OAuth Demystified for Mobile Application Developers

Eric Chen
Carnegie Mellon University
eric.chen@sv.cmu.edu

Yutong Pei
Carnegie Mellon University
ypei@andrew.cmu.edu

Shuo Chen
Microsoft Research
shuochen@microsoft.com

Yuan Tian
Carnegie Mellon University
yuan.tian@sv.cmu.edu

Robert Kotcher
Carnegie Mellon University
rkotcher@andrew.cmu.edu

Patrick Tague
Carnegie Mellon University
patrick.tague@sv.cmu.edu

ABSTRACT

OAuth is undoubtedly a highly influential protocol today, because of its swift and wide adoption in the industry. The initial objective of the protocol was specific: it serves the *authorization* needs for *websites*. What motivates our work is the realization that the protocol has been significantly re-purposed and re-targeted over the years: (1) all major identity providers, e.g., Facebook, Google, Microsoft and Twitter, have re-purposed OAuth for *user authentication*; (2) developers have re-targeted OAuth to the *mobile* platforms, in addition to the traditional web platform. Therefore, we believe that it is very necessary and timely to conduct an in-depth study to demystify OAuth for mobile application developers.

Our work consists of two pillars: (1) an in-house study of the OAuth protocol documentation that aims to identify what might be ambiguous or unspecified for mobile developers; (2) a field-study of over 600 popular mobile applications that highlights how well developers fulfill the authentication and authorization goals in practice. The result is really worrisome: among the 149 applications that use OAuth, 89 of them (59.7%) were incorrectly implemented and thus vulnerable. In the paper, we pinpoint the key portions in each OAuth protocol flow that are security critical, but are confusing or unspecified for mobile application developers. We then show several representative cases to concretely explain how real implementations fell into these pitfalls. Our findings have been communicated to vendors of the vulnerable applications. Most vendors positively confirmed the issues, and some have applied fixes. We summarize lessons learned from the study, hoping to provoke further thoughts about clear guidelines for OAuth usage in mobile applications.

1. INTRODUCTION

The essence of Software-as-a-Service (SaaS) is that different components of a web or mobile application are developed by different providers. The need for secure authentication and authorization across companies has never been so important. Today, the most widely adopted protocol in this space is OAuth. It is currently deployed by numerous major companies including Facebook, Google, Microsoft and Twitter.

Originally, OAuth was designed to provide a secure *authorization* mechanism for *websites*. It defines a process for end-users to grant a third-party website the access to their private resources stored on a service provider. The third-party website is often referred to as the consumer [26], client [27], or relying party [41] (we will use the term "relying party" exclusively in this paper). There are two versions of OAuth protocols, OAuth 1.0 and OAuth 2.0 [26, 27]. They are both actively use by real-world websites.

Ever since OAuth was successfully adopted by the industry, major companies have re-purposed OAuth for *authentication* as well. That is, the protocol enables a user to prove his or her identity to a relying party, utilizing his or her existing session with the service provider. The web industry's trend to obsolete other protocols and move toward OAuth is very decisive – the new authentication mechanisms provided by the aforementioned companies are all OAuth-based. Therefore, despite the fact that neither OAuth 1.0 or OAuth 2.0 documentation explicitly purposes itself for authentication, OAuth is now a de-facto authentication and authorization protocol.

OAuth for mobile applications. The years of OAuth's evolution, since 2007, happen to be the same period of the boom of mobile applications. Authentication and authorization are as needed by mobile applications as by traditional websites. Almost inevitably, OAuth became *the protocol* for implementing authentication and authorization functionality in mobile applications. According to our study, more than 24% of the 600 top Android applications taken from several Google Play categories use OAuth. We studied the OAuth 1.0 and OAuth 2.0 protocol documentation carefully, and realized that secure usage of OAuth for mobile applications could be mysterious since the protocol is primarily designed in the mindset of traditional web technology rather than mobile platforms. Motivated by earlier work to *demystify* the `setuid` system calls on UNIX systems [8], we believe that it is very necessary and timely to conduct a study to demystify OAuth for mobile application developers.

We observe that, despite its wide deployment, the OAuth protocol is very complicated for average developers to comprehend. For example: (1) the use-case for authentication is left unspecified for both OAuth specifications, which causes developers to assemble a set of OAuth concepts and crypto primitives more or less through their own intuition. (2) Both

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660323.

OAuth 1.0 and 2.0 use browser redirection extensively for delivering OAuth tokens. However, it is unclear how this operation can be performed in mobile applications without browser's involvement. (3) The two versions of OAuth protocols target different scenarios and contain different sets of concepts. One version does not subsume or obsolete another. They simply coexist with many unconsolidated discrepancies. (4) The OAuth 2.0 specification is extremely extensible, and in a way, underspecified by design. For all these reasons, we wondered how likely mobile application developers were to misinterpret the OAuth protocol, or fall into pitfalls that the protocol leaves unspecified.

Our work. Our work consists of two pillars: (1) an in-house study of the OAuth protocol documentation in the mindset of mobile application development that aims to understand what might be ambiguous or unspecified in the protocol; (2) a field-study of popular mobile applications to see how well developers fulfill the authentication and authorization goals. First, we study three canonical OAuth flows described in the OAuth specifications [26, 27] – the OAuth 1.0 flow, the OAuth 2.0 authorization code grant and the OAuth 2.0 implicit grant. We then analyze the two use-cases of OAuth: authorization and authentication, and factor out properties of the protocol that must be satisfied in order to achieve security. We proceed to show how the three aforementioned protocol flows realize these security properties. Finally, we investigate how OAuth is being interpreted by real-world mobile application developers, and point out several common misconceptions that ultimately undermine the security properties of OAuth.

Our study was conducted on 149 popular OAuth-capable mobile applications. The focus was to dissect the rationales behind different OAuth implementations, and to understand why some of these applications are secure while others are seriously vulnerable. Our analysis reveals that real-world OAuth implementations are extremely diverse; rarely do two service providers (or even relying parties of the same service provider) share the same protocol flow (e.g., Hulu, Spotify and Instagram all use Facebook for authentication but each have their own protocol flows). We believe that this diversity of mobile implementations reflects the real issues with OAuth. Not only is the protocol defined over multiple specifications with two different use-cases, but also its mobile usage is poorly defined and underspecified. This forces developers to resort to their own interpretations when implementing the protocol. Our study shows that 59.7% of these implementations were faulty and vulnerable to attacks.

