Towards Enhancing the Security of OAuth Implementations In Smart Phones

Mohammed Shehab and Fadi Mohsen
Department of Software and Information Systems
University of North Carolina at Charlotte
Charlotte, NC, USA
{*mshehab, fmohsen*}@*uncc.edu*

***Abstract*—With the roaring growth and wide adoption of smart mobile devices, users are continuously integrating with culture of the mobile applications (apps). These apps are not only gaining access to information on the smartphone but they are also able gain users' authorization to access remote servers on their behalf. The Open standard for Authorization (OAuth) is widely used in mobile apps for gaining access to user's resources on remote service providers. In this paper, we analyze the different OAuth implementations adopted by the SDKs of the popular resource providers on smartphones and demonstrate possible attacks on most OAuth implementations. By analyzing source code of more than 430 popular Android apps we summarized the trends followed by the service providers and by the OAuth development choices made by application developers. In addition, we propose an application-based *OAuth Manager* framework, that provides a secure OAuth flow in smartphones that is based on the concept of privilege separation and does not require high overhead.**

***Keywords*-OAuth; Smartphone apps; Security;**

I. Introduction

Open standards such as Open Authorization (OAuth), allow the resource owner (user) to grant permissions to a third-party (mobile app) access to their information hosted on a resource provider (Facebook). With the OAuth technology, the users are no longer required to share their credentials with third party apps in order to grant them authorizations. In addition, OAuth allows different access granularity, where users are able to grant access to specific resources, plus there are provisions for revoking access at any time. Other related authorization approaches include Google AuthSub [1], Microsoft Live ID [2], and Yahoo BBAuth [3]. OAuth is the most adopted with over one billion OAuth-based user accounts supported by the major online service providers. Major services providers offer software development kits (SDKs) that can be included in the mobile apps to seamlessly integrate them with their services. However, if other parties choose to develop their own libraries they are required to follow the standards specified by the service providers. Through the SDKs, service providers offer their own implementations of the authentication and authorization protocols. The different OAuth implementations adopted by popular mobile SDKs vary in their security assumptions and guarantees. For instance, several mobile SDKs rely on embedded web components to execute the OAuth authentication and authorization stages, which does not provide the required isolation and can easily be exploited by malicious apps.

In this paper, we analyze the different OAuth implementations adopted by the SDKs of the popular resource providers on smartphones, which include Facebook, Twitter, Dropbox, Microsoft Live, Google Plus, Box, Instagram, LinkedIn, and Flickr. We describe the design and security assumptions of each of the main OAuth flows in mobile apps, such as using an embedded web component, native web browser, and using a provider installed app. We demonstrate the attacks that can be performed on the different implementations and discuss the effect of these attacks on the trustworthiness of the OAuth flow. To the best of our knowledge, this is the first study that focuses on investigating the security of the different OAuth implementations in smartphone SDKs. We conducted an empirical study on the current OAuth implementation trends followed by the service providers and by the OAuth development choices made by application developers. We downloaded from the Google Play market more than 430 popular Android applications that are integrated with Facebook and Dropbox services, and we studied the decompiled source code of these applications to summarize the OAuth design choices taken by different developers. Each of these choices is subject to common vulnerability. In addition, we propose an application-based *OAuth Manager* framework, that provides a secure OAuth flow in smartphones that is based on the concept of privilege separation. We summarize our contributions as follows:

- We identifed the different OAuth implementations in smartphones, and summarized the vulnerabilities present in each of the implementations.
- We conducted an empirical study on the OAuth implementations in the SDKs offered by the popular resource providers, and by the app developers.
- We proposed and implemented OAuth Manager.
- We compared our solution with other OAuth implementations in terms of performance and security.

The remainder of this paper is organized as follows. Section II presents the OAuth flows on mobile applications, vulnerabilities, possible attacks and defines the adversary model. Section III shows the analysis results for the SDKs and the collected apps, and Section IV presents our proposed OAuth Manager approach. Section V shows the experimental results. Section VI summarizes the related work. We conclude the paper in Section VII by summarizing our contributions.

