Signing into One Billion Mobile App Accounts Effortlessly with OAuth2.0

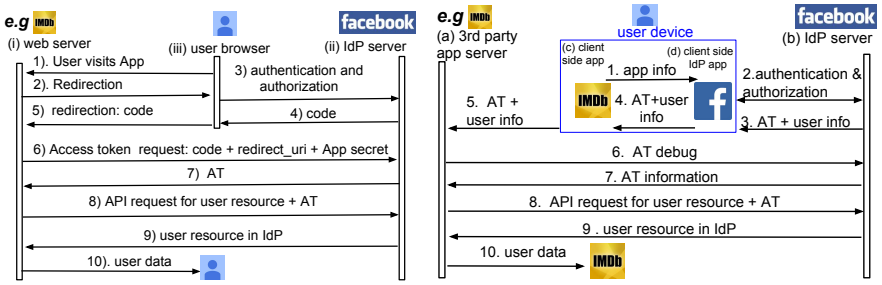
Ronghai Yang Wing Cheong Lau Tianyu Liu
The Chinese University of Hong Kong

Abstract

OAuth2.0 protocol has been widely adopted by mainstream Identity Providers (IdPs) to support Single-Sign-On service. Since this protocol was originally designed to serve the authorization need for 3rd party websites, different pitfalls have been uncovered when adapting OAuth to support mobile app authentication. To the best of our knowledge, all the attacks discovered so far, including BlackHat USA'16 [3], CCS'14 [2] and ACSAC'15 [5], require to interact with the victim, for example via malicious apps or network eavesdropping, *etc*. On the contrary, we have discovered a new type of widespread but incorrect usages of OAuth by 3rd party mobile app developers, which can be exploited remotely and solely by the attacker to sign into a victim's mobile app account without any involvement/ awareness of the victim. To demonstrate the prevalence and severe impact of this vulnerability, we have developed an exploit to examine the implementations of 600 top-ranked US and Chinese Android Apps which use the OAuth2.0-based authentication service provided by three top-tier IdPs, namely Facebook, Google or Sina. Our empirical results are alarming: on average, 41.21% of these apps are vulnerable to this new attack. We have reported our findings to the affected IdPs, and received their acknowledgements/ rewards in various ways.

1 Introduction

Riding on the widespread user adoption of OAuth2.0-based Single-Sign-On (SSO) services for 3rd party websites, many major Identity Providers (IdPs) such as Facebook, Google and Sina, have recently adapted the OAuth2.0 protocol to support SSO for 3rd-party Mobile Apps on their social-media platform. However, due to the differences in the end-to-end system setup and operating environment for mobile app SSO services, the original OAuth2.0 protocol becomes under-specified. In particular, the OAuth2.0 standard does not cover or define the critical security requirements and protocol details to govern the interactions between the 3rd-party (client-side) mobile app and its corresponding backend server during the SSO process. As a result, various IdPs have developed different home-brewed extensions of OAuth2.0-based Application Programming Interface (API) to support SSO of 3rd-party mobile apps in their own platform. Unfortunately, the implicit security assumptions and operational requirements of such home-brewed adaptations/ APIs are often not clearly documented or well-understood



(a) The protocol flow of web-based OAuth implementations

(b) The protocol flow of mobile OAuth implementations

Figure 1: Compare the OAuth protocol flow between the website and the mobile platform

by the 3rd-party mobile app developers. Worse still, there is also a lack of security-focused SSO-API-usage-guidelines for the 3rd-party app developers.

These problems have led us to discover a new type of widespread, insecure implementations of OAuth2.0-based SSO services among a large number of popular 3rd-party mobile apps which use various top-tier IdPs. The root cause of this vulnerability is a common, but misplaced trust in the authenticating information received by the 3rd party app's backend server from its own client-side mobile app, which in turn, relies on potentially tampered information obtained from the client-side mobile app of the IdP. Leveraging on this newly discovered vulnerability, we have developed a remote exploit which enables an attacker to effortlessly sign into a victim's mobile app account via OAuth2.0[1] without any need to trick or interact with the victim, for example via malicious apps or network eavesdropping, *etc*. While 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 OAuth2.0-based SSO service with the app before.