A real example. The result of our study highlights not only the complexity of the OAuth protocol, but also the misunderstandings amongst many OAuth developers. For instance, the security property of the OAuth access token differs across OAuth 1.0 and OAuth 2.0. In OAuth 1.0, each access token is bound to and can only be used by the relying party the access token was issued to. However, an access token in OAuth 2.0 (which is also referred to as a "bearer token") can be used by any party in possession of this token. "Any party in possession of a bearer token can use it to get access to the associated resources" [28]. In reality, the confusion between these two types of access tokens is reflected through several erroneous OAuth implementations. We found that a known vulnerability (described in references [7, 43]) still existed in the Friendcaster Android application's Facebook authentication service, which allowed an attacker to sign into an honest Facebook user's Friendcaster account. This error was caused by Friendcaster blindly accepting an access token received from a user's device then using this token to exchange for the user's Facebook ID. A malicious application could easily obtain a legitimate access token from a user, then use this access token to log into Friendcaster as the user. When we reported this issue to Friendcaster, the developers were confused because they thought Facebook's access tokens were bound to relying parties and checked with every API call: "From Facebook's perspective, the API calls wouldn't appear to be originating from Friendcaster, but the attacker's own app." In other words, the developers of Friendcaster thought Facebook uses the OAuth 1.0 interpretation of the access token, while in reality Facebook uses the OAuth 2.0 interpretation. We find similar misinterpretations of the OAuth protocol to be common amongst mobile developers. We will explain these cases in more detail in Section 5 and summarize the lessons learned from them.

The rest of the paper is organized as follows: Section 2 provides the background for OAuth 1.0 and 2.0. We give an overview about our study in Section 3. The details of the study are presented in Section 4 and Section 5. We describe related work in Section 6 and conclude in Section 7.

2. OAUTH BACKGROUND

The OAuth discussion group began in 2007 as a community effort to allow third-party access to users' protected resources without the need to reveal their credentials. The first version of the OAuth protocol (OAuth 1.0) was drafted in October 2007 and published as an RFC in April 2010 [26]. Since then, the protocol has gone through numerous revisions. The most notable changes to the protocol were released in October 2012 as the OAuth 2.0 framework [27].

2.1 OAuth 1.0 and OAuth 1.0a

When the first version of OAuth was drafted, there existed another popular protocol called OpenID for third-party user authentication [13]. Hence, OAuth was mainly designed to address an issue that was not covered by OpenID – secure API access delegation (i.e., authorization). While the term "API authentication" was occasionally used to describe the functionality of OAuth [29], the protocol specification itself was never meant for user authentication.

The OAuth 1.0 protocol flow is illustrated in Figure 1. All dashed lines in our figures represent browser redirection and solid lines represent direct server-to-server API calls (e.g., a SOAP or REST API call). In addition, parameters inside square brackets are signed using shared secrets, which we describe in detail in Section 4.1. For now, we present a summary of the protocol flow.

- **Unauthorized request token** (Step 1,2) – First, the relying party obtains a request token from the service provider using a direct server-to-server call.

- **Authorized request token** (Step 3-5) – Then, the relying party redirects the user to the service provider (possibly via a browser redirection) with the request token as a URI parameter. Then, the user grants the relying party access to his/her protected resource and is redirected back to the relying party.

- **Access token** (Step 6,7) – At this point, the relying party can exchange the request token for an access token using another direct server-to-server call with the service provider.

- **Protected resource** (Step 8,9) – Finally, this access token is used to obtain the user's protected resource.

Two years after the release of the OAuth 1.0 draft, a session fixation attack was discovered against the request token approval step of the protocol [23]. To fix the vulnerability, a revision to the original protocol was released (called OAuth 1.0a), which included a verification code to the final request token response (Step 5 of Figure 1). This code is used during the access token exchange to prove that the user completing the access token exchange is the same user who granted access. For simplicity, we will use the term "OAuth 1.0" to refer to OAuth 1.0a for the rest of this paper.

2.2 OAuth 2.0

OAuth 1.0 (and OAuth 1.0a) had several notable limitations for its usage scenarios. However, instead of augmenting the existing protocol, the working group decided to alter the specification completely to create a different protocol – OAuth 2.0. This decision was the result of a "strong and unbridgeable conflict" between different interest groups, according to a departing lead author of OAuth 1.0 [22].

One major change introduced by OAuth 2.0 was the concept of *bearer tokens* [28]. That is, a user's access token was no longer bound to a relying party; any party in possession of this token could freely access the user's protected resource. In addition, OAuth 2.0 also offers four methods for exchanging access token; these methods are referred to as *grants* and they can be viewed as different "versions" of OAuth 2.0. Our study in Section 3 reveals that out of the four grant types in OAuth 2.0, only two were used in practice for authorization and authentication – implicit grant and authorization code grant. We illustrate these two grants in Figure 2a and Figure 2b, and briefly describe them below.

2.2.1 Implicit grant

The implicit grant is the shortest of all OAuth flows. It consists of two steps. First, the user is redirected to the service provider to grant the relying party access to his/her protected resource. After the permission is granted, the service provider redirects the user back to the relying party along with an access token. The relying party can then use this access token to exchange for the user's protected resource.

There are two core features that differentiate the implicit grant from other OAuth flows. First, with exception to the final protected resource request, every message in the protocol is exchanged through the user agent (e.g., using browser redirection). Second, the implicit grant does not require the relying party to present a shared secret to the service provider. This is ideal for the mobile environment, where the relying party resides on an untrusted device.

2.2.2 Authorization code grant

The authorization code grant augments the implicit grant by adding an additional step to authenticate the relying party. After the user grants permission to the relying party, the service provider redirects the user back to the relying party with an *authorization code* (instead of an access token). Then, this authorization code is used to exchange for the user's access token through a direct server-to-server call. In this access token exchange step the relying party has to include its own identity, so the service provider can verify that the authorization code is granted to the same party.

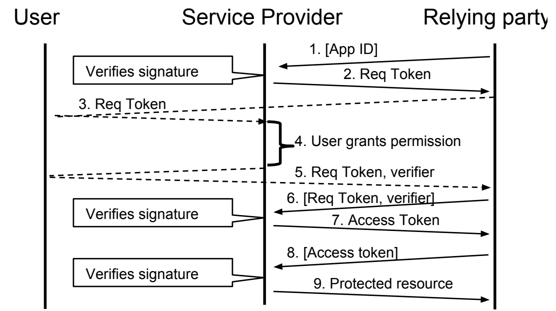
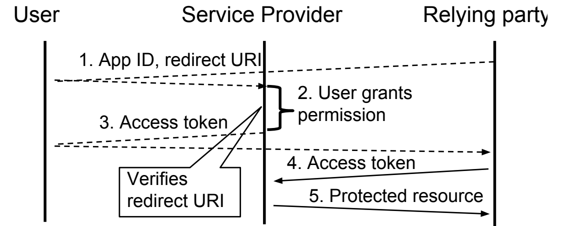
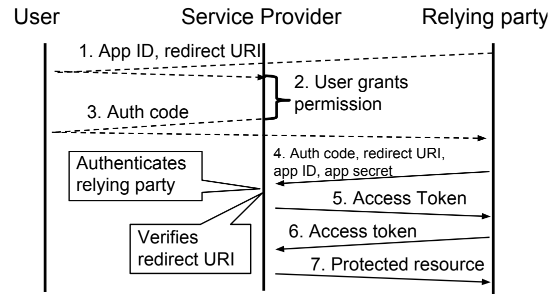


Figure 1: OAuth 1.0 and OAuth 1.0a.



