Vulnerability Assessment of OAuth Implementations in Android Applications

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

ABSTRACT

Enforcing security on various implementations of OAuth in Android apps should consider a wide range of issues comprehensively. OAuth implementations in Android apps differ from the recommended specification due to the provider and platform factors, and the varied implementations often become vulnerable. Current vulnerability assessments on these OAuth implementations are ad hoc and lack a systematic manner. As a result, insecure OAuth implementations are still widely used and the situation is far from optimistic in many mobile app ecosystems.

To address this problem, we propose a systematic vulnerability assessment framework for OAuth implementations on Android platform. Different from traditional OAuth security analyses that are experiential with a restrictive three-party model, our proposed framework utilizes an systematic security assessing methodology that adopts a five-party, three-stage model to detect typical vulnerabilities of popular OAuth implementations in Android apps. Based on this framework, a comprehensive investigation on vulnerable OAuth implementations is conducted at the level of an entire mobile app ecosystem. The investigation studies the Chinese mainland mobile app markets (e.g., Baidu App Store, Tencent, Anzhi) that covers 15 mainstream OAuth service providers. Top 100 relevant relying party apps (RP apps) are thoroughly assessed to detect vulnerable OAuth implementations, and we further perform an empirical study of over 4,000 apps to validate how frequently developers misuse the OAuth protocol. The results demonstrate that 86.2% of the apps incorporating OAuth services are vulnerable, and this ratio of Chinese mainland Android app market is much higher than that (58.7%) of Google Play.

1. INTRODUCTION

OAuth is the most widely accepted authorization protocol for third-party applications to obtain access to HTTP services provided by mainstream service providers (SPs) such as Google, Facebook, Twitter, and Sina Weibo. Over one billion OAuth-based user accounts are now provided by different service providers, which is a factor that attracts Android applications to integrate OAuth service as one of their user management mechanisms. Major SPs provide the relying parties (RPs) OAuth SDKs to integrate their OAuth services, and RP developers are required to follow the documents specified by the SPs. However, OAuth implementation on Android apps is quite complex for developers. Because SPs usually develop the OAuth SDKs referring to their implicit security requirements and business logic, there exists no standard OAuth implementation for Android apps. Meanwhile, developers may misunderstand SPs' specifications on how to use the SDKs. Since OAuth on mobile platform is so sophisticated and situation varies on how apps use OAuth, details in the OAuth authorization and authentication processes should be assessed thoroughly to ensure the security.

Current researches have pointed out many security threats of OAuth caused by incorrect implementation. As a multi-party protocol that is mainly designed to support third-party authentication and authorization, the application scenario is quite unique and difficult to comprehend. It is a combination of both password authentication and trusted third-party delegated authentication. If an SP provides an insecure OAuth implementation or the developers misunderstand/misuse the protocol, it will introduce serious security vulnerabilities. Wang et al. discover that the ignorance of implicit security assumptions of SP when using OAuth SDKs often leads to security issues [18]. Chen et al. reveal the significant differences of OAuth implementation between Web and mobile platforms. They further analyze the typical incorrect implementations introduced by mobile app developers due to the platform divergences [6]. However, those proposed researches do not consider multiple mobile apps as an integrated analysis object and most of the analyses are ad hoc. Due to the lack of systematic security assessment, security problems caused by incorrect OAuth implementations are far from being noticed or solved.

Without a comprehensive and systematical assessment, plenty of details and subtle cases in an Android based OAuth process may be ignored even by experienced analysts. For instance, the user-agents on the Web platform and the Android platform are different. On the Web platform, a user-agent is typically a web browser. But on the Android platform, there exist various user-agents including WebView, SP app as well as a system browser. If an RP app uses WebView as the user-agent to perform OAuth authentication or authorization, it could neither provide the isolation between

ACSAC '15, December 07-11, 2015, Los Angeles, CA, USA

DOI: http://dx.doi.org/10.1145/2818000.2818024

the user-agent and RP required by OAuth, nor verify the identity of the RP app. Another case is that previous researches rarely consider the validation of the identity of an SP app when it plays the role of a user-agent, the SP app installed in the user's smartphone can be a repackaged malicious app downloaded from an untrustful third-party market.

To help developers and security audit analysts, we propose a systematic assessment framework–AuthDroid to detect vulnerable OAuth implementations on the Android platform. Our framework adopts an extended OAuth security model that considers issues of both Web-based and mobile-based OAuth protocol to build a more comprehensive security model for Android app's OAuth usage. It covers the entire lifetime of OAuth authentication/authorization in an Android app. To perform the security assessment comprehensively, our framework expands the traditional three-party OAuth model to five-party. Apps of SP and RP are considered as important roles in our model. Our assessment then divides the OAuth authorization and authentication process into three stages according to the temporal sequence: **Stage I**: the login stage, a user clicks the social login/share button, then RP launches an authorization request and redirects the user to login the SP. **Stage II**: the authorization stage, the RP obtains an authorization grant and use it to trade for the access token from the SP. **Stage III**: the resource access stage, the RP uses the acquired token to access user's resources on SP's database. In an authentication process, RP uses the obtained resources to authenticate the user. Finally, security of the events in each stage is evaluated in two main aspects: a) whether the OAuth SDK implementations violate the security requirements of RFC specifications; b) whether developers misuse the SDKs or misunderstand SPs' OAuth specifications.

We conduct a comprehensive investigation on vulnerable OAuth implementations at the level of an entire mobile app ecosystem. The target of our investigation is a representative Android ecosystem–the Chinese mainland Android app markets. To the best of our knowledge, this is the first time an investigation on OAuth security is conducted at an ecosystem level. Based on our proposed assessment framework, we systematically evaluate 15 major OAuth implementations of the Chinese mainland app markets as well as top 100 apps that incorporate relevant OAuth services. Our AuthDroid adopts a hybrid approach with static code analysis and dynamic network traffic analysis to examine how those apps implement OAuth in Android, and particularly analyzes how RPs authenticate users with OAuth. The collected OAuth specifications, relevant SDKs, and apps are thoroughly analyzed to uncover the typical misuses of those OAuth implementations and the inconsistency with RFC specifications. We found that 14 out of the 15 SPs support at least one vulnerable OAuth implementation. According to the typical OAuth misuses summarized from our assessment of those 100 apps, we further statically analyze 4,151 apps to validate how frequently developers misuse the OAuth protocol, and find that 86.2% of the apps incorporating OAuth services are vulnerable to attacks. The result demonstrates a much higher faulty rate of the China Android app market compared with that (58.7%) of Google Play [6].