2 Background

There are four parties involved to support SSO for 3rd party mobile App, namely, (a) the backend server of the 3rd party mobile app, (b) the backend server of the IdP, (c) the 3rd party (client-side) mobile app and (d) the client-side mobile app of the IdP. The ultimate goal of OAuth is for the backend server of the IdP to issue an identity proof, *i.e.*, the access token, to the server of the 3rd party mobile app. Utilizing this access token, the 3rd party app server can then retrieve user information hosted by the IdP server so as to identify the user and log the user in.

2.1 The Protocol Flow of OAuth 2.0 on the Mobile Platform

Fig. 1 shows the call-flow when implementing the OAuth protocol on the website and the mobile platform. For simplicity, we first introduce the protocol flow for the mo-

[1]For the rest of the paper, we use OAuth to denote OAuth 2.0 if not specified otherwise.

bile OAuth implementations and then point out the differences in the web-based SSO services. Note that, neither the RFC nor the IdP provides a complete call-flow diagram for the 3rd party mobile app developers since OAuth was not designed for mobile apps. Nevertheless, there have been extensive efforts [2,3,5,6] on analyzing the OAuth security under the mobile environment and one believed-to-be-secure realization is as follows:

1. The user visits the third-party (client-side) mobile app and tries to log into the mobile app with the IdP. The third-party client-side app sends its app information (e.g., package name, signature and requested permissions, *etc*) to the client-side app of the IdP via a secure channel provided by the operation system of the mobile phone, *e.g.*, *intent* for Android.

2. By calling the low-level system APIs, the client-side IdP app can verify whether the app information of the 3rd-party app is correct or not. If so, the IdP client-side app then sends an authorization request to its backend server.

3. The IdP server compares information from the authorization request and that pre-registered by the 3rd party mobile app developers. If they are the same, the IdP server would issue an access token (AT) together with the optional user profile to the 3rd-party client-side mobile app via its own client-side app.

4. The client-side IdP app returns the access token (AT) to the 3rd-party client-side app via the secure channel.

5. The 3rd-party client-side app sends AT to its backend server.

6. The 3rd party backend server should call the security-focused SSO-API provided by the IdP to debug the access token.

7. After verifying the validity of the access token, the IdP server should respond the 3rd party app server with the authorization information including which app this access token is issued to.

8. Only if the authorization information is correct can the 3rd party app server retrieve the user data with the access token.

9. The IdP server returns the user data associated with the access token.

10. Leveraging the user data, the 3rd party app server can then identify the user and log the user in.

2.2 The OpenID Connect Protocol

Because 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(b). To better support authentication (*i.e.*, with less round-trips) using OAuth2.0, IdPs like Google and Facebook have developed the OpenID Connect (OIDC) protocol [4] and its variants. Specifically, IdPs need to digitally sign the user profile (*i.e.*,

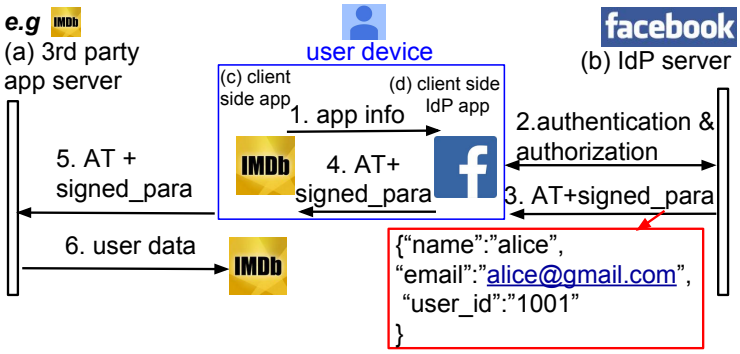
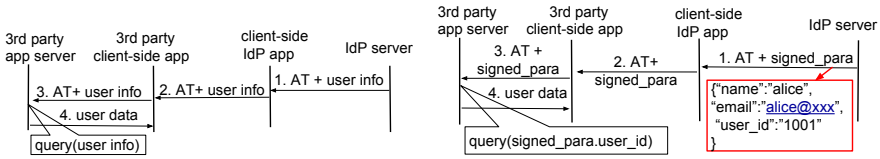


Figure 2: The protocol flow of OpenID Connect implemented on the mobile platform

signed_para). This signed user profile, as shown in Fig. 2, is then sent to the back-end server of the 3rd party mobile app along with the original access token. Since the signature cannot be tampered/ forged by an attacker, the 3rd-party app server can now directly identify the user via the signature. In other words, the 3rd-party app server can immediately extract the user profile from the signature without the high-latency API calls.

