led to the profile vulnerability, which enables an attacker to log into a susceptible RP as the victim by leveraging the victim's public user profile only. Note that all the


(a) RP app returns user information obtained via Android account management system

(b) The RP app returns user information obtained via API

**Fig. 4.** The RP app does not return correct identity proof of the user to the RP server




(a) No verification of the access token by the RP server

(b) No verification of the signature by the RP server

**Fig. 5.** The RP server does not verify the identity proof of the user

prior findings [12,29,34] require different types of interactions with the victim. In contrast, this newly discovered vulnerability can be exploited remotely and solely by the attacker without any need to trick or interact with the victim, for example via phishing attacks.

**Different Types of Incorrect Implementations.** We present two types of common but widespread mistakes (but the real-world misuses do not limited to these two types).

*Return Incorrect Identity Proof to the RP Server.* Some RP apps can directly retrieve the user information from the mobile device it is running on, regardless of the OAuth access token obtained from the IdP. Without the access token, the RP app only sends the user profile (*e.g.*, uid, email) to the RP server as the identity proof. As a consequence, the RP server has no way to correctly identify the user.

One interesting misuse is caused by Android Account Management System (AMS) [2,15] when using Google as the identity provider. Android AMS provides a centralized database (*i.e.*, */data/system/users/0/accounts.db*) for storing user accounts. While the main goal of AMS is to support seamless access user data via background synchronization, Google has integrated it to support SSO service. Specifically, when a user logs into his/her Google account, Google Login Service (*i.e.*, IdP app) will store the user's Google account information in the *accounts* table as shown below.

```
INSERT INTO "accounts" VALUES(1, 'alice@gmail.com', 'com.
    google', 'password', NULL);
```

With the Android permission of *GET_ACCOUNTS*, an app can easily get the user's Google account (*i.e.*, email address) by calling *getAccounts()* method. As shown in Fig. 4(a), some RP developers (*e.g.*, one antivirus app Psafe with 50+ million installs) directly return this email address (retrieved from the database) to the RP server as the identity proof, regardless of the access token (or id_token). As such, an adversary can insert a forged entry, *i.e.*, *victim's* email address, into the database on the *adversary's* mobile device.

Another typical example is shown in Fig. 4(b) where an RP app immediately retrieves the user data by calling the IdP API with the access token. However, this RP app only sends the user profile to its server as the identity proof. Although such a proof is protected by encryption/signature techniques in addition to TLS, an attacker can simply feed incorrect user information at Step 4 of Fig. 4(b).

*The RP Server Does Not Verify the Identity Proof.* As shown in Fig. 5(a), when the IdP servers return the user identity information (*e.g.*, user id/email address) along with the access token via OAuth, many RP servers (*e.g.*, Sohu with 80+ million monthly-active-user *etc.*) simply and incorrectly return sensitive user information to its own client-side app based on the received user-id WITHOUT verifying whether the received user-id is indeed bound to the issued access token (*i.e.*, lack of Step 6–Step 9 in Fig. 1).

Figure 5(b) shows another case where Facebook and Google adopt the OIDC-like protocol and digitally sign the user identity information. However, most RP servers just ignore this signature and insist on the traditional OAuth protocol. Worse still, some RP servers (*e.g.*, a free call/text app DingTong with 10+ million installs) even do not verify the signature but simply extract the user-id from the payload of the signature and accept the user-id as is the way without any authentication/validation.

**Exploiting the Profile Vulnerability.** Leveraging the same system setup of Fig. 3, an attacker can log into a susceptible app as the victim by exploiting the victim's profile with the following steps[4]:

1. The attacker setups a SSL-enabled MITM proxy for her own mobile device to monitor and tamper network traffic going into and leaving from her device.
2. The attacker installs the vulnerable RP app in her own mobile device.
3. The attacker signs into the vulnerable RP app with the attacker's own IdP login name and password.
4. When the IdP server returns the user profile to its client-side app, *i.e.*, Step 3 (ii) of Fig. 3, the attacker substitutes her own user-id (public user id for the case of Google+ and Sina users or guessable email address) with the victim's one using the MITM proxy. Although Facebook has started to issue private per-app user-id for each RP since May 2014, for backward compatibility reasons, to-date, Facebook still uses the public user-id to identify early adopters

[4] For the case of Google Android AMS, an attacker just needs to insert a forged entry in her own device and then follows the normal steps to complete the SSO procedure.

of a RP app. As such, a user of a vulnerable RP of Facebook is still susceptible to our attack as long as he/she has signed into the RP app via Facebook before May 2014.

5. Since the RP server directly uses the user identity information returned by its client-side app to identify the user WITHOUT further validation, the attacker can therefore successfully sign into the RP as the victim.