II. OAuth and Mobile Applications

Similar to web and desktop applications, native mobile applications (apps) in many usage scenarios require access to user resources hosted on a resource server. Native mobile apps that are installed and executed on mobile devices are considered public clients and utilize special kind of user-agents besides the system browser, called the *WebView*. Thus, OAuth authorization flow steps and the logistics beforehand differ accordingly. For instance, in case of web applications, the authorization server grants them credentials and passwords for the purpose of client authentication. The web application has to keep these information confidential and safe. In case of native applications, the authorization server can't use the same means (passwords and credentials) since these clients aren't capable of keeping such data confidential. Instead, the authorization server require these clients to register redirection URI and/or asking the resource owner to approve identity. During the client registration process, the client developer shall specify the client type, provide its client redirections URIs, and some other information needed by the authorization server.

Several of the OAuth authorization flow steps require a user-agent, which is usually a web browser. On desktops, the web browser's isolation mechanisms, such as the same origin policy, provides the required separation between the user-agent, client and authorization server. The user-agent presents the resource owner with the authentication and authorization information, and the user-agent is used to redirect and pass tokens between the client and the authorization server. In mobile applications, the user agent is implemented using WebView, system browser, and provider app. The details of each approach will be discussed in the following sections.

A. *Embedded Web Browser Component*

The web browser component is a UI view component that can be embedded in a mobile app to display online contents within the hosting app. This component is available in the different mobile frameworks, WebView in Android platform, UIWebView in iOS, and WebBrowser in Windows Phone. The Android WebView uses the WebKit rendering engine to display web pages and includes methods to navigate forward and backward through a history, zoom in and out, and perform text searches. We will focus our discussion on the Android platform, however the following discussion is applicable to other platforms. The WebView is used to perform the role of the user-agent during the OAuth authorization code flow, particularly, it is used to present users with the required OAuth authentication and authorization pages on mobile applications. The client app hosts a WebView as part of its' UI layout. The app can control the embedded WebView, for example it can load a specific url "`webview.loadUrl(URL)`", enable JavaScript in the WebView "`setJavaScriptEnabled(true)`", register event handlers to the WebView to respond to events and to monitor the WebView's activity "`onLoadResource()`" and "`onPageFinished()`", and inject a native (Java) object into the WebView to allow the object's methods to be accessed from JavaScript "`addJavascriptInterface()`".

Figure 1, shows the OAuth flow for client apps that use an embedded web browser (WebView). The client code triggers the WebView to load the service provider's authentication page (Step A0), where the user is promoted to provide her username and password. The WebView also loads the authorization page (Step B), which displays the permissions requested by the application. Upon submission of the authorization page, the authorization server responds and the authentication code is sent as part of the response page title or content (Step C0). The client code has initially registered the event handler "`onPageFinished()`" to be notified when the WebView is done loading pages. The event handler is notified with the loading of the authentication response page, which enables the client code to retrieve the authentication code from the loaded page title or content (Step C1). The client code then uses the authentication code to retrieve the authentication token from the authorization server (Steps D & E).

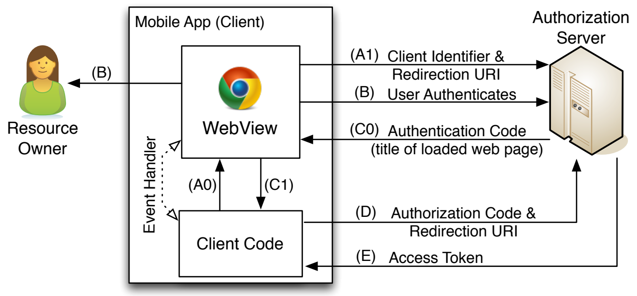

Figure 1. OAuth Embedded Web Client Flow

1) Attacks Against OAuth Embedded Web Browser: Using an embedded web browser as a user-agent in a mobile app, enables the hosting client app to fully control the hosted WebView, which exposes the user-agent and does not provide the required isolation between the client and the user-agent. A malicious hosting app can take control of the hosted WebView and launch attacks on both the user authentication and application authorization. In what follows, we demonstrate both attacks.