(a) Implicit grant



(b) Authorization code grant

Figure 2: Two grant types of OAuth 2.0

3. OUR STUDY

Our study consists of two main components, which are explained in detail in the next two sections:

- We first focus on understanding and comparing the protocol specifications of OAuth 1.0 and 2.0 in order to pinpoint their key steps in authentication and authorization. Pinpointing these key steps plays an important role in our work because they are implemented by different mechanisms on the mobile platforms, as compared to those on the web platform. Section 4.2 will elaborate on how such differences lead to security issues which mobile application developers may not foresee.

- In order to understand how real-world developers interpret and implement OAuth, we conducted a com-

prehensive study on 149 popular mobile applications. These applications included 133 Android applications from the following Google Play store categories: top 300 free applications in all categories, top 200 free applications in social and top 100 free applications in communication. In addition, we manually selected 16 popular iOS and Android OAuth applications (e.g., Quora and Weibo) that were not included in the top charts. 25 (16.8%) applications used in our study were service providers, 126 (84.5%) were relying parties and 2 (1.3%) were both service providers and relying parties. Furthermore, 52 (41.3%) of the relying parties were using OAuth for authorization, the remaining 74 (58.7%) were using it for authentication. Our study revealed that 59.7% of these protocol implementations were faulty and vulnerable to attacks. These results confirm our suspicion that, for a large population of developers, how to use OAuth securely on mobile applications is indeed unclear. Section 5 explains a set of representative cases among the vulnerable applications that we studied.

4. OAUTH SPECIFICATIONS AND MOBILE PLATFORMS

In this section, we start by studying three canonical OAuth protocol flows: OAuth 1.0, OAuth 2.0 implicit grant and authorization code grant, and their two use cases: authorization and authentication. Since the use case of authentication is largely unaddressed by the current OAuth specifications, we offer our insights on how to achieve it using different versions of OAuth. Furthermore, we demonstrate key factors that make the existing OAuth specifications error-prone (or even insufficient) for implementations on mobile platforms.

4.1 Dissecting the OAuth specifications

Our analysis is focused on authorization and authentication. For each of these two problems, we identify key elements within the two specifications (OAuth 1.0 [26] and OAuth 2.0 [27]) that account for their security. We focus on three OAuth protocol flows: OAuth 1.0 (Figure 1), OAuth 2.0 implicit grant (Figure 2a) and authorization code grant (Figure 2b).

Note that, as a prerequisite of any OAuth protocol flow, the relying party must obtain an ID and a secret from the service provider. This is typically done by registering the relying party application through the service provider.

4.1.1 *Authorization*

Authorization is a process that enables an end-user to grant a relying party access to his or her protected resource stored on a service provider. The security audience for authorization is the service provider. That is, the user's sensitive information is located on the service provider, and the service provider must verify that the protected resource is sent to the same party that the user had granted access to.

Although the descriptions of the three protocol flows in the OAuth specifications are fairly complicated, we believe that each has a few key elements for secure authorization, as we identify below.

OAuth 1.0 – The OAuth 1.0 specification requires every token request and protected resource request to be signed by the relying party using the secret obtained during the application registration stage. The security for authorization is achieved between Steps 8 and 9 of the protocol (see Figure 1) when the service provider verifies the signature of the protected resource request. Assuming that the relying party secret is only known to the relying party and the service provider, this step ensures that the receiver of the protected resource is the same party that the user granted the request to.

OAuth 2.0 implicit grant – The OAuth 2.0 framework requires the relying party to provide a redirection URI when registering itself to the service provider. This redirection URI is an essential element for achieving security for authorization using the implicit grant. In Step 2 of the implicit grant (see Figure 2a), the redirection URI provided by the relying party is checked against the registered redirection URI in the service provider's database. If the two URIs match, it means that the user is granting access to the same relying party that the access token is sent to.

OAuth 2.0 authorization code grant – The authorization code grant augments the implicit grant by adding an additional step that allows a service provider to verify the identity of a relying party. Security for authorization is achieved in between Step 4 and Step 5 of the protocol (see Figure 2b). At this point, the service provider authenticates the relying party using its application ID and secret provided in Step 4 (recall that this ID-secret pair is obtained by the relying party during the application registration stage). After authenticating the relying party, the service provider must verify that this relying party is the same as the one that the user had granted access to in Step 2 of the protocol. This step ensures that the protected resource is sent to the correct relying party.

4.1.2 *Authentication*

Although many websites and applications use OAuth for authentication, these use-cases are unspecified by both OAuth standards (1.0 and 2.0). This section provides our insights on the protocol details that are important to achieve secure authentication.

Authentication is a process where an end-user signs onto a relying party account by proving to the relying party that he/she is a certain user on the service provider. Unlike authorization, the security audience for authentication is the relying party. That is, the protected resource is located on the relying party (i.e., the user's relying party account), not the service provider.

To leverage OAuth for authentication, a relying party uses the OAuth authorization flow to request the user's account ID from the service provider. Once this account ID is retrieved from the service provider, it can be used to identify the user on the relying party. However, because the security goals of authentication are very different from those of authorization, not all OAuth protocol flows are secure for authentication.

In general, in order to determine whether an authorization protocol flow can be used for authentication, there are two properties that the relying party must ensure. First, the relying party must ensure that the user ID received from the service provider cannot be tampered with by the user. Otherwise, an adversary can impersonate arbitrary users. Second, the relying party must check that OAuth tokens used to retrieve the user ID is granted to the same relying party. If this check is not done, an adversary could use

tokens issued to a malicious application to sign onto users' benign relying party account.

We now examine the three canonical OAuth protocol flows and analyze whether they can be used for authentication.

OAuth 1.0 – There are two aspects of the OAuth 1.0 protocol that make it secure for authentication. First, the user ID exchange in Step 8 and 9 of the protocol (see Figure 1) are done using server-to-server API calls. These calls cannot be tampered by the user. Second, the signature check in Step 8 ensures that the access token used to retrieve the user ID is granted to the same relying party. That is, the user is using this access token to sign onto the same relying party.

OAuth 2.0 implicit grant – Unfortunately, the implicit grant is insecure for authentication. Since access tokens in OAuth 2.0 are no longer bound to relying parties (i.e., anyone with a valid access token can use it to exchange for the user ID), it is impossible to verify whether the user is using an access token to sign onto the same relying party. A malicious relying party could obtain access tokens from users signing onto itself, then use these access tokens to log into a benign relying party, impersonating these users.