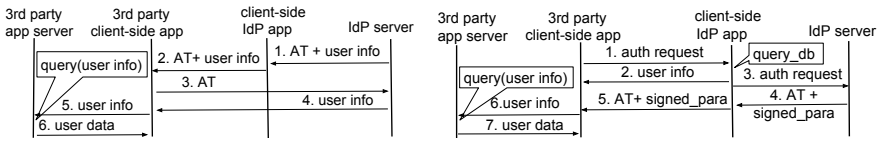
2.3 Key Differences in the Website-based OAuth Implementations

While the differences in the protocol flows between websites and mobile platforms are seemingly straightforward, such differences in fact have non-trivial security implications and have complicated the implementations on the mobile platforms. For example, there are three entities, namely, (i) the 3rd party web-server, (ii) the backend server of the IdP and (iii) the end-user's browser (which acts as the user-agent under OAuth terminology), involved in supporting OAuth2.0-based SSO for 3rd party websites, the support of SSO for 3rd party mobile apps involves four different entities as aforementioned. Firstly, both (c) and (d) are running on the end-user devices and subject to tampering. Secondly, the interactions as well as the associated security requirements between (a) and (c), as well as those between (c) and (d) are outside the scope of the OAuth2.0 protocol standard. Thirdly, with the presence of both (c) and (d) on the same end-user device, it may be tempting for the 3rd-party mobile app developers to conduct authentication exchange between (c) and (d) directly (as opposed to the direct verification between (a) and (b), just like the case of direct authentication exchange between (i) and (ii) as defined in the original OAuth2.0 standard). For another example, different from web-based service providers, mobile application developers are often urged by the IdPs to use a different authorization grant flow in OAuth which is tightly integrated with IdP-specific business logic, namely authorization code flow vs. implicit flow. Furthermore, the client side of a typical mobile application is responsible for more message exchange which, in contrast, is managed by the backend server in the case of 3rd party websites (and their corresponding web-based services).



(a) No verification of the access token (b) No verification of the signature

Figure 3: The 3rd party app server does not verify the identity proof of the user



(a) User information obtained via API call (b) User information obtained via Android account management system

Figure 4: The 3rd party client-side app returns incorrect identity proof to its server

3 Different Types of Insecure Implementations

Despite the major differences, no clear developer guides are provided by IdPs to demystify the possbile pitfalls for mobile OAuth implementations. As a result, 3rd party mobile app developers have made different types of common but widespread mistakes as follows:

1. As shown in Fig. 3(a), when IdP servers return user identity information (e.g., user id/email address) in addition to the original OAuth access token, many backend servers of 3rd-party apps simply (and incorrectly) login the user based on the received user-id WITHOUT verifying whether the received user-id is indeed bound to the issued OAuth access token.

2. Fig. 3(b) shows another case where Facebook and Google adopt the OpenID Connect protocol. In this case, IdPs need to digitally sign the user identity profile so that the backend server of a 3rd party app can directly authenticate the user by verifying this signature. However, some 3rd party apps never verify this signature and simply extract the user-id from the payload of the signature. This user-id is then treated as the identity proof without any authentication/ validation.

3. Regardless of the OAuth access token received from the IdP, some 3rd-party mobile apps directly retrieve the user information from the mobile device it is running on (*e.g.*, either via the API call provided by the IdP server illustrated in Fig. 4(a) or from the *Android account management* system which stores the user's Google accounts to support SSO service as shown in Fig. 4(b)). The client-side mobile app then only sends the user identifier to its backend server as the identity proof. Without the access token (for which the trust should be anchored), 3rd-party backend servers have no way to verify if the returned user identifier is actually bound to the OAuth access token.

4 Exploiting the Vulnerability

Given the various incorrect implementations by 3rd party app developers, an attacker therefore can log into a susceptible app as the victim by merely exploiting the victim's profile with the following steps:

1. As shown in Fig. 5, the attacker setups a ssl-enabled-MITM proxy, *e.g.*, mitmproxy, for her own mobile device to monitor and tamper the network traffic going into and leaving her device.