Stealing User Credentials. As discussed earlier the hosting app and the embedded WebView can communicate seamlessly. An attack to steal the user credentials (username and password) during the authentication stage can easily be executed by a malicious app in two stages, first injecting JavaScript in the WebView to retrieve the user's email and password upon clicking the submit button, and second registering a JavaScript interface that enables the embedded JavaScript to send the retrieved email and password to the hosting app. Below is the sketch of native and JS code:

```
//Java (Native Client App)
```

```
myWebView.getSettings().setJavaScriptEnabled(true);
myWebView.addJavaScriptInterface(this , "JSInterface");
myWebView.loadUrl("javascript:" + contents of attack.js);

//JavaScript (attack.js)
var submitBtn = document.getElementById('btn_id');
submitBtn.onclick = function(){
 var email = document.getElementById('email_id').value;
 var password = document.getElementById('pwd_id').value;
 JSInterface.jsCall(email, password);
 return true;
}
```

The native app (Java) initializes the required JS interface, then injects the required attack JS. The embedded JS executes in the WebView and registers the required listener to capture the email and password entered by the user. The retrieved information is sent to the native app via the `JSInterface.jsCall()` call. We developed this attack and tested it successfully on all WebView-based SDKs OAuth implementations. The code presented is simplified and the actual code requires using class and type HTML selectors.

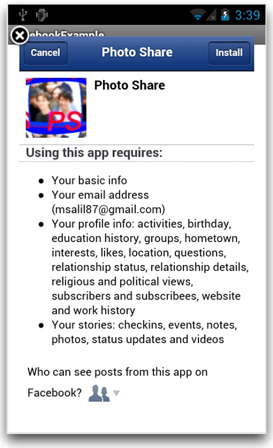

Photo Share

Using this app requires:

- Your basic info
- Your email address (msalil87@gmail.com)
- Your profile info: activities, birthday, education history, groups, hometown, interests, likes, location, questions, relationship status, relationship details, religious and political views, subscribers and subscribees, website and work history
- Your stories: checkins, events, notes, photos, status updates and videos

Who can see posts from this app on Facebook?

(a) Authorization Page

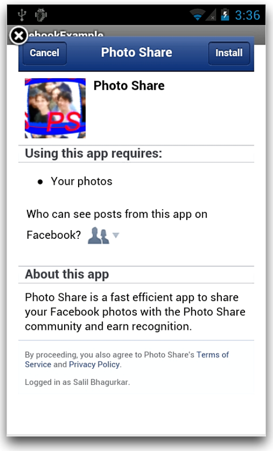

Photo Share

Using this app requires:

- Your photos

Who can see posts from this app on Facebook?

About this app

Photo Share is a fast efficient app to share your Facebook photos with the Photo Share community and earn recognition.

By proceeding, you also agree to Photo Share's Terms of Service and Privacy Policy.

Logged in as Salil Bhagurkar.

(b) Manipulated Page

Figure 2. Manipulating the Authorization Page

Modifying the Authorization Interface. Step B in Figure 1, the WebView presents to the user the authorization page which displays the list of permissions requested by the app being installed. Figure 2(a) shows our photo sharing Facebook application requesting access to the users' protected resources such as email address, profile attributes, checkins, events, notes, photos and many other permissions. Since the authorization page is displayed in a WebView, a malicious client app can easily modify the list of requested permissions displayed in the authorization page. This attack will trick the user into incorrectly interpreting the level of access requested by the app. Below is the JS code that should be injected into the WebView to replace all the requested permissions with simply "Your photos":

```
var permsUL = document.getElementById('perm_ul');
var permsUL.innerHTML = '<li><div>Your photos</div></li>';
```

Figure 2(b), shows the manipulated authorization page. Note that the user authorizing the app in Figure 2(b) is tricked into thinking that the app is authorized to only access her photos, while the app is being authorized for the permissions listed in Figure 2(a).

B. System Native Browser

Several service providers rely on the system native web browser for both the user authentication and the app authorization, where the native browser is used to play the role of the user-agent in the OAuth flow. The native browser is a separate app and is not embedded in the client app, which provides the required isolation between the user-agent (native web browser) and the client app. The native web browser communicates with the client app through the provided mobile framework channels, where most mobile frameworks provide mechanisms that enable the communication and binding between different apps. For example, the Android architecture provides an intent messaging system for run-time binding between components in the same or different apps. The intent holds a description of the operation to be performed, and can contain data to be delivered to the receiver of the intent. The apps should inform the Android system (Intent Manager) about the intents they are willing to receive by registering intent filters. Each intent filter describes the intent of interest, and is associated with a component to respond to that intent. In early system native browser approaches, the user was required to manually copy the access token from the authorization page and paste it in the client application, which was not convenient and was very confusing to users. The current OAuth flow starts when the client app sends an