OAuth 2.0 authorization code grant – We mentioned previously that the reason why the implicit grant cannot be used for authentication is because OAuth 2.0 access tokens are not bound to relying parties they are issued to. This problem is mitigated in the authorization code grant using an additional parameter called the *authorization code*. A service provider implementing the authorization code grant does not send access tokens to relying parties using browser redirection. Instead, access tokens are delivered using server-to-server API calls (Steps 6 and 7 of Figure 2b), which cannot be tampered by a malicious user. Access tokens are exchanged using authorization codes (Steps 4 and 5 of Figure 2b), where each authorization code is bound to the relying party it was issued to. By verifying the redirection URI associated with the authorization code in Step 4 of the protocol, the service provider can make sure that the access token is always sent to the relying party that the user tries to authenticate to.

4.2 Differences between mobile and web platforms that affect OAuth security

The previous subsection showed that the three OAuth protocol flows differ significantly. Each flow is carefully designed with unique checks (e.g., app secret or redirection check) and message delivery methodologies (e.g., browser redirection or server-to-server API call) to provide security. In this subsection, we shift our attention to mobile platforms and identify key factors that make OAuth difficult to adopt for mobile applications. Specifically, we demonstrate that since the OAuth specifications were written primarily for the web community, many core concepts that are essential for security cannot be trivially converted to the mobile world. This causes mobile developers to frequently resort to their own interpretations while implementing the protocol, making the development process error-prone.

Different redirection mechanisms – One web concept that is heavily used throughout both OAuth specifications is browser redirection (e.g., using the HTTP 302 status code). This mechanism is used for directing a user to the service provider and delivering OAuth tokens (e.g., request token or access token) to the relying party.

While the process of handling HTTP 302 status code is well defined for browsers, it is unclear how to perform the same redirection on mobile applications. Unfortunately, both OAuth specifications ignore this factor by labeling the methodology for performing redirection as an implementation detail. For example, the following excerpt was taken from the OAuth 2.0 specification [27].

> This specification makes extensive use of HTTP redirections, ... any other method available via the user-agent to accomplish this redirection is allowed and is considered to be an implementation detail.

The closest concept related to browser redirection on mobile platforms is the custom scheme mechanism on iOS and the Intent mechanism on Android. They are used by mobile applications to register custom URI schemes for themselves [2, 18]. When a URI with a custom scheme is visited inside a mobile browser or an embedded browser (e.g., WebView), the mobile OS will subsequently launch the application registered with the custom scheme. Using this approach, a service provider application can simulate a browser redirection by sending the access token to a URI with the honest relying party's custom scheme. However, unlike for web applications, where the Domain Name System (DNS) maintains a global one-to-one mapping between a host name and a unique web principal, mobile operating systems allow multiple applications to register for the same custom scheme. This many-to-one mapping between multiple locally installed applications and a single custom scheme is fundamentally flawed for redirection.

Lack of application identity – The absence of a proper redirection mechanism for mobile applications introduces another complication for mobile service providers – their inability to identify message receivers. The security of OAuth depends on the service provider's ability to send confidential messages (e.g., access tokens) to specific relying parties.

For web applications, this is accomplished through browser redirection, where the browser is entrusted to deliver the message to the web principal given by a host name. Additionally, the DNS is responsible for maintaining an association between a host name and a web principal. Hence, assuming that the DNS and the browser are well-behaved, a web-based service provider can effectively send confidential messages to a relying part of its choosing. Unfortunately, the same concept does not apply to mobile applications. That is, for both iOS and Android, installed applications do not have an origin associated with them. It is not immediately obvious how a mobile service provider could guarantee that a confidential message is only sent to its intended recipient.

Although in practice, different approaches have been used to facilitate message passing between different principals, most of these methods are incorrect. We demonstrate later in Section 5 that many developers have ill-conceived notions of the real recipients for different message passing mechanisms on mobile devices.

Client-side protocol logic – Although client-side message passing mechanisms such as the Android Intent [17] are frequently used as a replacement for browser redirection, there is a clear distinction between using browser redirection and using a client-side message passing mechanism. When a browser redirection is used by a web service provider, the request is directly sent to the relying party's server and han-

dled by server-side logic. However, when a client-side message passing mechanism (such as the Android Intent) is used by a mobile service provider, the logic that first processes the message is located on the user's device. Although in theory, all of the core protocol logic for a mobile relying party could reside on a server, while the client-side application is only used to deliver and receive messages to and from the server, this is difficult to do in practice. From our observations, many mobile developers incorporate parts of their applications' core logic into their client-side applications. The issue with this practice is that when OAuth is used for authentication, the user's device is assumed to be untrusted. Hence, including security sensitive protocol logic or data with the client-side application could allow a malicious user to bypass security checks and steal confidential data (e.g., the relying party secret).

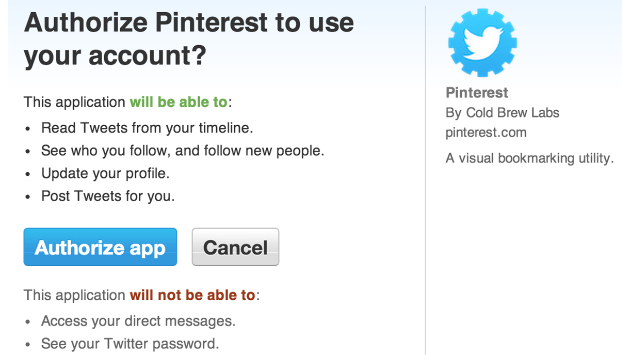


Figure 3: Pinterest's OAuth dialogue forged using a stolen application secret.

5. STUDY OF REAL-WORLD MOBILE APPLICATIONS

In the previous section, we highlighted several portions of the OAuth specifications that are critical for security, and also pointed out several differences between the mobile platforms, as compared to the web platform which OAuth is primarily designed for. Of course, a main concern our study aims to address is how well real developers are able to tackle these critical portions in the protocol specifications, and whether they have sufficient awareness about the platform differences. As mentioned in Section 3, our study includes 616 real-world mobile applications. Our selection was unbiased: the selected applications are the top 300 free applications in Google Play, the top 200 free social applications, the top 100 free communication applications, as well as 16 other clearly popular applications.

We found that 149 out of the 616 applications implement OAuth. Surprisingly, 89 of them (59.7%) were incorrectly implemented, which resulted in vulnerabilities. We have given our best effort to communicate with the application vendors, including Facebook, Twitter, Instagram, Quora and many others. In most cases, we have received their positive acknowledgments.

In this section, we delve into a number of representative vulnerability cases. The goal is to demonstrate different ways real-world developers interpret the OAuth specifications, and to provide our study result about the pervasiveness of each of these issues.