The contributions of this paper are twofold. First, we find that the root causes of OAuth security assessment omissions are due to two aspects: On the one hand, SPs and RP developers lack a security assessment guideline and an implementation reference that are able to cover the entire lifetime of the OAuth protocol. On the other, rare work has concerned about the inconsistency between the OAuth implementations adopted by the SDKs and the official standards (i.e., RFC specification). Also, the misuse of SP's SDK by developers is barely assessed. Thus developers' misunderstanding or casualness often leads to OAuth misuses in Android apps. AuthDroid helps securely implement OAuth in Android by assessing typical security vulnerabilities of OAuth implementations in Android apps in a systematic way. Second, our investigation illustrates that security analyses on OAuth should concern about restrictions from the mobile ecosystem. Service providers vary a lot in diverse mobile ecosystems. Many common security assumptions may not be followed in a certain market and thus it is hard or even impossible to implement a secure OAuth protocol in such market environment. In a well-developed mobile app ecosystem the SPs are more likely to provide secure OAuth specification, and help educate developers to implement correct authorization/authentication service. While in an immature mobile app ecosystem the SPs are more casual on both secure specification design and developer restriction.

2. SECURITY CONSIDERATIONS

OAuth 2.0 [12] is the latest version of the OAuth protocol which was originally created in late 2006. Compared to its predecessor spec, OAuth 2.0 is a more secure version, it focuses on client developer simplicity while providing specific authorization flows for different usage scenarios. Major SPs trend to obsolete OAuth 1.0 and move toward OAuth 2.0, Facebook, Sina Weibo, and Microsoft only support OAuth 2.0 now. Therefore, we mainly discuss the implementation of OAuth 2.0 (hereinafter referred to as "OAuth", unless otherwise specified) in this paper, and our analyzing targets are mobile apps on the Android platform.

2.1 Protocol Flow

A typical OAuth flow involves three parties: 1) **User**, also named as **Resource Owner**, has an account on the service provider. 2) **Relying Party (RP)**, a third-party application that wants to verify user's identity or access user's protected resources. RP needs to register its application with an SP before incorporating the SP's OAuth service. A registered OAuth application is assigned a unique *app ID* and *app Secret*. 3) **Service Provider (SP)**, who stores user's resources and offers APIs for authorized RPs to access user's information. Figure 1 describes the interaction between the three parties.

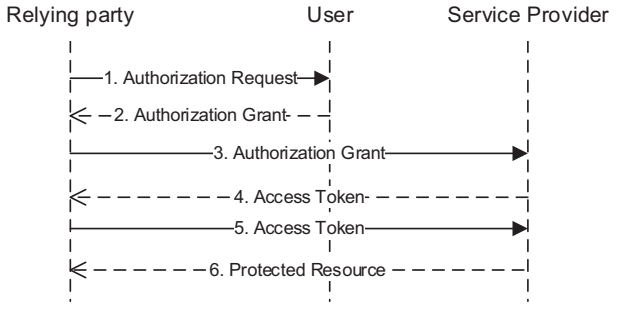

Figure 1: Abstract Workflow of OAuth

2.1.1 Authorization Grant

The authorization grant depicted in step 2 in Figure 1 is a credential representing the resource owner's authorization, which can be used by RP to obtain an access token from SP. The OAuth specification defines four grant types to exchange for access token: *authorization code*, *implicit*, *resource owner password credentials*, and *client credentials*.

The authorization code grant is a most commonly used OAuth grant type. In this grant, SP requires to authenticate the RP app by validating its app id and app secret before issuing the access token. Therefore, the app id/secret needs secure management and transmission.

The implicit grant is simpler than the authorization code grant, SP responses with an access token when an RP app requests for authorization. It removes the step to authenticate the RP app, as a result, the implicit grant is not suitable for authentication.

The latter two grants are rarely used. The credentials can only be used when the RP app is highly trusted and other authorization grant types are not available. Such RP apps are not the ones installed in mobile devices, which we assume not trustful.

In our study, we find out that the first three of the grant types are used in Android for authentication and authorization in practice.

2.1.2 User Agent

OAuth needs a user-agent to help implement the HTTP redirection mechanism [11], which directs a user to the SP or returns OAuth credentials (e.g., authorization code or access token) back to the RP. A user-agent displays authentication and authorization pages to users, and deliver messages between RP and SP.

In the web environment, browsers can accomplish the redirection by handling HTTP 302 status code, they also provide isolation between the RP and the user-agent to protect user credentials. However, it is infeasible for the user-agents on the mobile platforms to perform the same redirection. There are three types of user-agent in Android: embedded **WebView** in RP app, **SP app** and **system browser**, which vary a lot from the traditional user-agent (i.e., web browser) on the Web.

WebView is a custom webkit browser embedded in an app, which can be used to present users with the OAuth authentication and authorization pages. The hosted RP app can control it by calling the methods *loadUrl*, *setJavaScriptEnabled*, *addJavascriptInterface*, etc, which may compromise the security of OAuth.

To prevent client impersonation, authentication is required before delivering messages. The only secure authentication method is to validate the developer's key hash of the recipient [6]. Android requires that all apps be digitally signed with a certificate before they can be installed, and the app developer holds the certificate's private key. However, the RP app's developer's key hash can only be verified using a native mobile application, and as a consequence, a web-based SP inside a WebView or a system browser is unable to authenticate the RP app, using SP app as a user-agent is the only possible way to implement OAuth correctly on the Android platform. Table 1 illustrates the security features of the three user-agents.

Type	RP app validation	Isolation between user-agent and RP
SP app	Feasible	Yes
WebView	Infeasible	No
System browser	Infeasible	Yes

Table 1: Features of User-agents

2.2 Attack Surface

Before considering the concrete attack surface, we make the assumption according to the habits of common users: We first assume that the user mobile device is not compromised, which means the standard security isolation of Android is always effective. However, 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.