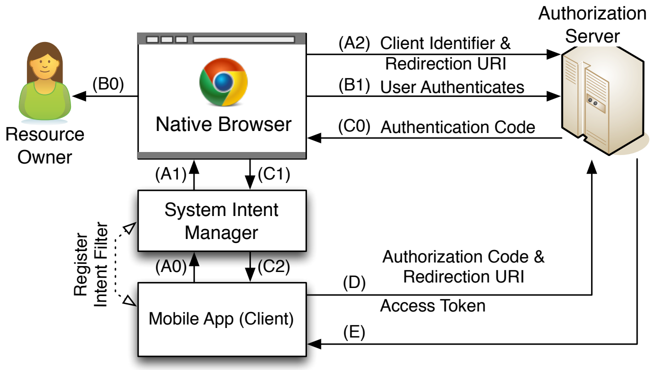

Figure 3. OAuth Native System Browser

intent `(view, http://auth-page, parameters)` to the system intent manager to view the authorization page, see Step A0 in Figure 3. The parameters sent in the intent include url, app id, callback url, and requested access (scope). The intent manager launches the native browser which loads the requested url (Step A1). The user is authenticated, the authorization server identifies the app and displays the authorization page to the user (Step B). If the user agrees to grant the requested permissions, the authorization server redirects the native browser to authorization success page (callback url), which includes the authorization code in the url parameters (C0). To be able to pass the authorization code from the native browser to the client application, the callback url is set to a specific Multipurpose Internet Mail Extensions (MIME) which instantiates an intent to view this special callback url, for example, the url `db-key://connect?oauth_token=xx`, has a special MIME: `db-key`. To be able to receive intents for this special MIME, the client application should register an intent filter of the form `(view, db-key://.)`. Below is the required intent filter to be included in the client application manifest to respond to the special view request.

```
<intent-filter>
  <action android:name="android.intent.action.VIEW"/>
  <category android:name="android.intent.category.DEFAULT"/>
  <category android:name="android.intent.category.BROWSABLE"/>
  <data android:scheme="db-INSERT_APP_KEY"></data>
</intent-filter>
```

The intent manager receives the view intent from the system browser and locates the client application that registered an intent filter to receive this intent (Step C1 and C2). When the client app receives the intent it retrieves the access token embedded in the intent's data. The process is seamless to the user and client app does not communicate directly with the browser, instead all the communication is done through the system intent manager.

1) Attacks against System Native Browser: Compared to the embedded web component, the system browser provides stronger isolation guarantees. However, a malicious app can exploit the channel between the intent manager and the client app. For instance, a malicious app could register similar intent filter as the client app, which could result in passing the access token to the malicious app. If the user has installed both the client app and the malicious app, or if the attacker gains a physical access and does the installation II-D, then the system will present the user with all apps that registered to receive the intent and will ask the user to pick the app that should be used to receive the intent. The user can easily be tricked into selecting the malicious app. Figure 4 shows a demonstration of this attack, The system prompts the user to choose between the two applications (malicious and legitimate). If the user chooses the malicious application, *GT - Document for Dropbox* can steal the access token and impersonate the legitimate app on the resource server.

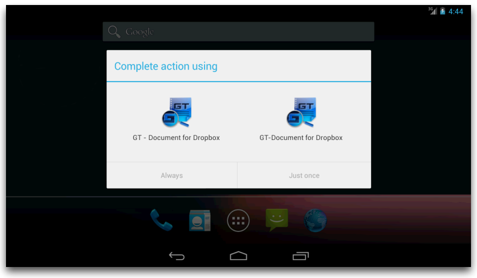

Figure 4. Attack using the System Native Browser

C. *Resource Provider Native App*

This OAuth flow requires an installed resource provider native app as part of the authentication and authorization flow. For example, using the Facebook application as part of the authorization of a client app. In this approach the authentication and authorization processes are performed in the context of the installed resource provider app. This approach is similar to the OAuth flow using the native web browser, while the only difference is that the flow is processed through the installed resource provider app. This approach provides the required separation between the client and the user-agent (resource provider app) and some provider apps maintain a user authentication session, which does not require the user to re-authenticate during the OAuth flow and reduces the user effort. However, this approach requires the user to install the resource provider app for each client that requires to access a specific resource provider. In addition, each of the resource provider apps implements different authorization interfaces that might differ from their web interfaces, which can add to the factors affecting user perception of the security warnings and can lead to users disregarding the authorization prompts [4].