5.1 Storing relying party secrets locally

One common issue was that many developers fail to understand the purpose of the relying party secret, and thus store it locally inside the client application. We believe that the terminology of OAuth confuses developers significantly – the relying party secret is referred to as the "consumer secret" by OAuth 1.0 and "client secret" by OAuth 2.0, where the terms consumer and client are used by each specification to describe the relying party. These names are extremely misleading for developers who have never studied the specifications. For application developers, it is very reasonable to believe that the term consumer or client is referred to the user. Hence, many relying parties believed that it was safe to bundle their secrets with their mobile applications.

Two notable developers making this mistake were Pinterest and Quora. Both Pinterest and Quora used Twitter as a service provider for authentication, and both of them bundled their relying party secrets with their mobile applications. To make the matter worse, after obtaining their secrets using simple reverse engineering, we discovered that the same secrets were used for the Pinterest and the Quora web applications. Since Twitter mainly used OAuth 1.0, this gave an adversary the ability to generate arbitrary OAuth request tokens for Quora and Pinterest. Using these tokens, we (acting as the attacker) could direct users (both web and mobile users) to a legitimate Twitter page with the dialogue box shown in Figure 3. Once the user clicks authorize, Twitter's access token will be sent to the attacker, giving him/her full access to the user's account.

We have reported this problem to Pinterest and Quora, both of them have acknowledged this issue and revoked their application secret. Quora also took the extra step of disabling their Twitter authentication mechanism completely for its Android application.

5.2 Using authorization flows for authentication

Another common confusion amongst mobile developers is treating authorization and authentication as the same problem. Vulnerabilities due to this issue were described in the literature [7, 43]. As we mentioned in Section 4.1, the use-case of authentication is not considered in both the OAuth 1.0 and the OAuth 2.0 specifications. However, the terms "authenticate" and "authentication" are still used frequently in both specifications. For example, Section 3 of OAuth 1.0 [26] and Section 2.3 of OAuth 2.0 [27] describe a method called "client authentication". In actuality, these sections describe the method for which the relying party proves its identity to the service provider using its application ID and secret, not the method to identify the user. However, it is easy to see that without contexts, this can lead to developer confusions. Many relying parties using various service providers fall victim to this misunderstanding. In this section, we primarily focus on Facebook's relying parties, but the same concept can be applied to others as well.

Facebook advocates its mobile relying parties to use the implicit grant to access core APIs. The implicit grant is a simpler method for authorization, and in a way provides better immunity against developer mistakes since it does not rely on the secure storage of relying party secrets (since this secret is not used in the implicit grant). However, as

mentioned previously in Section 4.1, the implicit grant is not safe to use for authentication. This is because the access token used in the implicit grant is not bound to its intended relying party. Hence, an adversary could use a user's access token issued to the malicious application to login as the user for the benign relying party's application.

Facebook realized this issue, and added an additional step to enhance its implicit grant for authentication, called the *appsecret_proof*. This step essentially transformed the implicit grant into a hybrid between itself and the OAuth 1.0 flow. This security-enhanced implicit grant is shown in Figure 4a. The key difference between the security-enhanced implicit grant and the regular OAuth 2.0 implicit grant is the addition of an appsecret_proof parameter. This parameter is a cryptographic hash of the access token using the relying party secret as its key that is included with every Facebook API call. To provide security for authentication, Facebook verifies during each API call (Step 4 of Figure 4a) that the principal that provides the appsecret_proof is the same principal that the access token was issued to (in Step 2). Unfortunately, because the use-case of authentication is not well understood by many mobile developers, the security-enhanced implicit grant is seldom used in practice by Facebook's relying parties.

We now present our observations on the different Facebook protocol flows real-world mobile developer were using for authentication. Our study included 72 relying parties in total, all of which were using Facebook for authentication.

Usages of the regular implicit grant. We observed a large number (61 out of 72, or 84.7%) of the relying parties using the standard implicit grant for authentication. All of these applications were vulnerable to the attack previously described in Section 4.1 against the implicit grant. Some notable applications included Avast Mobile Security & Antivirus and Instagram. We have reported our findings to all the vulnerable applications. So far 22 (36%) of them have acknowledged our reports, of which 11 have fixed their issues by switching to the security enhanced version of the implicit grant. Facebook rewarded us with a $5000 bounty for the vulnerability we discovered in Instagram.

Correct usages of appsecret_proof. Of the 11 relying parties that used the security enhanced implicit grant (i.e., using appsecret_proof) for authentication, 10 had the correct implementation (e.g., following the protocol flow described in Figure 4a). Examples of these applications include Hulu and airbnb.

Incorrect usage of appsecret_proof. Interestingly, we discovered a different interpretation of the appsecret_proof flow by the mobile application Keek (a social video application with more than 60 million users), illustrated in Figure 4b. In Keek's flow, the appsecret_proof was sent from the user's mobile device (i.e., Keek's mobile application) to Facebook's server. When this proof was accepted by Facebook and returned to Keek's mobile application, Keek informed its server that the appsecret_proof was accepted by Facebook. At this point, Keek fell back to the standard implicit grant, and chose to complete the user ID exchange using the standard implicit grant. Unfortunately, the issue with this flow is that a malicious user launching an attack against the implicit grant could forge Facebook's response in Step 6 of Figure 4b (since the attacker is also the user), and convince Keek that Facebook accepted the appsecret_proof. This would negate the purpose of the appsecret_proof check and downgrade the protocol into an implicit grant. We have reported this problem to Keek, but they have not responded. Our contact in Facebook offered to follow up with all vendors who are subject to this issue, including Keek.

Although we focused on Facebook's relying parties in this section, the problem is universal. For example, 71.4% of Google relying parties also had the same misconception, and were using authorization flows for authentication.

5.3 Handling redirection in mobile applications

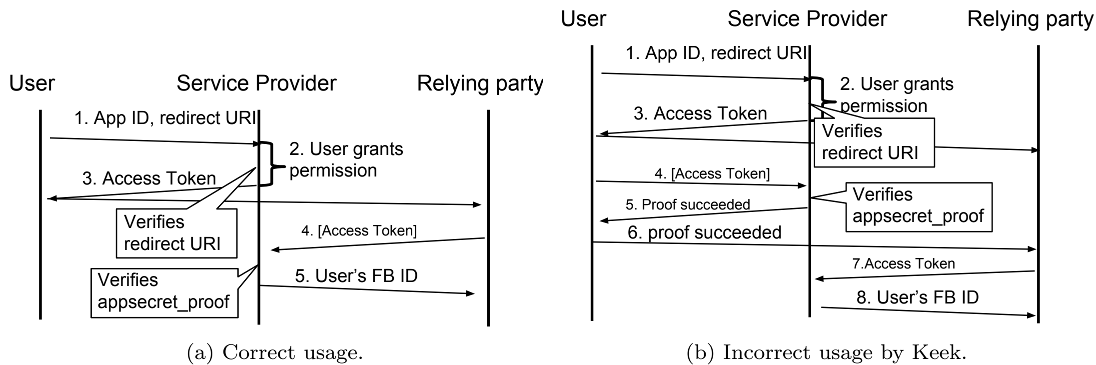
In Section 4.2, we pointed out that there currently exist no definitive, platform-independent mechanism to perform redirection for mobile applications. In this section, we present four methodologies used by real-world developers for handling redirection: iOS custom scheme, Android Intent, mobile browser and WebView. In addition, we analyse the security of each of these methods.