According to our observation, most attack surfaces of OAuth in Android relate to the incorrect implementation of authorization grants or the use of improper user-agents. We summarize these attack surfaces in the following:

User-agent hijacking: When WebView is used as the user-agent, a malicious RP app can hijack the hosted WebView to launch attacks on user authentication and app authorization [13]. The attacker is capable of stealing any information submitted by the user in the WebView and modifying information displayed in the WebView.

When the SP app plays the role of a user-agent, a malicious SP app (e.g, a repackaged one downloaded from an untrustful market) has the access to all the information it receives and forwards, including username/password, authorization code, access token, refresh token, etc.

Client impersonation: Secret management remains a challenge when the authorization code grant is used to exchange for the access token. If the app id/secret is hard coded in the RP app, an attacker can perform static analysis on the RP app to extract the client credentials. If the app id/secret is obtained during runtime and stored in the shared preferences, a malicious third-party app installed in the device may attempt to read from the shared filesystem, and the app id/secret can be leaked when improper permissions are set for the shared preferences file.

With the obtained app id/secret, a malicious third-party app can impersonate the legitimate RP app to interact with SP. Meanwhile, if the RP app is not authenticated by the SP app in the authorization code grant, malicious third-party apps can register similar intent filters to the legitimate RP app to intercept the Intent sent by the SP app, and access the data inside the Intent.

Network attack surface: OAuth relies completely on SSL for confidentiality and server authentication. A network attacker could attempt to launch different attacks (e.g, SSL stripping or SSL certificate replacement) to the communication between a mobile device and an SP server, or a mobile device and an RP server. If the transport security measures are not properly set, the trans-

mission of security-sensitive information (e.g, user credentials, OAuth credentials) can be sniffed or tampered by a network attacker.

3. VULNERABILITY ASSESSMENT

3.1 Assessment Model

We first describe our proposed assessment model for OAuth implementations in Android apps. This model considers the features of OAuth implementations including the main participants and covers the entire OAuth procedure. Figure 2 depicts the model of OAuth authorization with an example (an SP app is used as the user-agent).

To better describe the features of OAuth in Android, our methodology extends the traditional three-party model of OAuth to a five-party model, which includes **user**, **SP server**, **RP server**, **RP app**, and **user-agent**. User-agent and client applications of SP and RP are considered as important roles in our model. The three-party model ignores the interactions between the RP app and the user-agent, as well as the interactions between the clients and their servers. With our five-party model, we can assess the isolation between RP app and user-agent, the authentication of RP app and SP app, and the communications among different parties thoroughly.

Meanwhile, our analysis model divides the OAuth authentication and authorization process into three typical stages according to the protocol flow: **login stage**, **authorization stage**, and **resource access stage**, to analyze the message sequences among different participates.

3.2 Assessment Methodology

The security assessment for OAuth implementation in Android is based on our five-party model. We adopt a hybrid methodology combining static code analysis and dynamic network traffic analysis to evaluate how OAuth is implemented in Android and extract the five-party model for each implementation. Most prior studies on OAuth security evaluation focus on the authorization stage. However, security flaws occurred at any stage can break the security of OAuth. Our proposed assessment methodology takes all three stages into account to conduct a comprehensive investigation on the implementation of OAuth in Android. In detail, security of the events in each stage is evaluated from two aspects: (a) **SP inconsistency**: the inconsistency between the SP's OAuth implementation and the OAuth official specification. (b) **RP misuse**: RP developers's misuse of SPs' SDKs or misunderstanding of SPs' OAuth specifications.

As the OAuth implementations of different SPs vary from each other, and an SP may have different implementations for different authorization grant types or user-agent types, we need to figure out how typical SPs implement OAuth in Android to help build the five-party model. OAuth specifications, relevant SDKs, and client applications of these SPs are analyzed thoroughly to extract the use-agent, key methods, requests and parameters used in each implementation.

Taking the characteristics of Android into consideration, we compare the extracted model of each implementation against the RFC specifications [11, 12], to identify the vulnerabilities introduced by the SPs.

We also audit a small sample set of popular RP apps which use the OAuth services provided by the typical SPs, to figure out how OAuth is implemented in Android apps in detail. The basic characteristics of the key parameters (e.g., app secret, authorization code, access token) are gathered in this analysis.

Based on the prior knowledge, we build AuthDroid, a semi-automatic assessment framework for security assessment of OAuth implementations in Android apps. It extends the Androguard [1] reverse engineering suite.

AuthDroid first utilizes static pattern matching to extract basic elements (e.g., user-agent, identity of SP) from RP app to get the major participants in the five-party model. The following aspects are assessed automatically:

Service Providers. AuthDroid analyzes the *strings.xml*, *AndroidManifest.xml* and hard-coded strings in RP apps to extract the character strings relevant to each SP's OAuth service, such as *share.tencent.OAuthV2AuthorizeWebView*, *share_tencent*, etc. Then the SPs whose OAuth services are incorporated by the RP app can be determined.

User-agents. AuthDroid analyzes the methods used to implement OAuth and the activities registered in the manifest file, to figure out the supported user-agents for each SP in the RP app.

Hard-coded Strings. AuthDroid extracts the suspicious strings which are likely to be the app secrets for each RP app based on the gathered characteristics of the app secrets. These suspicious strings are analyzed further to exclude the false positives.

SSL Validation. AuthDroid examines apps with respect to inadequate SSL validation. This functionality is achieved by referring to MalloDroid [8].

After the static analysis, we can first determine the identify of the SPs and the type of user-agents. Then, since a certain SP uses uniform parameter names in different authorization grant types, we can determine the corresponding model(s) to guide the traffic analysis.

AuthDroid performs dynamic network traffic analysis to obtain the message sequences among different participants. The network traffic is generated manually and analyzed automatically in this part. We use Burp Suite [2] as the intercept proxy to capture the requests/responses and identify the key parameters in different stages, like *app id*, *app secret*, *authorization code*, *access token*, etc, as shown in Figure 2.