*Additional Challenges for the Exploit.* The above exploit involves additional challenges when the IdP client-side app, e.g., the one by Facebook, applies the certificate pinning. In this case, the message sent by the IdP server (and then tampered by the attacker's MITM proxy) to its client-side app will not be accepted by the latter. As a workaround, the attacker can simply uninstall the IdP app so that the IdP SDK (widely used by RP apps) would automatically downgrade to carry out OAuth authentication via the built-in WebView browser. Being a general built-in browser, the WebView does not support the certificate pinning for a specific IdP.

But some IdPs do not support WebView. In this case, the attacker cannot just use off-the-shelf tools like Xposed SSLUnpinning [6] module, since the IdPs often use customized methods (instead of the native Android framework) to implement the certificate pinning. For such IdPs, the attacker has to reverse engineer the IdP app and manually remove the certificate pinning. To demonstrate the feasibility of this approach, we have successfully implemented a proof-of-concept hack on the Facebook app by reverse engineering the apk. But the RP app (more precisely, the IdP SDK) will also compare the certificate of Facebook with a previously hard-coded one (*i.e.*, the real certificate of Facebook). As such, we also need to bypass the certificate comparison function.

**Vendor Responses.** All of three IdPs under study acknowledged the security issue and pledged to help to notify the affected third-party app developers. In particular, Sina already sent a specific notification to ALL RP developers on its platform to inform them about the problem. The company also granted us the maximum amount of reward credits allowed by their bug-bounty program. It has also updated the Single-Sign-On section of its programming guide accordingly. Google has acknowledged our finding via their Google Security Hall of Fame and indicated that they will modify the corresponding documentation for their 3rd party app developers. Facebook has informed us that they are seeking a way to make their RP developers aware of this problem.

### 5.2 Inconsistent RP App Identity for the User

After authenticating the user at Step 2 of Fig. 1, the IdP app should retrieve the authorization information including the RP name, the requested permissions from its backend server. From the perspective of the user, the IdP app will pop up a dialog to indicate which RP requests what permissions from which user. Here, the name of the RP represents the identity of the RP which is verified by the IdP.

Since this authorization information is also stored by the RP app on the mobile device, we find that Google immediately retrieves such information from the device. Specifically, Google presents the name of the RP according to the value of *android:label* in the *AndroidManifest* file, when an RP app adopts the Intent[5] scheme (which is the default method used by Google's official SDK) to support SSO services. On one hand, Google in fact can correctly learn the identity of the RP. On the other hand, the *AndroidManifest* file can be arbitrarily defined by the RP app. As such, the displayed authorization page is inconsistent with Google's understanding.

Taking the advantage of this inconsistence, a malicious RP app can convince the user that the interacting RP, as verified by Google, is a benign and privileged app like IMDB. Due to the great trust placed on Google, the user is willing to grant the permissions if Google verifies the RP app as IMDB rather than some random app. As such, it would be easier for the malicious RP to obtain an access token. This access token enables an attacker to retrieve the victim's data hosted by Google. When combined with other attacks, *e.g.*, token hijacking, the attacker can also log into the victim's account on other Google-enabled benign apps.

**Vendor Response.** Google acknowledged this security bug. But as Google security team claims, this security issue was found independently by another concurrent work (but this issue is not fixed yet). Unfortunately, only the first report is in the scope of Google vulnerability reward program.

### 5.3 Treat the IdP App as a Special RP

Some IdPs including Google and Sina treat their client-side app as a special RP. If the user can successfully authenticate with the IdP, the IdP server would issue an access token to its client-side app. Note that this access token has higher privileges. It enables the IdP app to make sensitive transactions on behalf of the user, for example, to issue access token for any other RPs, *etc.* Since the adversary (acting as a normal user) can also obtain this access token, the adversary can easily launch application impersonation attack [20]. For example, the adversary can utilize this highly privileged access token to invoke sensitive APIs (*e.g.*, query the user information in a batch) with higher API quota. Note that such APIs/quotas originally are not allowed by the adversary or users.

Worse still, this privileged access token is not protected by HTTPS for Sina app. Therefore, an adversary can easily obtain it via eavesdropping. Using this special access token, the attacker can pretend to be the victim and make any transactions, for example, signing into any RP app on the Sina platform as the victim.

**Vendor Response.** We have reported this issue to Sina. Sina directly acknowledged this problem and have applied the corresponding fixes for their Android app.

[5] When an RP app chooses to use the WebView scheme, Google can correctly get the RP name from its own server.

## 6 Empirical Evaluation

We have studied the Android IdP apps and OAuth SDKs provided by three top-tier IdPs, *i.e.*, Facebook, Google and Sina. The number of registered users in these IdPs ranges from more than 800 million to over 2.5 billion as depicted in Table 1. We then comprehensively test the implementations of 600 top-ranked Android applications in US and China. Since more Chinese RPs support OAuth, we only select top 100 RPs (in overall category) and another top 100 apps in different categories for Sina from one major Chinese app store [4]. By contrast, we select 300 top-ranked RPs (in overall category) and 100 top-ranked RPs (in different categories) for Facebook and Google from Google Play. The top 100 apps in different categories is selected as follows: top 30 free apps in social, top 30 free apps in travel and local, top 30 free apps in fitness, top 10 free apps in communication. Out of these 600 apps, we identify that 182 apps use OAuth authentication service provided by one or more of the 3 IdPs mentioned above.