5.3.1 Custom scheme and Intent

Previously in Section 4.2, we discussed that the custom scheme mechanism in iOS and the Intent mechanism in Android bear close resemblance to browser redirection. It is not immediately obvious whether they can be used securely for redirection. Recall the issue with these mechanisms is that it is difficult to determine the true recipients of their messages. According to Apple, if two iOS applications attempt to register the same custom scheme, behavior regarding which app will be referenced is undefined [1]. As for Android, we discovered that many mobile service providers were not directly invoking their relying party applications (e.g., using Intents), but rather sending data through the return values of their own applications (e.g., when invoked by a relying party). However, when passing data through return values, additional checks are needed to verify that the caller application matches with the intended recipient of the data. We investigated several service providers using custom schemes and Intents, including Google, Facebook and Foursquare. We present our results below.

For iOS, we concluded that no service provider in our study was using custom schemes correctly. That is, no service provider could correctly determine the identity of its message recipient. When we presented this problem to Facebook, Facebook acknowledged this issue and added that "There is unfortunately no completely secure way to transfer an access token from our application to a client application." This problem has severe implications because the scheme mechanism is used by Facebook's iOS application to deliver access tokens to their relying parties. Since Facebook cannot guarantee that an access token is sent to its intended relying party application, there is currently no secure way of performing authentication using Facebook on iOS (even the security-enhanced implicit grant described in Section 5.2 is vulnerable).

Fortunately, the Android platform is slightly different. Our study revealed that both of the two major Android service providers (i.e., Facebook and Google) were using the Intent mechanism securely. Unlike for service providers in iOS, Android service providers can actually verify the recipient's identity when Android Intent is used for message passing, using the Android key hash. By default, every Android application in the Google Play store is signed using its developer's secret key (this secret is different from the OAuth relying party secret). When an Android relying party reg-



(a) Correct usage.

(b) Incorrect usage by Keek.

Figure 4: Usages of the Facebook appsecret_proof flow. Parameters inside square brackets are cryptographically hashed using the relying party secret.

isters itself with the service provider, the service provider would ask the relying party to provide a hash of its Android developer key. Then, when the service provider wishes to send sensitive messages (e.g., access token) to the same relying party, it can use the following code to fetch for the developer's key hash:

```
relying_party =  Activity.getCallingPackage();
dev_key_hash = getPackageManager().
       getPackageInfo(relying_party,
            PackageManager.GET_SIGNATURES);
```

Once the relying party's key hash is obtained, the service provider can check if this key hash matches with the key hash provided by the developer during its application registration stage (this is akin to checking for the redirection URI). If the two key hashes match, then the service provider can trust that the message is returned to the correct relying party. We have verified that both Google and Facebook use this mechanism to deliver access tokens to Android applications. Out of all the access token delivery methods, this is the only method in Android that was found to be secure.

5.3.2 Mobile browser and WebView

Since the OAuth specification does not specify how to perform user-agent redirection for mobile applications, it may seem natural to use a mobile browser or an embedded browser (i.e., WebView [3, 19]) to perform web-based OAuth redirections on mobile devices. Previous work showed some instances of insecure WebView usages jeopardizing security [31, 42]. It is valuable to understand how pervasive and fundamental the problem is in our set of applications.

The WebView usage is very common for service providers that utilize a single protocol flow for both web and mobile relying parties. In our study, we observed several notable service providers that fall into this category, including Twitter, Microsoft LiveID, Flikr, and Renren. Unlike Facebook and Google, these service providers do not facilitate OAuth flows for mobile relying party using their own mobile applications. Instead, they choose to use their websites to conduct all mobile OAuth transactions.

Many mobile developers naïvely believe that since the OAuth specification is specifically designed for web usages, one can securely apply it to mobile platforms by using the web-based flow inside a mobile web browser (or a WebView instance). This is a common misconception, as we have not found a single case in our study where a mobile browser or WebView is used securely for OAuth.

The fundamental reason why one cannot securely perform OAuth transactions between a mobile relying party and a web-based service provider is that it is difficult, if not impossible, for the service provider website to determine the identity of the mobile relying party. To the best of our knowledge, there currently exists no secure method in either iOS or Android to allow a service provider website to deliver sensitive OAuth tokens (e.g., request token or access token) to the honest relying party application. In our study, we observed two flawed methods used by web-based service providers to deliver OAuth tokens to mobile relying party applications. These two methods are described below.

- **Using custom schemes and custom Intent filters** – One way for a web-based service provider (inside a mobile browser or WebView) to deliver OAuth tokens to a mobile relying party is by redirecting the user to a URI with the relying party's custom scheme. Recall in Section 5.3.1, we concluded that: (1) the custom scheme mechanism in iOS is insecure, and (2) the custom Intent filter mechanism in Android can be used securely by verifying the relying party's developer key hash. Unfortunately, an Android relying party's developer key hash can only be verified using a native mobile application. A web-based service provider inside a browser cannot verify the relying party's key hash without using the relying party as an oracle.

- **Using URI parameters** – Another technique for sending access token to a mobile relying party is by directly attaching the access token to the service provider's URI (e.g., `provider.com/?token=TOKEN`). If the service provider is contained inside a WebView, the relying party that hosts the WebView instance can recover this URI using WebView API calls (e.g., getURL() in Android). For instance, the Renren Android SDK was using this method to perform its access token exchange. Unfortunately, when using this technique, there is currently no way for the embedded service provider to determine the identity of the host application.

Interestingly, the problem with message passing between a WebView instance and its host application is similar to the

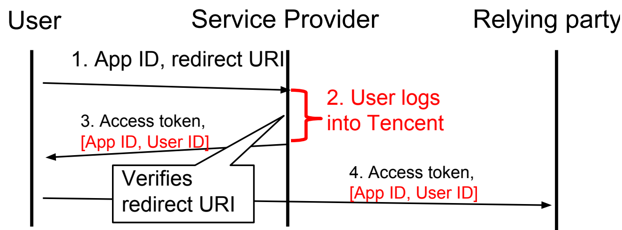


Figure 5: Tencent's enhanced implicit grant for authentication. The variables inside the square brackets are cryptographically hashed using a secret key known to only Tencent.

problem described by Barth et al. several years ago regarding inter-frame communications in web browsers [6]. In this work, the authors proposed a method called PostMessage that allows embedded principals to specify the origin of the message recipient. A similar technique for mobile applications has been proposed by Wang et al. [42]. However, this proposal is yet to be adopted in practice by mobile operating systems.