Specifically, as the user authentication methods adopted by RPs vary greatly, the critical parameters used by each RP for authentication need to be extracted first and their roles in the authentication should be further analyzed. AuthDroid employs MitmProxy [3] to perform differential fuzzing analysis to help understand how RPs authenticates users with OAuth (two accounts A and B are registered manually for each SP before analysis).

To eliminate the redundant parameters, AuthDroid uses account A to perform OAuth authentication first, dumping the OAuth authentication traffic with MitmProxy and extracting the authentication request. AuthDroid then checks each parameter by replaying the extracted request with this parameter removed. If the request results in a successful authentication, this parameter is a redundant element. After obtaining the critical parameters, AuthDroid uses account B to perform OAuth authentication and dump the

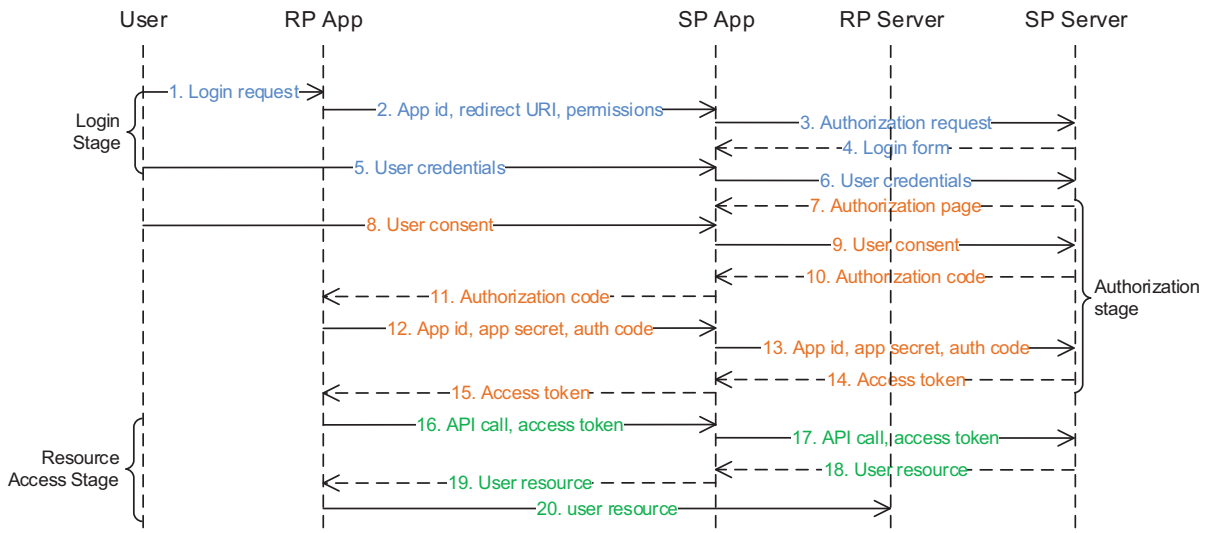

Figure 2: Model of OAuth authentication with SP app as the user-agent

traffic. AuthDroid extracts the authentication request of account B as well, replacing one parameter in this request a time with the corresponding parameter of account A, and replaying the request with other parameters unchanged. This operation is executed iteratively until the last critical parameter is checked. Then AuthDroid analyzes RP server's responses to study how the RP server deals with these parameters and identify the root causes of their failure to authenticate users with OAuth. The analysis of user authentication with OAuth is detailed in Section 4.2.3.

3.3 Vulnerabilities

Based on the attack surfaces identified in Section 2.2, we give comprehensive security considerations for OAuth in Android beyond those in the OAuth 2.0 specification [12], and summarize the most severe vulnerabilities as follows.

Vulnerability I (V_1): Improper User-agent

Using an embedded WebView as the user-agent may compromise the security of OAuth, a malicious RP app can launch attacks on both the user authentication and application authorization [5].

Moreover, WebView and System browser are not suitable for RP app authentication, using them as the user-agent may result in client impersonation.

Vulnerability II (V_2): Lack of Authentication

Apps can leak information when delivering messages without authenticating the recipient. If the SP app fails to authenticate the RP app, a malicious third-party app can register similar intent filters to the legitimate RP app to intercept the Intent sent by the SP app, and access the data contained in the Intent, such as OAuth credentials.

Meanwhile, some SP apps provided by third-party markets are potentially repackaged malicious apps, as OAuth protocol lacks authentication on SP apps by design, once these malicious SP apps are installed in the mobile device and used as the user-agent, it could receive and forward all the security-sensitive information, including the user credentials.

Vulnerability III (V_3): Inadequate transmission protection

Messages are supposed to be transmitted securely during an OAuth transaction. A network attacker could try to eavesdrop transmission of the user credentials and the OAuth credentials between the mobile device and the SP server. If the security-sensitive information is transmitted in a plaintext or the transport security measures (e.g., TLS) are not correctly implemented, the security of OAuth cannot be guaranteed.

Vulnerability IV (V_4): Insecure secret Management

When the authorization code grant is used for authentication or authorization, the app secret management remains a challenge. Some RPs prefer to hard code the app id/secret in the client application. Attackers can perform static analysis on the RP apps downloaded from app markets to obtain the app id/secret from source code or binary, with the app id/secret he can obtain access tokens on behalf of the attacked RP app.

Some other RP apps prefer to obtain app secrets from their back-end servers at runtime instead of hard coding it, and cache it in a shared preferences file in the local device. However, if the permissions of the secret file are not properly set, for example, the file is world-readable, other malicious apps installed in the mobile device can read from the shared filesystem to obtain the app secrets. The situation is the same when RP app caches the authorization code or access token in shared preferences.

Vulnerability V (V_5): Problematical server-side validation

When RP app sends a resource request along with its access token to the SP server, some SP servers may not validate the access token correctly and return resources according to the user id. If the communication between RP and SP is not well protected, or the RP app can be controlled by an attacker, the attacker could exploit this vulnerability by modifying the request and replaying it to obtain other users' protected resources without their authorization.