D. *Adversary Model*

In this paper, we assume an attacker is a skillful developer that implements malicious apps and makes them available on the internet. The malicious apps provide a service, like connecting to Facebook, Twitter, etc. The apps contain code segments for collecting users' credentials with the different service providers. The victim (Android user) downloads these apps to enjoy the services they provide. An attacker may also have a physical access to the victim device to manually install malicious apps. The malicious apps take advantage of the vulnerabilities that exist in the providers' SDKs and developers' apps. The attacker needs to have good knowledge of these SDKs and reverse engineering techniques.

III. Apps-SDKs Analysis

In this section, we show the results of analyzing the popular service providers SDKs and collected apps.

Policy. If an SDK or an App support or implement WebView based OAuth flow then they are vulnerable to attacks mentioned in Section II-A1 . Whereas, if they support or implement system browser based OAuth then they are vulnerable to the attack mentioned in Section II-B1. We consider the adversary model defined in Section II-D.

Implementation. Applications for the Android platform are comprised of Dalvik executable (DEX) files that run on Android's Dalvik Virtual Machine. First we downloaded the applications' APK files posted on the Google Play Market, we then extracted the .jar files using the `dex2jar` tool, then we extracted the source files from the jar files using the `jd-gui` decompiler tool. For each app, we analyzed the source code and reviewed the adopted OAuth flow. Similarly,

Platform	Resource Provider SDK	Embedded Web Component	Native Browser	Installable App	OS Integrated
Android	Facebook [5]	√		√	
	Twitter [6]	√			
	Dropbox [7]		√	√	
	Microsfot Live [8]	√			
	Box [9]	√			
	Google Plus [10]				√
	Instagram [11]	√			
	Linkedin [12]		√		
	Flickr [13]		√		
iOS	Facebook [14]		√	√	√
	Twitter [15]				√
	Dropbox [7]		√	√	
	Microsoft Live [16]	√			
	Box [17]	√			
	Google Plus [18]		√	√	
	Instagram [19]	√	√		
	Linkedin [20]	√			
	Flickr [21]		√		

Table I
OAUTH SDKS AND AUTHENTICATION METHODS

we manually investigate the SDKs source code to identify the OAuth flow methods.

We conducted an empirical study on the current OAuth implementation trends followed by the service providers and by the OAuth development choices made by application developers. We reviewed the OAuth based SDKs supported by the popular resource providers such as Facebook, Twitter, Dropbox, Microsoft Live, Google Plus, Box, Instagram, LinkedIn, and Flickr. Table I shows the types of OAuth implementations adopted by the SDKs of the different resource providers for both the Android and iOS frameworks. As can be noted from our results, some SDKs support more than one OAuth flow. For example, the Dropbox SDK for Android provides both native browser and installable provider app based OAuth flows. It is also notable that several SDKs such as Microsoft Live, Box and Instagram, for Android only, support the WebView based OAuth flow. Also for iOS, the Microsoft Live, Box and LinkedIn only support the UIWebView based OAuth flow. This shows that service providers have varying security aware implementations for their OAuth SDKs, some providers are more security aware than others by limiting their implementation to more secure OAuth flows such as native browsers or installable apps.