5.4 Inventing home-brewed protocol flows

Since the OAuth specification does not specify the use-case of authentication, instead of leveraging existing authorization flows, several service providers have decided to come up with their own "OAuth-based" protocol flows. In this section, we study one of these home-brewed OAuth protocols and demonstrate the difficulties in designing a completely secure authentication flow.

Tencent is a popular Chinese OAuth service provider that owns Tencent Weibo (a micro-blogging platform with 825 million users [40]) and QQ (an instant messaging application with 798 million active accounts [39]). Tencent claimed to provide authentication using the OAuth 2.0 implicit grant. Upon investigation, we discovered that the implicit grant used by Tencent is actually a modified version of the OAuth 2.0 implicit grant. We illustrate Tencent's implicit grant in Figure 5 and analyze it below.

Tencent's developers seemed to understand that because access tokens in OAuth 2.0 are not bound to their relying parties, the standard implicit grant is inherently insecure for authentication. In order to make the implicit grant safe for authentication, Tencent added a new ID hash parameter into the protocol flow. This ID hash is a concatenated string of the relying party's application ID and the user's Tencent ID, cryptographically hashed using a secret key that is only known to Tencent. For authentication purposes, instead of using the access token to exchange for the user's Tencent ID, relying parties can simply use this ID hash directly *as* the user ID. Since the value of the ID hash is different for each application, an adversary cannot utilize a user's ID hash generated for one application to sign onto the user's account for another application.

Another unique attribute of Tencent's OAuth flow is how Tencent authenticates the user in Step 2 of the protocol. In the canonical OAuth 2.0 implicit grant, this step is defined as follows [27]:

> The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.

For most service providers, this step involves the user to first sign onto the service provider using her log-in credentials, then manually click through a permission dialogue box with the relying party's name and the permission scope (i.e., the type of protected resource the relying party wishes to access). Only then can the protocol transaction proceed. However, Tencent interpreted this step differently: it issued the user's ID hash to the relying party immediately after authenticating the user (without prompting the user with an additional permission dialogue box). This step seemed innocuous at first – since if the user voluntarily decided to log into the relying party application by entering her Tencent credentials, an additional permission dialogue might seem unnecessary. Unfortunately, when the protocol transaction is performed inside a WebView (which is the default method used by Tencent's official SDK), the following attack is feasible:

1. A user signs onto a malicious application using Tencent in a WebView. However, the adversary supplies Tencent with the application ID and redirection URI of a benign relying party.

2. The user authenticates with Tencent, thinking that she is signing onto the malicious application. Unfortunately, unbeknownst to the user, Tencent treats this authentication request as one that comes from the benign relying party. Before proceeding, Tencent verifies that the redirection URI supplied in Step 1 matches with the registered URI for the application specified by the app ID in Step 1 (both of which are the correct information of the benign relying party provided by the attacker).

3. Tencent redirects the user to the redirection URI that belongs to the benign relying party. This redirection request includes the user's ID hash for the benign relying party.

4. At this point, the malicious application can obtain the user's ID hash associated with the benign relying party from its WebView using getURL() in Android and currentWebView.request.URL in iOS.

After the attacker retrieves the user's ID hash for the benign relying party, she can use this to sign onto the benign relying party's application as the user.

It is important to note that this attack was enabled by two implementation details that were not well-defined in the OAuth specifications. First, Tencent used the same service provider website for both web and mobile OAuth flows. This forced its mobile relying party applications into using WebView for authentication. Second, because Tencent's user authentication step did not include a permission dialogue box, users could not determine the identity of the relying party application they were signing onto.

After we reported our findings, Tencent immediately acknowledged this issue and patched their user authentication mechanism by adding an additional permission dialogue box.

5.5 Lessons learned

As we have shown in this section, OAuth usages on mobile applications require detailed understandings and considerations about the protocol specifications and the mobile platform capabilities. We hope that this study can provoke

the OAuth Working Group to come up with clear guidelines for mobile application developers. Some lessons we learned from this study are summarized here.

General lessons. The security of OAuth partially lies in its access token delivery methodology. We showed that it is difficult for a mobile service provider to ensure that an access token is sent to its intended recipient. For this, we come up with an informal rule-of-thumb for mobile developers to decide on whether a certain mechanism can be safely used for access token delivery:

- A mechanism can be safely used to distribute access tokens if the service provider can always identify the recipient using a *globally unique identifier*.

For example, on the Web, browser redirection can be considered as a secure access token delivery method because the relying party can always be uniquely identified through its host name. In Android, an Intent can be used to securely send an access token to its intended mobile relying party application because the relying party can be uniquely identified through its developer key hash. On the other hand, a custom scheme cannot be used to transfer an access token because iOS allows multiple applications to register for the same scheme without offering an accessible method to identify these applications.

Lessons for authorization. In addition to using a secure method to deliver access tokens, another important security criterion for authorization is to ensure that the user's permission is willfully granted to the relying party. For this, the service provider should always obtain permission from a user by presenting a dialogue box including the relying party's information and the scopes of its permissions.

Lessons for authentication. When OAuth is used for authentication, the user's device must not be trusted. This rule has two implications. First, the relying party must refrain from bundling any security related protocol logic (e.g., security checks) or any sensitive information (e.g., the application secret) into its own mobile application. Second, the relying party must assume that the attacker could tamper with any data sent from the user's device.

For this reason, an integral step of the protocol is to ensure that the relying party receiving the user's ID in the last step of the protocol is the same relying party that the user intends to authenticate to. For instance, the OAuth 1.0 flow is secure for authentication because a given access token can only be used to exchange for the user's ID by the same relying party that the access token was granted to.

6. RELATED WORK

Over the years, several prominent attacks have been discovered against various OAuth implementations [23, 41, 37, 33, 43, 42, 16, 15, 24, 25]. However, despite being the subject of constant scrutiny from the security community, the OAuth protocol remains a mystery for the majority of mobile developers. Instead of focusing on individual attacks, our work aims to provide deeper insights into how real-world mobile developers interpret OAuth and why some interpretations are correct while others are not.

We are not the first to discover flaws in commercially deployed authentication protocols [41, 5, 4, 35, 38]. However, much of the prior work focuses on the security of web-based protocol implementations. Our study revealed that mobile platforms significantly differ from the Web. Hence, it can be non-trivial for developers to translate secure web-based authentication mechanisms into the mobile environment.

Our motivation for demystifying OAuth came from Chen et al.'s work on demystifying setuid UNIX system calls [8]. Recent studies also indicated situations where APIs and SDKs present enough mysteries and challenges for developers to use securely. For example, Georgiev et al. showed that developers of mobile applications were often unable to implement SSL certificate validation logic [14]. Wang et al. showed that popular SDKs often contain implicit security assumptions that developers are unaware of [43].