Vulnerability VI (V_6): Wrong authentication proof

Some RP servers prefer to use wrong authentication proofs (e.g, access token or user id) to authenticate the users. If the communication between the RP app and RP server is not well protected, the attackers may obtain the OAuth credentials, which can be used to access user's protected resources

SPs	Installs	Authorization grant	User-agent	RP app authentication	Enforced HTTPS
Sina Weibo	100-500 million	Auth code/Implicit	W/A	yes	yes
Tencent Weibo	10-50 million	Modified implicit	W/A	no	no
Qzone	100-500 million	Modified implicit	W/A/B	no	no
QQ	1-1.5 billion	Modified implicit	W/A/B	no	no
Wechat	1-1.5 billion	Auth code	A	yes	yes
Youdao Note	10-50 million	1.0a/Auth code	W/A	yes	W no/A yes
Evernote	10-50 million	1.0	W	no	yes
Yixin	10-50 million	Auth code	W/A	yes	no SSL
Douban	5-10 million	Auth code	W	no	yes
Renren	50-100 million	Auth code/Implicit	W/A	no	W no/A yes
Kaixin	10-50 million	Auth code/Implicit/Password	W	no	no SSL
Baidu	100-500 million	Auth code/Implicit	W	no	no
Taobao	0.5-1 billion	Auth code/Implicit/Password	W/A	no	no
Laiwang	10-50 million	Auth code	A	no	no
Alipay	100-500 million	Auth code/Implicit	W/A	no	no

Table 2: Features of major SPs' OAuth implementations. Three authorization grants are used: auth code (Authorization code grant), implicit (implicit grant), password (Resource Owner Password Credentials). Three types of user-agent in Android: WebView (W), SP app (A) and System browser (B).

or even login to user's RP account without authorization.

4. EMPIRICAL EVALUATION

Overall, we investigated 4,151 apps with AuthDroid to study their properties with respect to the usage of OAuth, of which 1,372 incorporate OAuth services. Figure 3 illustrates the numbers of apps using the OAuth services provided by each SP or third-party OAuth SDK. Specifically, the numbers of apps using WebView as the user-agent are indicated in the chart as well.

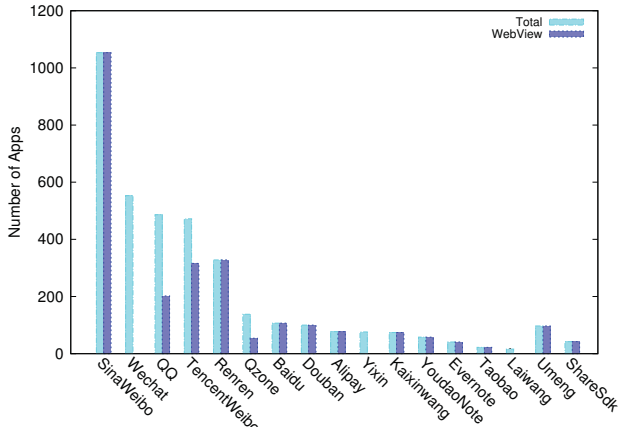

Figure 3: The numbers of apps using OAuth services provided by each SP

Meanwhile, we investigate 15 major SPs' OAuth implementations for the App market in Chinese mainland, including *Sina Weibo*, *Tencent Weibo*, *Qzone*, *QQ*, *Wechat*, *YoudaoNote*, *Evernote*, *Renren*, *Kaixinwang*, *Baidu*, *Taobao*, *Laiwang*, and *Alipay*. Table 2 lists the features of these SPs' OAuth implementations.

In the following, we introduce the results of our investigation for different stages of OAuth flow. Table 3 summarizes the vulnerabilities in each SP's OAuth implementation in different stages.

4.1 SP Inconsistency

4.1.1 *Login Stage*

OAuth specifies a process for users to authorize RP apps to access their protected resources stored in the SP servers without sharing their credentials. In other words, the RP apps should not be able to obtain user credentials (e.g., username/password).

SP	Stage I		Stage II			Stage III
	V_1	V_3	V_1	V_2	V_3	V_5
Sina Weibo	√	×	√	×	×	√
Tencent Weibo	√	×	√	√	×	×
Qzone	√	×	√	√	×	×
QQ	√	×	√	√	×	×
Wechat	×	×	×	×	×	×
Youdao Note	√	√	√	×	√	×
Evernote	√	×	√	√	×	×
Yixin	√	×	√	×	×	×
Douban	√	×	√	√	×	×
Renren	√	√	√	√	√	√
Kaixin	√	√	√	√	√	×
Baidu	√	×	√	√	×	×
Taobao	√	√	√	√	√	×
Laiwang	×	×	×	√	×	×
Alipay	√	√	√	√	√	×

Table 3: Vulnerabilities of each SP's OAuth implementation.

As shown in Table 3, 13 (86.7%) SPs support to use the embedded WebView as the user-agent which cannot provide the required isolation between the user-agent and the RP client. A malicious RP app can take full control of the embedded WebView and obtain user credentials with no effort.

Only four SPs enforce to use HTTPS to protect the communication. *YoudaoNote* and *Renren* enforce HTTPS when using SP app as the user-agent, while WebView plays the

role of the user-agent, HTTPS is not enforced. *Kaixinwang* even transmits the user credentials in a plaintext without any protection. Some SPs use HTTP to transmit user credentials in the login phase but encrypt the password independently, which is safe as well.

4.1.2 Authorization Stage

The authorization stage is the core part of an OAuth flow. However, the implementation of OAuth in this stage is error-prone for the most mobile developers.

The WebView user-agent threats this stage as well, a malicious RP app can modify the requested permissions displayed in the authorization page to get more permissions than the user grants, which may leak user's resource.

When the SP app plays the role of the user-agent, the RP app and the SP app are supposed to authenticate each other. However, only four SPs support to validate the developer's key hash of the RP app. Other SPs do not provide any method to verify the identity of the RP app. We repackaged an RP app (the developer's key hash of the repackaged RP app is different from the legitimate one) to test the rest 11 SPs, all client applications of these SPs trusted the repackaged RP app and performed OAuth authorization without security warning. Furthermore, none of the 15 SPs provide a way for RP apps to authenticate the SP clients.