2. The attacker installs the vulnerable 3rd party app in her own mobile device.

3. The attacker signs into the vulnerable mobile app with OAuth using the attacker's own IdP login name and password.

4. During the OAuth message exchange triggered by Step 3, the attacker substitutes her own user id (user id in the IdP or email address) with the victim's one using the ssl-enabled-MITM proxy. The victim's user-id is either a publicly available information (available from the victim's public web page for the case of Google+ and Sina users) or easily guessable (in the case where the app use user-email-address as the user name). Although Facebook has started to issue private per-app user-id for each third-party app since May 2014, for backward compatibility reasons, to-date, Facebook still uses the public user-id to identify early adopters of a 3rd-party app. As such, a user of the latest version of a vulnerable app is still susceptible to our attack as long as he/ she has signed into the app via OAuth before May 2014.

5. Since the third-party backend server directly uses the user's identity proof returned from its client-side app to identify the app user WITHOUT further validation, the attacker can therefore successfully sign into the app as the victim and in most cases have full access to the victim's sensitive information hosted by the third-party app's backend server.

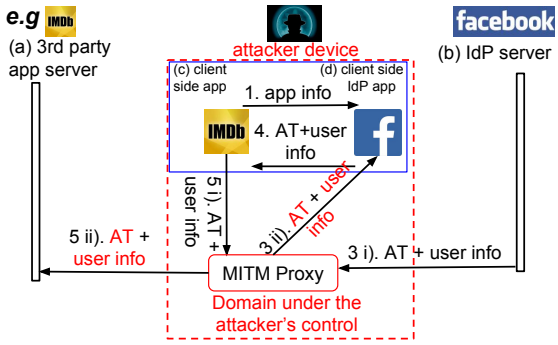


Figure 5: The platform to exploit the discovered vulnerability

Besides being protected by SSL/HTTPS, the message exchanges between the client-side of a 3rd-party mobile app and its backend server are often further encrypted or signed by the 3rd-party app client/server. Therefore, it is usually easier to tamper the user-id information returned by the IdP server to the IdP client-side app running on the attacker's mobile device.

In the case where the IdP client-side app, *e.g.*, the one by Facebook, applies the certificate pinning, the message sent by the IdP server to its client-side app will not be accepted by the latter if it has been tampered by the attacker's MITM proxy. As a workaround, the attacker can simply uninstall the IdP client-side app so that the IdP SDK (widely used by 3rd-party mobile apps for OAuth2.0) 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 specific IdP. As a result, the attacker's ssl-enabled-MITM proxy can now tamper the user-id info sent by the IdP server to the client-side of the 3rd-party mobile app.

There are additional cases where the IdPs choose not to support webview-based OAuth-authentication. For such IdPs, the attacker can either use off-the-shelf tools such as SSLUnpinning (if they use the native Android framework to implement the certificate pinning) or reverse engineer the IdP client-side app to manually remove the certificate pinning (if they use cutomized methods). To demonstrate the feasibility of this approach, we have successfully implemented a proof-of-concept hack on the Facebook app to manually disable its certificate pinning function so that we can feed it with incorrect user-id info via the ssl-enabled-MITM proxy.

5 Empirical Results

We have studied the OAuth2.0-based APIs provided by three top-tier IdPs, namely, Sina, Facebook and Google, which support SSO services for many 3rd party mobile apps worldwide. The number of registered users in these IdPs ranges from more than 800 millions to over 2.5 billions as depicted in Table 1. Since there are more Chinese moible apps supporting SSO services, we thereby select Top-200 mobile apps for Sina while select Top-400 mobile apps for Google and Facebook. We then identify those apps which use the OAuth2.0-based authentication service provided by one or more of the 3 IdPs mentioned above. We finally test these apps against our OAuth2.0-based exploit. Our findings are alarming: on average, 41.21% of the mobile apps under test are found to be vulnerable to the new attack. Table 2 depicts a partial list of the vulnerable mobile apps we have identified so far: this incomplete list already includes two Top-5

Table 1: Statistics for the Prevalence of Vulnerable Mobile Apps

IdPs	Alexa Rank	# of Users (in Millions)	# of Apps which use the SSO service provided by the IdP	# of Vulnerable Apps
Sina	Top 20	>800	83	58 (69.88%)
Facebook	Top 10	>1,500	59	9 (15.25%)
Google	Top 10	>2,500	40	8 (20%)

Table 2: A Partial List of Vulnerable Mobile Apps and the Sensitive Information They Exposed

Type of 3rd-party 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	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, Sina	>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	-

travel planning mobile apps, one popular hotel-booking app, a top private-chat app designed for couples/ partners, a Top-5 dating app, two top-ranked personal finance apps, as well as other popular apps for video-streaming or online-shopping, just to name a few. 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 [1], we conservatively estimate that more than one billion of different types of mobile app accounts are susceptible to our newly discovered attack as of this writing.