The issues with custom schemes have been studied by several others [9, 42]. Unlike previous studies, the focus of our work is not on the specifics of the attacks, but rather how and why these attacks happen. We highlight several nuances within the OAuth protocol that are prone to developer misconceptions when implemented in a mobile environment.

Permission re-delegation is another type of privilege escalation attack. It happens when privileged services are exposed by an application with permission to an application without permission [10, 12]. Various defenses have been proposed to mitigate permission re-delegation attacks [12, 11, 9, 21, 30]. The attacks covered in this paper do not belong to the same category as permission re-delegation attacks, because our adversary is not interested in gaining access to privileged device resources. Rather, our attacker aims to obtain application specific resources that are located on the service provider and the relying party.

Several defense mechanisms have been proposed that utilize privilege separation to secure mobile advertising libraries and to prevent click frauds [20, 36, 32, 34, 11]. However, defenses based on privilege separation cannot be used to address logic flaws induced by developers' misinterpretation of the OAuth protocol.

7. CONCLUSION

The OAuth protocol was initially designed for website authorization, but the industry has imposed additional duties on the protocol over the years. In particular, it has become *the de-facto protocol* for authentication and authorization in mobile applications. As we show in this paper, a number of key steps in the OAuth protocol flows and some concepts of OAuth are confusing, vague, or unspecified when they are put in the context of mobile platforms. The consequence is serious: 59.7% of OAuth-capable mobile applications in our study were vulnerable. The mistakes were diverse: developers did not know where to store application secrets, had confusions about the difference between authentication and authorization, used arbitrary client mechanisms to redirect secret tokens, and even invented home-brewed OAuth protocol flows. Our findings have been communicated to vendors of the vulnerable applications. Most vendors positively confirmed the issues, and some have applied fixes, which is encouraging. Nevertheless, we believe the ultimate solution to this problem has to rely on the OAuth Working Group's effort to come up with clear usage guidelines specifically targeting mobile platforms. We hope that our work provokes such an effort.

8. ACKNOWLEDGEMENTS

We thank Rui Wang and anonymous reviewers for valuable comments.

9. REFERENCES

[1] Apple Inc. Advanced app tracks. https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/AdvancedAppTricks/AdvancedAppTricks.html.

[2] Apple Inc. Implementing custom url schemes. https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/AdvancedAppTricks/AdvancedAppTricks.html#//apple_ref/doc/uid/TP40007072-CH7-SW50.

[3] Apple Inc. Uiwebview class reference. https://developer.apple.com/library/ios/documentation/uikit/reference/UIWebView_Class/Reference/Reference.html.

[4] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra. Formal analysis of saml 2.0 web browser single sign-on: Breaking the saml-based single sign-on for google apps. In *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering*, FMSE '08, pages 1–10, New York, NY, USA, 2008. ACM.

[5] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong. Authscan: Automatic extraction of web authentication protocols from implementations. In *NDSS*. The Internet Society, 2013.

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

[7] J. Bradley. The problem with oauth for authentication. http://www.thread-safe.com/2012/01/problem-with-oauth-for-authentication.html.

[8] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *Proceedings of the 11th USENIX Security Symposium*, pages 171–190, Berkeley, CA, USA, 2002. USENIX Association.

[9] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[10] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 23–23, Berkeley, CA, USA, 2011. USENIX Association.

[12] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*. USENIX Association, 2011.

[13] B. Fitzpatrick and D. Recordon. Openid authentication 1.1. http://openid.net/specs/openid-authentication-1_1.html.

[14] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM.

[15] N. Goldshlager. How i hacked any facebook account...again! http://www.breaksec.com/?p=5753.

[16] N. Goldshlager. How i hacked facebook oauth to get full permission on any facebook account (without app "allow" interaction). http://www.breaksec.com/?p=5734.

[17] Google Inc. Intent. http://developer.android.com/reference/android/content/Intent.html.

[18] Google Inc. Intents and intent filter. http://developer.android.com/guide/components/intents-filters.html.

[19] Google Inc. Webview. http://developer.android.com/reference/android/webkit/WebView.html.

[20] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 101–112, New York, NY, USA, 2012. ACM.

[21] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*. The Internet Society, 2012.

[22] E. Hammer-Lahav. Oauth 2.0 and the road to hell. http://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell/.

[23] E. Hammer-Lahav. Oauth security advisory: 2009.1. http://oauth.net/advisories/2009-1/.

[24] E. Homakov. How we hacked facebook with oauth2 and chrome bugs. http://homakov.blogspot.ca/2013/02/hacking-facebook-with-oauth2-and-chrome.html.

[25] E. Homakov. Oauth1, oauth2, oauth...? http://homakov.blogspot.ca/2013/03/oauth1-oauth2-oauth.html.

[26] Internet Engineering Task Force (IETF). The oauth 1.0 protocol. http://tools.ietf.org/html/rfc5849.

[27] Internet Engineering Task Force (IETF). The oauth 2.0 authorization framework. http://tools.ietf.org/html/rfc6749.

[28] Internet Engineering Task Force (IETF). The oauth 2.0 authorization framework: Bearer token usage. http://tools.ietf.org/html/rfc6750.

[29] Internet Engineering Task Force (IETF). Oauth core 1.0 revision a. http://oauth.net/core/1.0a/.

[30] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[31] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *Annual*

Computer Security Applications Conference, pages 343–352, 2011.

[32] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 71–72, New York, NY, USA, 2012. ACM.

[33] M. Shehab and F. Mohsen. Towards enhancing the security of oauth implementations in smart phones. In *Proceedings of the IEEE 3rd International Conference on Mobile Services*, 2014.

[34] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[35] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen. On breaking saml: Be whoever you want to be. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 21–21, Berkeley, CA, USA, 2012.

[36] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. In *IEEE Mobile Security Technologies (MoST)*, 2012.

[37] S.-T. Sun and K. Beznosov. The devil is in the (implementation) details: An empirical analysis of oauth sso systems. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 378–390, New York, NY, USA, 2012. ACM.

[38] S.-T. Sun, K. Hawkey, and K. Beznosov. Systematically breaking and fixing openid security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures. *Computers & Security*, 31(4):465–483, 2012.

[39] Tencent Holdings Limited. Tencent announces 2012 fourth quarter and annual results. `http://www.prnewswire.com/news-releases/tencent-announces-2012-fourth-quarter-and-annual-results-199130711.html`.

[40] Tencent Holdings Limited. Tencent announces 2013 first quarter results. `http://www.prnewswire.com/news-releases/tencent-announces-2013-first-quarter-results-207507531.html`.

[41] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *IEEE Symposium on Security and Privacy*, pages 365–379, 2012.

[42] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer; Communications Security*, CCS '13, pages 635–646, New York, NY, USA, 2013. ACM.

[43] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating sdks: Uncovering assumptions underlying secure authentication and authorization. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 399–414, Berkeley, CA, USA, 2013. USENIX Association.