Among these SPs, *Taobao* and *Kaixinwang* support to use the insecure resource owner password credentials grant, which may leak the user credentials in transmission. We find a popular music app named *Xiami* (install base between 30,000,000 and 50,000,000) uses this grant implemented by *Taobao* for authentication and leaks the user credentials.

4.1.3 Resource Access Stage

Most studies in recent years focused on the security of the authorization phase [10, 16, 17, 18, 20]. The resource access phase is often neglected by SPs. In the authorization code grant, access token are bound to both the user id and the app id, while in the implicit grant, access tokens are only bound to the user id. This would mean that the access token used in the implicit grant is different from the one used in the authorization code grant, SPs need to provide different validation mechanisms for different tokens. When RP trades an access token for user's protected resources, the corresponding SP needs to verify the binding between the token and the user id in the authorization code grant, as well as the binding between the token and the app id. And in the implicit grant, the binding between the token and the user id needs to be verified.

Sina Weibo and *Renren* implement OAuth incorrectly in this phase, both of them fail to verify the binding between the access token and the uid, their servers only check the validity of the access token and return user resources according to the uid, even if the uid does not match the one bound to the access token. A malicious RP app can exploit this flaw by modifying the user id in the resource request to obtain other users' resources without authorization.

4.2 RP Misuse

4.2.1 Login Stage

Most RP apps prefer to use WebView as the user-agent, in case the SP app is not installed in the Android device. Of all the 1,372 apps incorporating OAuth service, 1,182 (86.2%) of them support to use WebView as the user-agent. However, most of these RP apps fail to validate the certificate used in the SSL connection. Meanwhile, RP apps installed in the Android device are potentially repackaged malicious apps downloaded from untrusted third-party markets, the RP app authentication cannot be accomplished when WebView plays the role of the user-agent, as the developer's key hash can only be achieved by a native app. If an SP app is used as the user agent, the malicious RP app may fail in the RP app authentication. Even if some SPs ignore the RP app authentication, the malicious RP app can simply get the authorization code or access token, rather than the user credentials. While in the case of the WebView user-agent, the malicious RP app can obtain the user credentials and bring more threats.

The RP app is also responsible to verify the identity of the SP app, it can perform the validation in the same way as the SP app authenticates the RP app. If the validation fails, RP app should stop the login operation and provide visual feedbacks to the user. However, no RP app verifies the identity of the SP app in our investigation.

4.2.2 Authorization Stage

We use AuthDroid to analyze the 1,372 apps incorporating OAuth service, 946 of them contained suspicious strings which are likely to be the app secret. To exclude the false positives, we implement a client to simulate the OAuth authorization. A suspicious string and the corresponding app id is used as the app id/secret pair in the authorization code grant to exchange for the access token, the false positives are ruled out based on the server responses. As the authorization code in the authorization grant can only be used once, we utilize the Robotium [4] framework to help perform the authorization process automatically, so as to obtain the authorization code for each test. Finally, we got 383 (27.9%) apps containing app secrets, including six apps sharing the same app secret for *Sina Weibo*, and two sharing the same app secret for *Tencent Weibo* and *QQ*. Most of the false positives are hard-coded keys used in other cryptographic algorithms, it remains a challenge to distinguish them from the app secret, as the app secret appears to be completely meaningless just like the cryptographic keys.

For those RP apps who avoid hard coding the app secret, the key management can be insecure as well. We analyzed the top 100 apps from a third-party market, and find an app named *Hupu Jogger* (install base between 6,000,000 and 9,000,000) sets the secret file permissions incorrectly and leaks the app secrets and OAuth credentials.

Even if the RP developers obey SPs' specifications strictly, the resulting apps may be insecure as well. The SDKs and documents provided by the SPs may have security flaws as discussed in previous sections and in Wang et.al's study in [18]. RP developers are supposed to avoid using the insecure implementations provided by SPs, otherwise the security flaws in the insecure implementations will be inherited. 85 of the top 100 apps incorporate *Sina Weibo*'s OAuth service, we evaluate these apps to figure out how RPs misuse SPs SDKs. The vulnerabilities exist in the RPs are shown in Figure 4.

4.2.3 Resource Access Stage

Figure 5 depicts the top 100 apps' support for OAuth authentication and authorization when using different SPs.

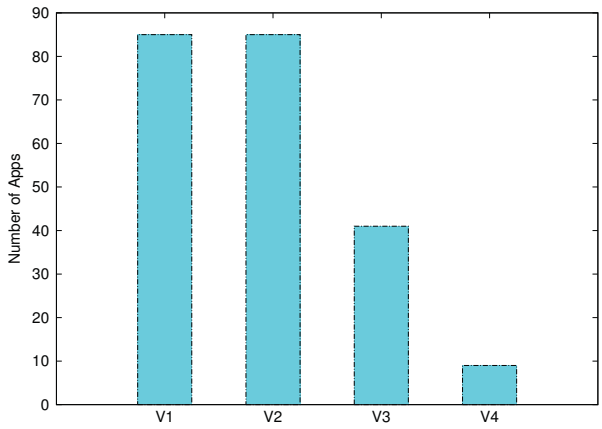

Figure 4: RPs' misuse of Sina Weibo's SDK

OAuth was designed for authorization. Even though it's likely to use OAuth to build an authentication protocol, OAuth is not an authentication protocol in nature. During the process of authentication, RP needs to know who the current user is and whether or not he is present. Unfortunately, OAuth tells RP none of that. Access token is designed to be opaque to the relying party. RPs may trade the access token for a number of attributes (e.g. a unique identifier) about the user to know who he is, but these attributes cannot prove the user's presence with the RP app. We study how RPs authenticate users using OAuth as the base and whether they make the common mistakes described in [9].