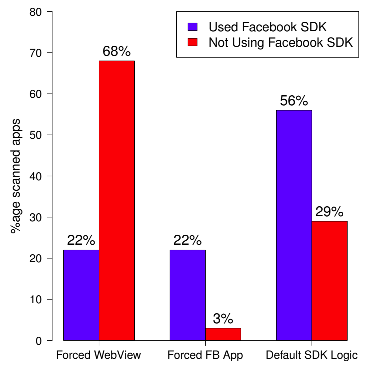

Figure 5. Facebook OAuth Implementations

To study the behavior of SDKs that provide multiple OAuth flow implementations we focus on the Facebook SDK for Android, which allows the OAuth flow to take place either through a WebView or through the Facebook installable app. The default Facebook Android SDK [5] initially queries the system to check if its Facebook app is installed, if it is, the OAuth flow will proceed through the installed Facebook app, otherwise an embedded WebView will be used. The SDK is open source and app developers can adapt its behavior. In order to study how the developers adapt the default Facebook SDK, we analyzed popular applications which are integrated with Facebook and Dropbox services. We analyze 231 Facebook integrated apps, 68% of these apps imported the Facebook SDK and 32% did not use the Facebook SDK and included their own implementations of OAuth. For the apps that used the Facebook SDK we found that 56% of the developers imported the SDK without any changes, 22% of the apps chose to force the WebView based flow (OAuth Dialog), and 22% requested authentication using the installed Facebook App. On the other hand, for the apps that did not use the Facebook SDK we found that 68% used the WebView based flow, only 3% requested authentication using the installed Facebook App, and 29% implemented a flow similar to the default Facebook SDK flow. Figure 5, summarizes the statistics collected for the Facebook integrated apps.

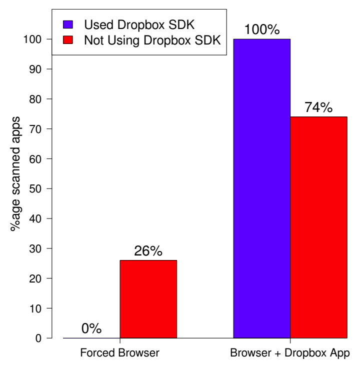

Figure 6. Dropbox OAuth Implementations

We also found that, only 3% of all apps prompted the users to install the Facebook App from the market upon the first run. Moreover, we analyze 202 Dropbox integrated apps, 50% of these apps imported the Dropbox SDK and the other 50% did not use the Dropbox SDK and included their own implementations of OAuth. All the apps that used the Dropbox SDK did not change its default behavior and adopted both the native browser and Dropbox app flows. On the other hand, for apps that did not use the Dropbox SDK we found that 26% used the native browser flow, and 74% implemented a flow similar to the default Dropbox SDK flow. Figure 6, summarize the statistics collected for the Dropbox integrated apps. We also found that, only 0.5% of all apps prompted the users to install the Dropbox app from the market upon the first run.

Summary. Our study shows that 47% of collected Facebook apps are vulnerable to the attacks mentioned in Section II-A1. Moreover, up to 87% of collected Dropbox Apps are vulnerable to the attack mentioned in Section II-B1.

IV. Proposed Approach

We propose to use the privilege separation [22] concept to ensure that the client application has no control over the user-agent. We removed the critical OAuth components and implemented it in a separate application (secure sandbox), which we refer to as the *OAuth Manager*. While many aspects of the proposed solution is applicable for other smart phone platforms (iOS, and Windows Phone), we focus our discussion on Android platform. The user-agent used is a WebView embedded in the trusted *OAuth Manager* app, which isolates it from the client application and can be accessed only through the secured channel that is managed by the system (Intent Manager).

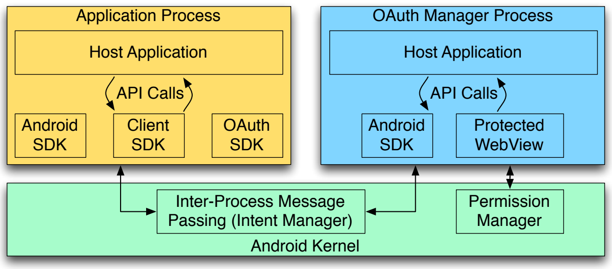

Figure 7. OAuth Manager running on a separate application and communicating with the client apps via the intents