**Table 1.** Statistics for the usage of the protocol

| IdPs (# of third-party RP) | # of IdP users (in Millions) | OAuth2.0 | | OpenID Connect | | |
|---|---|---|---|---|---|---|
| | | Insecure usage | Correct usage | Ignore id_token | Not verify id_token | Correct usage |
| Facebook (59) | >1,500 | 9 | 10 | 35 (20*) | 2 | 3 |
| Google (40) | >2,500 | 24 | 14 | $0^{+}$ | 0 | 2 |
| Sina (83) | >8,00 | 78 | 5 | N.A | N.A | N.A |

– *: 20 out of 35 RPs are incorrectly implemented.
– $^{+}$: Google customizes the OIDC protocol where typically only the id_token is issued to the RP. Therefore, this id_token cannot be ignored. Otherwise, there is not a valid identity proof.

In addition to different types of vulnerabilities, our studies also present first-hand information regarding the adoption rate as well as the misuse rate of OAuth2.0 and OIDC. As shown in Table 1, all of three IdPs support the OAuth2.0 protocol. Facebook and Google additionally develop and advocate OIDC-like protocols for SSO services. Since OIDC by default is supported by Facebook SDK, 68% apps of Facebook, as opposed to only 2 RP apps of Google, employ the OIDC protocol. Regardless, there are two types of misuses for these OIDC-enabled RPs:

– Ignore *id_token*: Some RPs ignore the *id_token*, in which case these RPs revert to the OAuth2.0 protocol and rely on the access token to authenticate the user. As such, these RPs (20 out of 35) share the same security issues of OAuth2.0 as illustrated in Table 2.
– Not verify *id_token*: Some RPs indeed rely on the *id_token* to verify the user. But 29% of them do not check whether the *id_token* is well signed.

**Table 2.** Statistics of the vulnerabilities for OAuth

| IdPs (# of 3rd-party RP) | Profile attack | Token hijacking | Improper user agent | Access token disclosure | App secret disclosure | # of vulnerable RPs[a] |
|---|---|---|---|---|---|---|
| Facebook (59) | 9 (15%) | 27 | 1 | 2 | 0 | 31 (53%) |
| Google (40) | 8 (20%) | 20 | 1 | 1 | 0 | 24 (60%) |
| Sina (83) | 58 (70%) | 15 | 7 | 13 | 4 | 78 (94%) |
| Summary (182) | 75 (41%) | 62 | 9 | 16 | 4 | 133 (73%) |

– [a]: One RP may be susceptible to multiple vulnerabilities, *e.g.*, the profile attack and token hijacking, at the same time.

These observations show that OAuth2.0 is still the most popular SSO protocol. Unfortunately, the implementations of OAuth2.0 (including the ignore *id_token* case of OIDC) are also more susceptible: 75% OAuth2.0-enabled RPs have at least one vulnerability whereas 29% OIDC-supported RPs are vulnerable. Different OAuth security vulnerabilities are summarized in Table 2. Below we first discuss the implication of the profile attack and then demonstrate the pervasiveness of existing vulnerabilities.

### 6.1 The Implication of the Profile Attack

As illustrated in Table 2, 41% of the RP under test are found to be vulnerable to the newly discovered profile attack. Table 3 depicts a partial list of the vulnerable mobile apps we have identified so far. Notice that the total number of downloads for this incomplete list of popular but vulnerable apps already exceeds 2.4 billion. Based on the SSO-user-adoption-rate of 51% according to the recent survey by Janrain [7], we conservatively estimate that more than one billion of different types of mobile app accounts are susceptible to the profile attack as of this writing.

After signing into the victim's vulnerable RP app account using our exploit, the attacker will have, in many cases, full access to the victim's sensitive and private information which is hosted by the vulnerable RP server. Just for the vulnerable apps listed in Table 3 alone, a massive amount of extremely sensitive personal information is wide-open for grab: this includes detailed travel itineraries, personal/intimate communication archives, family/private photos, personal finance records, as well as the viewing or shopping history of the victim. For some RPs, the online-currency/service credits associated with the victim's account are also at the disposal of the attacker.

Although our current attack is demonstrated over the Android platform, the exploit itself is platform-agnostic: any iOS or Android user of the vulnerable mobile app is affected as long as he/she has used the OAuth SSO service with the app before. As a proof of concept, we have conducted the same attack on two vulnerable iOS apps.

### 6.2 Re-Discover Known Vulnerabilities

Table 2 shows that our testing also rediscovers different types of existing security issues.