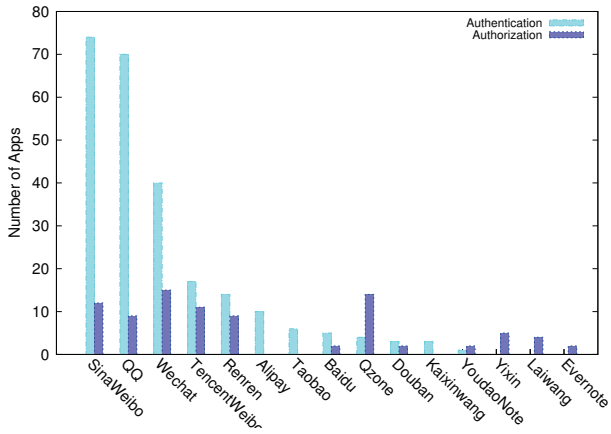

Figure 5: The numbers of apps using the SPs to conduct authentication or authorization

Sina Weibo's OAuth service is the most popular service used for authentication as shown in Figure 5. We take *Sina Weibo* for example and analyze the 85 RP apps incorporating *Sina Weibo*'s OAuth service. In our investigation, RPs prefer to use signature, access token, authorization code, user id or some other parameters as authentication proof. Figure 6 illustrates the situation. 31 (36.5%) RPs made mistakes when using OAuth for authentication, most RPs neglect to take transport security measures to protect the authentication process.

13 (17.81%) of them use access token as the authentication proof, and the access token is transmitted in a plaintext in the authentication request. It is nature to assume that the access token can be used as the proof of authentication, as the access token is usually issued after a user being authenticated by the SP server. However, that's not the only way to obtain the access token. Refresh tokens can be used to obtain access tokens as well, a refresh token is issued to the RP app by the SP server to obtain a new access token before the current access token expires, this process can proceed without user intervention. In this method, as the access token cannot be parsed or understood by RPs, RPs may trade the access token for user's attributes to identify the user. Even so, the attributes cannot tell whether the user is present or not.

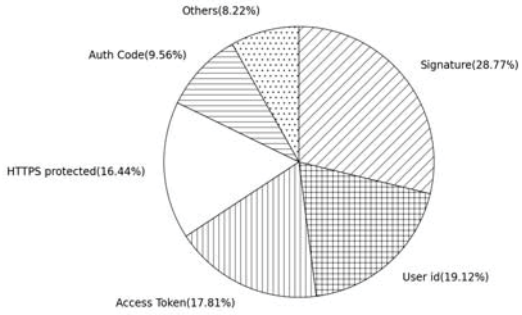

Figure 6: Authentication proof

14 (19.12%) RPs use user id along with the access token to authenticate users, and the access token is transmitted without protection. This is similar to the former one, but if the SP suffers from *Vulnerability V*, an attacker can modify the user id in the authentication request to login to other users' RP account without authorization. *Sina Weibo* is such an SP.

There are other seven RPs use the authorization code to authenticate users. As the authorization code can only be used once, RP servers can exchange the authorization code for an access token, an authorization code is useless to an attacker if he is not aware of the app secret. Moreover, the authorization code is short-lived, such an authentication can only be accomplished when the user is present.

Considering the complexity of the OAuth protocol and the lack of expertise among Android app developers in general, service providers are supposed to provide standard solutions to use OAuth for authentication. They can design APIs that are more intuitive and easier to use, to minimize the emergence of such vulnerabilities.

4.3 MBaaS

As each of the OAuth services provided by SPs has its own API that must be individually incorporated into an RP app, it is a time-consuming and complicated process for app developers. Many developers utilize **Mobile Backend as a Service (MBaaS)** to simplify the incorporation. MBaaS provides mobile app developers with a way to link their applications to backend APIs while also providing features such as user management, push notifications, and integration with social networking services. These services are provided through customized SDKs and APIs.

We examined three popular MBaaS platforms in China, including *Umeng*, *ShareSDK* and *Frontia*. All of their im-

plementations exist some vulnerabilities. *Umeng* is used by more than 420,000 apps, and 60,144 apps incorporate *ShareSDK*'s service. Both of them support all Chinese social platforms and major international ones: *QQ*, *Sina Weibo*, *Wechat*, *Facebook*, *Twitter*, etc. However, they provide incorrect specifications and poor secret management. *Umeng* guides developers to hard code app id/secret for some SPs in the integration codes, and *ShareSDK* requires users to store all app id/secret pairs in a configuration file named "sharesdk.xml". Under the circumstances, an attacker can obtain the app secrets easily by performing static analysis for the RP apps. *Frontia* is an MBaaS platform owned by *Baidu*, it supports OAuth services provided by *Sina Weibo*, *Tencent Weibo*, *Qzone*, etc. Instead of hard-coding app secrets in the application, *Frontia* requires developers to mandate secrets to an application created on the *Baidu* platform. The secret escrow method adds a new participant in the OAuth flow and complicates the secret management issue, which may expand the attack surface.

4.4 Case Study

While a single vulnerability may not appear to pose a significant threat, a combination of different participants with different vulnerabilities may allow attackers to seriously compromise an OAuth transaction. For example, assume that an RP's OAuth implementation suffers from *Vulnerability IV*, an attacker is unable to exploit this vulnerability to impersonate the RP app only if the corresponding SP app suffers from *Vulnerability II*, exploit can be constructed in combination with these two vulnerabilities. In the following, we present two typical cases exploiting combinations of vulnerabilities to attack OAuth transactions.

Case #1: *Sina Weibo & Phoenix News*

The first case involves *Sina Weibo*, the most popular social network application in China with around 280 million active users, and *Phoenix News*, a famous news application with more than 100 million downloads. In this case, *Sina Weibo* is the SP and *Phoenix News* is the RP. *Phoenix News* relies on *Sina Weibo* to authenticate users.

In our investigation, during the login stage, the server of *Sina Weibo* responses the login request with user's basic information as well as the access token. As the login request is transmitted in HTTP (with the password encrypted independently), the access token is returned in plaintext. In the resource access stage, the *Phoenix News* app returns the obtained user id and access token in plaintext to its server to authenticate the user, which are exactly the ones in *Sina Weibo*'s login response.

Sina Weibo exposes two security problems in our assessment: a) it returns access token before user's authorization and transmits it without protection; b) the same access token is issued to different RP apps, which means that the access token is not bound to RP app when OAuth is used for authentication, any RP app with the access token could access user's protected resources without authorization. For *Phoenix News*, it suffers from *Vulnerability VI* in this process as it uses an incorrect method to authenticate user.