After signing into the victim's vulnerable mobile 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 backend server(s) of the vulnerable mobile app. Just for the vulnerable apps listed in Table 2 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 victims. For some of these mobile applications, the online-currency/ service credits associated with the victim's account are also at the disposal of the attacker.

6 Recommended Remedies

Our discovery shows that it is urgent for the various parties involved to take the following preventive/ remedial actions when implementing or using OAuth2.0-based SSO services:

1. IdPs should provide 3rd-party application developers clearer, and more security-focused usage guidelines for their OAuth2.0-based SSO APIs.

2. The 3rd party backend server of a mobile app should not trust any information

even if it is signed by its own client-side mobile app or by the client-side mobile app of the IdP. Trust should be anchored on the IdP server directly.

3. Instead of relying on a global user identifier for 3rd-party app authentication/ authorization, IdPs should issue private user identifier on a per-mobile-app basis. In fact, such a practice has been adopted by Facebook since May 2014. However, Facebook still insists on the global user identifier if the user started using the mobile app before May 2014. As such, the attack is still applicable to the early users of a vulnerable app.

4. IdPs should conduct or insist on more thorough security testing of 3rd party mobile apps, especially with respect to their implementation of Single-Sign-On services via OAuth2.0 or other similar protocols such as the OpenID Connect (OIDC) protocol.

7 Related Work

Despite the wide deployment, there are relatively few security analyses on the mobile OAuth implementations. Chen *et al.* [2, 3] show that most 3rd party mobile app developers do not debug the access token and a malicious app may be able to impersonate a benign one if the Android intent is not properly used. Wang *et al.* [5] systematically summarize the known vulnerabilities for 15 mainstream IdPs. Compared to the state-of-the-art, our attack is new: all the existing attacks against mobile OAuth implementations require an attacker to obtain a valid access token of the victim by interacting with the victim via malicious apps or non-HTTPS channels, *etc*. On the contrary, our attack can be conducted remotely and solely by the attacker without any involvement of the victim. As a result, the impact of our newly discovered vulnerability and its exploit are considerably wider and deeper than those proposed by other prior works.

8 Conclusion

In this paper, we have identified a previously unknown vulnerability which can be exploited remotely by the attacker to hijact the victim's mobile app account without any involvement with the victim. We have examined the implementations of Top-200 US and Chinese Android Apps which use the OAuth2.0-based authentication service provided by three top-tier IdPs, and demonstrated to what extent these popular apps are vulnerable to this new vulnerability. Our discovery shows that it is urgent for the various parties to re-examine their SSO implementations and take the suggested remedial actions accordingly.

9 Responsible Disclosure

In April 2016, we reported our findings to all the affected IdPs under study. All of them 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 3rd

party app 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 for 3rd party developers 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 app developers aware of this problem.

Acknowledgments

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

References

[1] "Social login continues strong adoption," 2014. [Online]. Available: http://janrain.com/blog/social-login-continues-strong-adoption/

[2] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "OAuth demystified for mobile application developers," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014.

[3] C. Eric, P. Tague, R. Kotcher, S. Chen, Y. Tian, and Y. Pei, "1000 ways to die in mobile OAuth," in *BlackHat USA*, 2016.

[4] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore, "OpenID Connect core 1.0," *The OpenID Foundation*, p. S3, 2014.

[5] H. Wang, Y. Zhang, J. Li, H. Liu, W. Yang, B. Li, and D. Gu, "Vulnerability assessment of OAuth implementations in Android applications," in *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 2015.

[6] Q. Ye, G. Bai, K. Wang, and J. S. Dong, "Formal analysis of a Single Sign-On protocol implementation for Android," in *20th International Conference on Engineering of Complex Computer Systems, ICECCS 2015*, 2015.