**Table 3.** A Partial list of vulnerable apps and the sensitive information exposed

| Type of apps | IdP supported | # of app downloads (in Millions) | Type of private/sensitive information exposed | Feasible transactions by the attacker |
|---|---|---|---|---|
| Travel plan app | Sina | >270 | Travel itineraries | - |
| Hotel booking app | Facebook, Google | >5 | Lodging history | Pay for room bookings |
| Private chat app | Sina | >10 | Private message/album | Send forged messages |
| Dating app | Google, Sina | >5 | Dating history, preferences | Purchase gifts |
| Finance app1 | Sina | >25 | Personal income/expenses | - |
| Finance app2 | Sina | >50 | Stock list of interest | - |
| Call app | Facebook | >10 | Contact list and call history | Call for free |
| Live video app | Sina | >15 | The host the victim likes | Purchase gifts |
| Download app | Sina | >60 | Download history | Enjoy VIP speed |
| Shopping apps | Facebook, Google | >100 | Shopping history | - |
| Browser | Sina | >40 | Browsing history | - |
| Video apps | Sina | >700 | Video watching history | Purchase videos |
| Music apps | Google, Sina | >800 | Playlist | Purchase sound-tracks |
| News apps | Sina | >350 | News-reading history | - |

1. Token hijacking [1]: At Step 6–Step 7 of Fig. 1, the RP server must check that the received access token is granted to the same RP. Unfortunately, 34% RPs fail to do so, which enables an adversary to sign onto a victim's benign RP account by leveraging an access token issued to a malicious RP.
2. Improper user agent [12]: In addition to the infeasibility to identify the RP app, the WebView, as a custom webkit browser embedded in the app, is also untrustworthy to be a OAuth user-agent. Since the WebView is under the control of the RP app, a malicious RP app is capable of stealing any information submitted by the user in the WebView (*e.g.*, the user's IdP password) and modifying authorization information displayed by the WebView. Unfortunately, most RP apps support WebView, and worse still, 5% RP apps only support this problematic scheme.
3. Access token disclosure [27]: An access token should be transmitted securely. Thanks to the higher adoption rate of TLS, only 9% mobile RPs disclose their access token whereas 32% 3rd-party websites made this mistake [27].
4. App secret disclosure [29]: The confidential app secret should only be shared between the RP server and IdP server. There are 15 mobile RP apps deploy the authorization code flow rather than the implicit flow. Unfortunately, 27% of these apps inadvertently pass this secret through the user device. This allows an attacker to make operations on behalf of the RP, *e.g.*, changing the RP's security setting.

## 7 Plausible Root Causes

Surprised by the prevalence of various incorrect implementations, we also try to analyze the underlying reason by examining the SDKs and the OAuth APIs provided by IdPs.

### 7.1 Unclear Developer Documentation

We found that many authentication-related security issues are caused by the lack of clear guidelines. Since OAuth2.0 (RFC 6749 [18]) was not designed for mobile app authentication, various IdPs have developed different home-brewed extensions of OAuth2.0-based APIs and SDKs to support SSO for mobile apps. Unfortunately, the implicit security assumptions and operational requirements of such home-brewed adaptations are often not clearly documented or well-understood by RP developers. For example, when Sina returns the user profile to the RP app at Step 4 of Fig. 1, the only purpose is to allow the RP app to display the user info (*e.g.*, user name, avatar, etc.). Despite of this specific intention, Sina makes the following confusing claim[6] in its programming guide [5]:

> For the convenience of app developers, the returned user information can avoid calling the user-profile API, *i.e.*, *users/show*.

As pointed out in Sect. 3, a server-to-server call is inevitable for the standard OAuth2.0 protocol to verify the untrusted identity proof. However, the above claim can mislead RP servers not to make the user-profile API call to the IdP server at Step 8 and Step 9. Instead, the RP server may directly use the returned user information from its client-side app as the identity proof and thus be vulnerable to the profile attack.

For another example, Google assumes the RP developers would adopt the OIDC protocol, and thus only shows how an RP app can authenticate with its backend server using the *id_token*. Unfortunately, the majority of apps use the OAuth2.0 protocol instead. For these apps, Google does not define the interactions between the RP app and RP server. Therefore, the RP developers without adequate security expertise have to implement the error-prone authentication services by themselves.

After reporting the security issues to the three IdPs, all of them recognize the need to improve their documentation by explicitly pointing out the implicit security requirements. For example, Sina now updates its claim [5] as follows:

> The third-party apps should not use *uid* to identify the logged-in user. Note that the access token is the only valid identity proof.

[6] Since Sina developers are not native English speakers, the statements are translated by the author.

### 7.2 Poor API Design of Sina

Since more Sina apps are vulnerable to the profile attack, we further examine its SDK and API design. We find that the poor API design of Sina may compound this problem. As an open social network, Sina by design allows any RP in possession of a valid access token, no matter whom the access token is issued to, to retrieve *any* user's basic profile via the *users/show* API: https://api.weibo.com/2/users/show.json?access_token=x&uid=x. Even if the RP server does not trust the user id and would like to use the access token to identify the user, it may incorrectly use the above API. In this case, Sina server would return the victim's user profile only based on the value of *uid* in the URL. Without realizing the exact semantic of the returned information, the RP developer may incorrectly interpret that the returned user profile is bound to the access token, and thus, log the user in.