Figure 7, shows the *OAuth Manager* framework. The OAuth Manager is implemented as a separate application, and is responsible for displaying the authorization and authentication in an embedded web component hosted in the OAuth manager and is accessible to the client app. The flow starts when the client application sends an intent to the Intent Manager requesting to start the OAuth Manager application and passes it the required OAuth parameters such as the client app id, secret, and requested permissions (scope). *OAuth Manager* is started and passed the required parameters. The *OAuth Manager* ensures that the client application is a signed application and contacts the systems permission manager to verify that the client application is granted the internet permission `android.permission.INTERNET`, which avoids the privilege escalation scenario. The OAuth flow would terminate if the client application is not granted the internet permission, otherwise the authentication and authorization steps are completed, and the auth code is retrieved in the web component embedded in the *OAuth Manager*. This form of isolation will ensure that the client application is not able to manipulate and control the authentication and authorization pages. Moreover, for the purpose of preventing from the impersonation attack (hijaking the token) *OAuth Manager* includes a routine that works upon running it on the device to the first time and whenever a new app is installed. The routine is meant to verify that no other app is registered to handle intents similar to the one that *OAuth Manger* handles. In the manifest, we statically register a Broadcast receiver that listens to the following system actions: `android.intent.action.PACKAGE_INSTALL` and `android.intent.action.PACKAGE_ADDED`. The system will notify *OAuth Manager* if a new package is installed. In the OnReceive method, an instance of the PackageManager is used to filter the apps that included in their manifests an intent filter for the *OAuth Manger* custom intent. If such apps exist, *OAuth Manger* notifies the user of their existence and ask her to uninstall them. As the final step, the *OAuth Manager* sends the retrieved OAuth access code to the client application through the Intent Manager as a result to the requested intent result. Then the *OAuth Manager* automatically finishes and destroys its process. To the user the execution is transparent and is very similar to the WebView experience. To the developer the SDK can be easily updated to open the *OAuth Manager* instead of opening an embedded WebView, which is a very minor change to the original resource provider SDK.

V. Performance

For the purpose of comparing our proposed *OAuth Manager* approach with the other OAuth flow implementations, we conducted a performance study based on memory consumption and response time. We performed our experiment on a standard Android developer phone, the Nexus S, that has android version 4.1.2, 1007.89 MB internal memory, 13624.34 MB SDCard, 343 MB RAM, system browser 4.1.2-485486. The performance analysis is focused on studying the overhead caused by adopting our method as an authentication and authorization method.

A. *Response time*

We performed benchmarking to estimate the overhead of OAuth manager on displaying the authentication page. It is the required by the OAuth flow implementation to complete the loading of the authentication page after the user authentications. For this purpose, we used Android Logging System, we added hooks to the code to record the time samples immediately after the user clicks the login button, and promptly after successfully loading the authentication page. Our experiment is conducted using the Facebook

OAuth flow. Table II shows the time in milliseconds required by the different OAuth flow implementations. The *OAuth Manager* is faster than the system browser and the embedded WebView. The *OAuth Manager* is slightly slower than the Facebook App, this is because the Facebook App does not use any embedded WebViews and relies on simple api calls. In addition, it is important to note our code was not optimized and was designed to provide OAuth flows for different service provider apps.

Method	Response(milliseconds)
System Browser	3429
Embedded WebView	8077
Facebook App	1879
OAuth Manager	1892

Table II
COMPARISON OF RESPONSE TIME (MILLISECONDS)

B. Memory overhead

We used the Android Debug Bridge (adb) to measure memory overhead. The adb is a versatile command line tool that enables us to communicate with an emulator instance or a connected Android-powered device. In our case, we used Android-powered device, Nexus S. We ran our test application multiple times and each time we used different a authentication method. We recorded the memory consumption for each method. In interpreting our results we are primarily concerned with the proportional set size, which is (Pss) the amount of memory shared with other processes, divided equally among the processes who share it. Table III shows that *OAuth Manager* memory requirements a memory footprint lower than the native browser and the Facebook App. However, the *OAuth Manager* requires more memory than the embedded WebView, this is due to the required security checks. In addition, the *OAuth Manager* code was not optimized for memory usage. The embedded WebView has shown to be insecure and has the possibility of leaking users' sensitive data. In contrast, *OAuth Manager* offers a secure solution and protects the content from being stolen or manipulated.

C. Security Analysis

The OAuth flow based on *OAuth Manager* is more secure than the other flows: II-B and II-A. *OAuth Manager* flow is safe and provides the measures to prevent from the three attacks: the two in Section II-A1, and the attack in Section II-B1. By taking the WebView component out and hosting it in a separate isolated process, it removes the vulnerabilities that are caused by WebView based attacks. Moreover, it uses explicit intents plus a routine that is called upon installation and upon installing any new App to avoid other apps from listening to the same intent and thereby avoiding attacks discussed in Section II-B1.