Based on these vulnerabilities, we have constructed an attack that uses the *Sina Weibo* account to login to other RP apps to obtain the same access token as the one obtained by the *Phoenix News*, and utilize the access token to login to Phoenix News without victim's authorization. We have reported this flaw to both apps' manufacturers and the flaw has been patched now.

Case #2: *Renren & Hupu Jogger*

The SP in this case is *Renren*, a Facebook-like social network application with 83 million active users, and the RP is *Hupu Jogger*, a popular sports app in China.

When using the *Renren* account to login to *Hupu Jogger*, during the authorization stage, *Hupu Jogger* obtains the app id/secret from its back-end server and stores them in a shared preferences file named *hupurun.xml*, which is global-readable and can be accessed by any other app installed in the same device. And in the resource access stage, when *Hupu Jogger* attempts to access user's protected resources, *Renren* returns information according to the submitted user id.

In this case, *Hupu Jogger* stores the secrets incorrectly as it set the wrong file permission of the shared preferences file, and *Renren* fails to validate the binding between the access token and the user id, anyone can access a specific user's protected resources by submitting the user id and an unrelated valid access token.

The combination of these vulnerabilities may allow attackers to gather sensitive information of many different users without authorization. A malicious third-party app could attempt to read from the shared filesystem on the device to obtain the app id/secret of *Hupu Jogger*, then the malicious app is able to impersonate *Hupu Jogger* to interact with *Renren*. In this way, the attacker can modify the user id in the resource request during the resource access stage and replays the request to *Renren* to gather sensitive information corresponding to the modified user id.

5. RELATED WORK

A large body of recent work [6, 7, 11, 14, 15, 16, 17, 18, 19] has sought to discover various attacks against OAuth implementations. Wang et al. studied the security-critical logic flaws in commercial Web SSO systems that can totally defeats the purpose of authentication [17]. Sun et al. performed a security analysis through empirical examinations of real-world SP and RP implementations and uncovered several critical vulnerabilities that allow an attacker to gain unauthorized access to the victim user's resource [16]. Homakov highlighted some of the security flows in Facebook's OAuth 2.0 implementation and Google Chrome that enable an attack on user's OAuth secrets [7]. While much of the prior work focuses on the security of OAuth implementations on Web platforms, our work aims to analysis the security problems which may arise when implementing OAuth on the Android platform.

The issues with OAuth implementations in the mobile environments have been studied by several others. Chen et al.'s work revealed that mobile platforms significantly differ from the Web and pinpointed the key portions in OAuth protocol flows that are confusing for mobile application developers [6]. Wulf showed stealing passwords is easy in native mobile apps despite OAuth [5]. McGloin et al. concluded the OAuth 2.0 threat model and security considerations, they warned the risk of using an embedded browser in the end-user authorization process [11]. Unlike previous studies, our work is not focused on the individual attacks, but rather the systematical security assessment of OAuth implementations in Android. While most of the prior work focuses on the authorization phase, our analysis covers the entire authentication/authorization process. Apart from the

authorization phase, we take the login stage and the resource access stage into consideration as well. Meanwhile, we identify the attack surface of OAuth in Android and summarize the most severe vulnerabilities in OAuth implementations.

6. CONCLUSION

In this paper, we report a security study of OAuth implementations in Android applications at ecosystem level. The study proposes a systematic assessment framwork AuthDroid that has detected 6 security vulnerabilities in OAuth implementations in Android. With AuthDroid, we conduct an investigation on 15 major OAuth service providers and more than 4000 popular apps from the Chinese mainland app markets. The results demonstrate a higher ratio of OAuth misuse (86.2% compared with 58.7% of Google Play) and many complex vulnerabilities due to the combination of two main factors: the inconsistency between service provider's design and RFC standard, and the misunderstanding of developer to the specification of service providers.

7. ACKNOWLEDGEMENTS

We would like to thank our shepherd, Long Lu, and the anonymous reviewers for their insightful comments that greatly helped to improve the manuscript.

This work was supported in part by the National Key Technology Research and Development Program of China under Grants No.2012BAH46B02, the National Science and Technology Major Projects of China under Grants No.2012-ZX03002011, and the Technology Project of Shanghai Science and Technology Commission under Grants No.1351150-4000 and No.15511103002.

8. REFERENCES

[1] Androguard. https://github.com/androguard.

[2] BurpSuite. http://portswigger.net/burp.

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

[4] Robotium. http://code.google.com/p/robotium/.

[5] Stealing Passwords is Easy in Native Mobile Apps Despite OAuth. http://goo.gl/QskLq.

[6] Eric Y Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. OAuth Demystified for Mobile Application Developers. In *Proc. of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

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

[8] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, Lars Baumgärtner, and Bernd Freisleben. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *Proc. of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[9] Nir Goldshlager. End User Authentication with OAuth 2.0. http://oauth.net/articles/authentication, 2013.

[10] Internet Engineering Task Force (IETF). The OAuth 1.0 Protocol (RFC 5849), 2010.

[11] Internet Engineering Task Force (IETF). OAuth 2.0 Threat Model and Security Considerations (RFC 6819). 2013.

[12] Internet Engineering Task Force (IETF). The OAuth 2.0 Authorization Framework (RFC 6749), 2013.

[13] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on WebView in the Android system. In *Proc. of the 27th Annual Computer Security Applications Conference*, 2011.

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

[15] Ethan Shernan, Henry Carter, Dave Tian, Patrick Traynor, and Kevin R. B. Butler. More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations. In *Proc. of the 12th Detection of Intrusions and Malware, and Vulnerability Assessment International Conference (DIMVA)*, 2015.

[16] San-Tsai Sun and Konstantin Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *Proc. of the 19th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2012.

[17] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *Proc. of the 33rd IEEE Symposium on Security and Privacy (SP)*. IEEE, 2012.

[18] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *Proc. of the 22nd USENIX Security Symposium*, 2013.

[19] IETF OAuth WG. OAuth Security Advisory: 2009.1. http://oauth.net/advisories/2009-1/.

[20] Yuchen Zhou and David Evans. SSOScan: Automated Testing of Web Applications for Single-Sign-On Vulnerabilities. In *Proc. of the 23rd USENIX Security Symposium*, 2014.