In fact, Sina also provides a correct API (*i.e.*, *account/get_uid*) to get the user-id of the one, whom the access token is bound to. But Sina never specifies which API to use[7]. Due to the richer information provided by the inappropriate *users/show* API[8], we believe RP developers prefer the incorrect API. By contrast, Facebook and Google use the *people/me* API, which is more self-explained, and more importantly, is typically handled by IdP SDKs. For example, the PHP SDK of Facebook hard-encodes the user id *me*, and as such the *uid* fed by the attacker would be ignored by the SDK.

## 8 Defense

The community has proposed the following Current Best Practices (CBPs):

1. IdPs should provide clearer, and more security-focused developer guidelines.
2. The RP server should not trust any information even if it is signed by its own app. Trust should be anchored on the IdP server directly.
3. To implement OAuth2.0 for mobile apps, RP developers should use the authorization code flow instead of implicit flow by strictly following [14].
4. Use OIDC for authentication whenever possible.

If these CBPs can be correctly followed, then most vulnerabilities will not exist. Unfortunately, most RP developers never adhere these CBPs at all. Towards this end, the crux of the defense is to enforce the defined security checks, which cannot simply rely on the RP developers, but instead must be strongly enforced by the IdPs, since the latter has adequate security expertise. Therefore, we develop a general and foolproof solution, from the perspective of the IdP, to help the RP developers to automatically handle the error-prone SSO services. Our solution should achieve three goals:

[7] Worse still, Sina seems to recommend the incorrect one in their unclear developer documentation.

[8] The other API only returns the *uid* of the user.

1. All the identified vulnerabilities (including existing ones) can be prevented.
2. To ease the transition, the code changes by the RP developers should be as few as possible.
3. The solution should be backward compatible in a sense that the RP server can still support those users who use the older version of the RP app. After all, it is difficult for every user to upgrade his/her RP app.

With these goals in mind, we first review the architecture of existing SSO systems. As shown in Fig. 6, both the RP server and RP app can be split into two modules: the SDKs provided by the IdP serve the interactions with the IdP app/server and the upper layer codes implemented by the RP developer manage its own business logic. Currently, the identity proof is also (incorrectly) handled by the upper layer code. To prevent from SSO errors, we migrate such functionality to the lower layer SDKs, with the belief that the SDKs provided by IdPs can correctly process this security-critical identity proof. Specifically, upon the reception of the identity proof at Step 4 of Fig. 1, the client-side SDK can send it to the server-side SDK. The latter then performs the real authentication task by utilizing this identity proof. Below we will discuss more details.

### 8.1 Defense on the Client-Side SDK

As shown in Fig. 4, when the RP apps only return user information, there is no way for the RP servers to correctly identify the user. Therefore, the enforced version of the client-side SDK should automatically send required information including the access token (or the id_token for OIDC) to its backend server. With due consideration to the ease of transition, below we first discuss the design of the existing client-side SDK.

**API Design of the Existing Client-Side SDK.** We only take Sina, [9] one OAuth2.0 IdP, as the example. When authorization succeeds, the client-side SDK (used by the RP app) will utilize Android API *onActivityResult* to receive an access token along with a user id from Sina app. Before forwarding this result to the upper layer, *authorizeCallBack* API is first invoked to check whether the access token is in the correct format.

```
protected void onActivityResult(...){
  mSsoHandler.authorizeCallBack(requestCode,resultCode,result);
}
```

**Initial Attempt Using Cookie-Based Scheme.** Without affecting the upper layer behavior, we manage to authenticate the user in the SDK. Referring to the scheme in websites, one natural attempt, as shown in Fig. 6, is to use *cookie*.

[9] Our solution is applicable to OIDC protocol or other IdPs since they follow the same flavor.

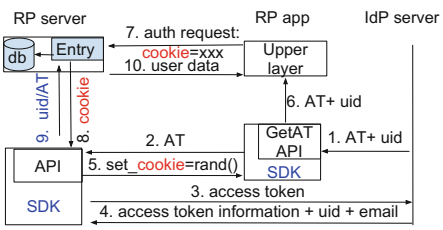



**Fig. 6.** The cookie-based defense

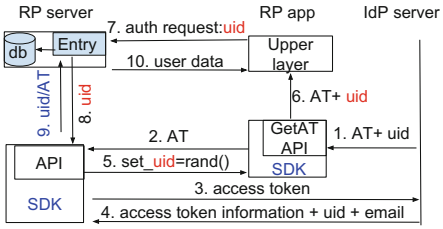



**Fig. 7.** The refined defense

1. After the client-side SDK (*i.e.*, *authorizeCallBack*) checks the format of the access token, instead of forwarding it to the upper layer, the client-side SDK first delivers the access token to the server-side SDK.
2. The server-side SDK exactly follows Step 3–Step 4 in Fig. 6 (corresponding to Step 6–Step 9 in Fig. 1) to identify the user via the access token.
3. If the verification succeeds, the server-side SDK then sets a *cookie* to the client-side app with a random nonce at Step 5 of Fig. 6. Only then would the client-side SDK forward the access token and the user profile to the upper layer code.
4. From the view of the upper layer, everything remains the same. Thus it can follow its original logics to interact with the RP server. The only difference is that a cookie would be automatically attached to the authentication request at Step 7 of Fig. 6.
5. Regardless of other information sent by the RP app, the server-side SDK only relies on the *cookie* to identify the user.

Unfortunately, this cookie-based solution is not applicable to every RP app: Unlike websites where a central browser can help to manage cookies, the Android app developers need to manage the cookies by themselves, for example using the *CookieManager*. Therefore, the *cookie* set by the underlying SDK may not be automatically used by the upper layer, if the latter adopts a customized *CookieStore*.

**Refined Defense.** Nevertheless, the cookie-based scheme still provides great insights. Note that the cookie is used to bind the requests of Step 2 and Step 7 in Fig. 6. Towards this end, the client-side SDK and its upper layer code must share some information like the *cookie*. Furthermore, the shared information must be a *secret*. Otherwise, an adversary can easily guess/compute this information and pretend to be anyone else.

Given these requirements, we revise the cookie-based solution. Referring to the practice of Facebook which issues private user-id on a per-app basis, we will randomly generate a one-time user id, instead of the *cookie*, as the secret to bind these two requests. More specifically, if authentication succeeds, the server-side SDK would return a randomly generated *uid* to its client-side SDK at Step 5 of Fig. 7. This *uid* plays the same role as *cookie*. Note that this *uid* can only be

used for once so that an adversary cannot guess/compute its value. The old *uid* will be deleted once expired or used.

### 8.2 Defense on the Server-Side SDK

At Step 2 of Fig. 7, the server-side SDK can follow Step 3 and Step 4 in Fig. 7 (*i.e.*, correspond to Step 6–Step 9 in Fig. 1) to identify the user. Once authentication succeeds, the server-side SDK randomly generates a one-time *uid* with a specific prefix and maintains the mapping of $< uid, uid_{real} >$. At Step 7 of Fig. 7, the server-side upper layer should forward the authentication request to its SDK. Our newly added function in the SDK can then handle this request according to its content.

1. If the request only contains the user id, *i.e.*, *uid*, we check whether this *uid* starts with a specific prefix. If so, we check the mapping of $< uid, uid_{real} >$ and then get the user information $uid_{real}$. Otherwise, just abort the request.
2. If the request only contains the access token, we follow Step 3 and Step 4 in Fig. 7 to identify the user.
3. If the request contains the access token and user id, we first follow Case (1) to process the user id. If it fails, we then follow Case (2) to process the access token.

**Remark.** The correctness of our defense is based on two assumptions. Firstly, a correct implementation of OAuth, as shown in Figs. 1 and 2, is automatically immune to all the existing attacks. In fact, a formal proof is presented by [34]. More precisely, this work utilizes the model checking method to analyze the implementation-level protocol of Facebook and can only discover unauthorized storage access if the malicious app has root privileges. However, such a strong threat model is not considered in this paper. Secondly, the IdPs (*i.e.*, SDKs) with enough security expertise can accurately implement the protocol (as opposed to the RP developers who often make mistakes). Therefore, the crux of our solution is to enforce the IdP developers to correctly implement the OAuth protocol for the RP developers.

Although the design of the defense seems complicated, we only add two more requests (*i.e.*, Step 2 and Step 5 at Fig. 7) into the current system. Note that the other steps (*e.g.*, Step 6, Step 8, *etc.*) already reside in the existing systems. For the ease of presentation, we intentionally hide these detailed (SDK-level) steps when discussing the protocol flow in Fig. 1. Note also that the proposed remedies does not affect the authorization code flow, although we can use the same idea to improve its security.

### 8.3 Implementation and Evaluation of the Proposed Defense

To demonstrate the feasibility of the proposed remedies, we implement the solution on a sample app provided by Sina. Like examples from the other IdPs, the sample app is built on top of the Android SDK. Note that this SDK is in the

form of executable Jar file. To modify it, we thereby need to decompile the Jar file into Java code. While there are off-the-shelf Java decompiler tools like CFR or JAD, the extracted source code is not well structured and contains various errors, which requires non-trivial manual resolve.

We then follow the common practices to build a backend server on top of the official PHP SDK. The only change in the server-side SDK is to add a new function which performs the real authentication task. Meanwhile, we only add 5 lines of code in the server-side upper layer, which demonstrates the least programming efforts from the RP developers.

**Table 4.** The average running time under three different settings

| Two types of app | User info (in ms) | Access token (in ms) | Access token & user info (in ms) |
|---|---|---|---|
| Vulnerable app | 4490 | 7604 | 5092 |
| Fixed app | 6414 | 9667 | 6009 |

**Evaluation.** To demonstrate the effectiveness, we have launched different attacks listed in Table 2 on the sample app. It turns out that our solution can prevent from (or alert for insecure transmission, *e.g.*, token disclosure) all these attacks. For example, the profile vulnerability becomes impossible since an adversary cannot guess the randomly-generated one-time *uid*. Take token hijacking as another example. Since our defense exactly follows Fig. 1, our server-side SDK will verify whether this access token is issued to itself or not at Step 3 of Fig. 7 (corresponds to Step 6 of Fig. 1). Thus an access token of another RP will not be accepted.

To show the efficiency of our solution, we measure the running time of a complete SSO process. Specifically, we enumerate all three possible cases of the authentication request, as mentioned in Sect. 8.2. For each case, we operate the sample app to complete the OAuth2.0 process for 20 times under the same phone and network environment. The average time is presented in Table 4. It shows that the fixed app has only a small impact on the user performance.

**The State-of-the-Art OAuth Defense.** The Current Best Practices (CBP) [14] suggests the usage of authorization code flow to prevent from many existing problems. However, a correct implementation of these CBPs is still challenging for the RP developers (as is the case for the revised implicit flow). Furthermore, it requires lots of efforts to migrate from implicit flow (the current *de fact* standard) to authorization code flow. Xing *et al.* [32] develop invariants of HTTP parameters to protect third-party web service integrations like OAuth2.0. However, it cannot discover many attacks such as the profile attack. Another defense [11] uses program verifier to check whether the sequence of method calls satisfies the defined predicate. Despite its power, this method requires significant

programming efforts for *all* involved parties. Compared to the state of the art, our defense not only can prevent from all the existing attacks, but also requires the minimal programming efforts from the RP developers.

## 9 Related Work

**Security Analyses from the Protocol Perspective.** IETF has presented comprehensive security considerations and threat models in RFC6749 [18] and RFC6819 [22] for OAuth2.0 protocol from the initial design. Additionally, the authorization code flow has proven to be secure cryptographically [10] as long as the TLS is properly used. Hu *et al.* [20] present the App Impersonation attack. These works mainly prove the authorization security for the web from the protocol design standpoint. However, we consider whether the protocol is securely implemented on mobile platforms for authentication.

**Formal Security Proof for OAuth.** The model checking method is extensively used to analyze the protocol specification [18] by numerous works including [8,9,16,24], just to name a few. Researchers also attempt to use the same method to reason about the protocol implementations by modeling the complex runtime platform, *e.g.*, browser [16,17]. All of these works assume a correct implementation of the protocol call-flow. However, we show that protocols are often implemented incorrectly.

**Real-World Study on the Web-Based SSO Systems.** More efforts have been contributed to the vulnerability detection of the website-based real-world systems, including [13,23,27,30]. Another research direction is to conduct large-scale testing on the web SSO systems, including SSOScan [26,35], OAuthTester [21,33]. Nevertheless, all the works mentioned so far only study the OAuth/OIDC specification/implementations on the website. In contrast, we focus on the mobile platforms.

**Analyses of Mobile OAuth SSO Systems.** There are relatively few security analyses on OAuth under mobile environment. Chen *et al.* [12] shows how real-world OAuth systems can fall into the common pitfalls when leveraging the operating-system-provided components (*e.g.*, Intent, WebView, *etc.*). Ye *et al.* [34] utilize the model checking method to evaluate the OIDC-like protocol implemented by Facebook on Android platform. Summarizing these works, Wang *et al.* [19,29] collect the statistics of these known vulnerabilities. Prior works mainly focus on how the classic vulnerabilities (or features) of mobile systems can be leveraged to compromise SSO systems. In contrast, we analyze how the modified protocol call-flow itself can be incorrectly implemented.

## 10 Conclusion

In this paper, we report a field-study of mobile OAuth2.0-based SSO systems. We perform an in-depth static code analysis on three first-tier IdP apps and their

official SDKs as well as dynamic testing on Top-600 Android apps in China and US. Besides the discovery of three previously unknown security flaws, we also demonstrate the prevalence of different types of existing vulnerabilities. The pervasiveness of these loopholes motivates us to design and implement a foolproof defense for those susceptible RPs with the aim of minimizing the programming efforts of RP developers. Our discoveries show that it is urgent for the various parties to re-examine their OAuth implementations and apply the fixes accordingly.

**Acknowledgements.** This project is supported in part by the Innovation and Technology Commission of Hong Kong (project no. ITS/216/15) and NSFC Grant (No. 61572415).

## References

1. Access token hijacking. https://developers.facebook.com/docs/facebook-login/security#tokenhijacking
2. Android account manager. http://developer.android.com/reference/android/accounts/AccountManager.html
3. Man in the middle proxy. https://mitmproxy.org/
4. One major Chinese App store. http://sj.qq.com/myapp/category.htm
5. Sina access token API. http://open.weibo.com/wiki/OAuth2/access_token
6. SSL unpinning. https://github.com/ac-pm/SSLUnpinning_Xposed
7. Social login continues strong adoption (2014). http://janrain.com/blog/social-login-continues-strong-adoption/
8. Bai, G., Lei, J., Meng, G., Venkatraman, S.S., Saxena, P., Sun, J., Liu, Y., Dong, J.S.: AUTHSCAN: automatic extraction of web authentication protocols from implementations. In: NDSS (2013)
9. Bansal, C., Bhargavan, K., Maffeis, S.: Discovering concrete attacks on website authorization by formal analysis. In: IEEE CSF (2012)
10. Chari, S., Jutla, C.S., Roy, A.: Universally composable security analysis of OAuth v2.0. Cryptology ePrint Archive, Report 2011/526 (2011)
11. Chen, E.Y., Chen, S., Qadeer, S., Wang, R.: Securing multiparty online services via certification of symbolic transactions. In: IEEE S&P (2015)
12. Chen, E.Y., Pei, Y., Chen, S., Tian, Y., Kotcher, R., Tague, P.: OAuth demystified for mobile application developers. In: ACM CCS (2014)
13. Mainka, C., Vladislav Mladenov, J.S., Wich, T.: SoK: Single Sign-On security- an evaluation of OpenID Connect. In: IEEE EuroS&P (2017)
14. Denniss, W., Bradley, J.: OAuth 2.0 for native apps (2016)
15. Elenkov, N.: Android Security Internals: An In-Depth Guide to Android's Security Architecture. No Starch Press, San Francisco (2014)
16. Fett, D., Küsters, R., Schmitz, G.: An expressive model for the web infrastructure: definition and application to the browser ID SSO system. In: IEEE S&P (2014)
17. Fett, D., Küsters, R., Schmitz, G.: A comprehensive formal security analysis of OAuth 2.0. In: ACM CCS (2016)
18. Hardt, D.: The OAuth 2.0 authorization framework (2012)
19. Homakov, E.: The Achilles Heel of OAuth or Why Facebook Adds Special Fragment (2013)

20. Hu, P., Yang, R., Li, Y., Lau, W.C.: Application impersonation: problems of OAuth and API design in online social networks. In: ACM Conference on Online Social Networks, COSN (2014)
21. Li, W., Mitchell, C.J.: Analysing the security of Google's implementation of OpenID Connect. In: SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment, DIMVA (2016)
22. Lodderstedt, T., McGloin, M., Hunt, P.: OAuth 2.0 threat model and security considerations (2013)
23. Mladenov, V., Mainka, C., Krautwald, J., Feldmann, F., Schwenk, J.: On the security of modern Single Sign-On protocols: OpenID Connect 1.0. CoRR abs/1508.04324 (2015)
24. Pai, S., Sharma, Y., Kumar, S., Pai, R.M., Singh, S.: Formal verification of OAuth 2.0 using Alloy framework. In: IEEE International Conference on Communication Systems and Network Technologies, CSNT (2011)
25. Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., Mortimore, C.: OpenID Connect core 1.0. The OpenID Foundation (2014)
26. Shernan, E., Carter, H., Tian, D., Traynor, P., Butler, K.: More guidelines than rules: CSRF vulnerabilities from noncompliant OAuth 2.0 implementations. In: Almgren, M., Gulisano, V., Maggi, F. (eds.) DIMVA 2015. LNCS, vol. 9148, pp. 239–260. Springer, Cham (2015). doi:10.1007/978-3-319-20550-2_13
27. Sun, S., Beznosov, K.: The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In: ACM CCS (2012)
28. Wang, H., Zhang, Y., Li, J., Gu, D.: The achilles heel of OAuth: a multi-platform study of OAuth-based authentication. In: ACM ACSAC (2016)
29. Wang, H., Zhang, Y., Li, J., Liu, H., Yang, W., Li, B., Gu, D.: Vulnerability assessment of OAuth implementations in Android applications. In: ACM ACSAC (2015)
30. Wang, R., Chen, S., Wang, X.: Signing me onto your accounts through Facebook and Google: a traffic-guided security study of commercially deployed Single-Sign-On web services. In: IEEE S&P (2012)
31. Wang, R., Xing, L., Wang, X., Chen, S.: Unauthorized origin crossing on mobile platforms: threats and mitigation. In: ACM CCS (2013)
32. Xing, L., Chen, Y., Wang, X., Chen, S.: InteGuard: toward automatic protection of third-party web service integrations. In: NDSS (2013)
33. Yang, R., Lee, G., Lau, W.C., Zhang, K., Hu, P.: Model-based security testing: an empirical study on OAuth 2.0 implementations. In: ACM ASIACCS (2016)
34. Ye, Q., Bai, G., Wang, K., Dong, J.S.: Formal analysis of a Single Sign-On protocol implementation for Android. In: International Conference on Engineering of Complex Computer Systems, ICECCS (2015)
35. Zhou, Y., Evans, D.: SSOScan: automated testing of web applications for Single Sign-On vulnerabilities. In: USENIX (2014)