VI. RELATED WORK

Prior work on OAuth client-flow security by Sun and Beznosov [23] concluded that the client-flow is inherently insecure and warned from putting OAuth 2.0 at the hand of developers. R. Paul [24] highlighted some of the security flaws in Twitter OAuth implementations that enables an attack on client credentials in desktop applications. A similar OAuth flaw was detected by Pai et al. using formal verification of the OAuth protocol. Formal verification was also used by Wang et al. [25] on some service providers' SDKs, their findings showed that apps constructed by importing these SDKs were found to be vulnerable to serious exploits. Their work distinguished five types of secrets in the studied SDKs: access_tokens, codes, refresh tokens, app secrets and session IDs. In our work we focus on the possibility of compromising resource owners' credentials by apps developers. McGloin and Hunt [26] defined the threat caused by misusing an embedded browser in the end-user authorization process, or by presenting its own user-interface instead of allowing trusted browser to render the authorization user interface. As a result, the user would not be aware and all information in the authorization exchange could be captured such as username and password. As a recommendation, they suggested to educate developers and end-users to trust an external System-Browser only. A. Wulf [27] explained how native mobile applications developers are still able to access (steal) users' password even when using OAuth for login. Prior work on WebView vulnerability has not considered its impact on OAuth. Luo et al. [28] explained the risk of using WebView and its APIs on web's security. They presented some attacks and analyzed their fundamental causes, however, they didn't offer any solution. TaintDroid [29] and PiOS [30] study information flow on the two most dominant smartphone platforms: Google's Android and Apple's iOS. Social networking applications were among their list, apps that violate users' privacy. The method of isolating the privileged parts of mobile applications has been used in the mobile advertisement arena in [31], [32] to enable applications to show advertisements without requesting privacy-sensitive permissions. Leontiadis et al. [33] used separate applications solution but also utilized in-application widgets and IPC.

Method	Memory (kB)
System Browser	41386
Embedded WebView	5525
Facebook App	22114
OAuth Manager	13518

Table III
COMPARISON OF MEMORY CONSUMPTION (KB)

VII. CONCLUSION

The different OAuth implementations adopted by popular mobile SDKs vary in their security assumptions and guarantees. In this paper, we described the design and security assumptions of each of the main OAuth flows in mobile apps, such as using an embedded web component, native web browser, and using a provider installed app. We demonstrated the attacks that can be performed on the different implementations and discussed the effect of these attacks on the trustworthiness of the OAuth flow. We analyzed the SDKs provided by major resource providers. We conducted an empirical study on the current OAuth implementation trends followed by the service providers and by the OAuth development choices made by application developers. We downloaded from the Google Play market more than 430 popular Android applications that are integrated with Facebook and Dropbox services, and we studied the decompiled source code of these applications to summarize the OAuth design choices taken by different developers. We compared the implementations of the developers who adopted the SDKs and the developers that built their own implementations. We proposed an application-based *OAuth Manager* framework, that provides a secure, light, and fast OAuth flow for mobile applications that is based on the concept of privilege separation. For developers, we urge them to consider users' privacy when implementing OAuth by: avoid using WebView-based OAuth, prompt the user to install service providers native app, and if the web browser is the only choice left then make sure to add a routine to detect malicious apps that may hijack the auth token.

VIII. Acknowledgments

This research was partially supported by grants from the National Science Foundation (NSF-CNS-0831360, NSF-CNS-1117411) and a Google Research Award.

References

[1] Google, "Google's AuthSub authentication." http://code.google.com/apis/accounts/docs/AuthSub.html, 2008.

[2] Microsoft, "Microsoft Live Connect." http://msdn.microsoft.com/en-us/windowslive/default.aspx, 2010.

[3] Yahoo Inc., "Yahoo Browser-based Authentication." http://developer.yahoo.com/auth, 2008.

[4] A. Mylonas and A. Kastania, "Delegate the smartphone user? Security awareness in smartphone platforms," http://www.sciencedirect.com/science/article/pii/S0167404812001733, pp. 47–66, 2013.

[5] Facebook, "Facebood SDK for android." https://github.com/facebook/facebook-android-sdk, 2012.

[6] Open Source, "Unofficial Java library for the Twitter API (version 3.0.4-SNAPSHOT)," https://github.com/yusuke/twitter4j/, 2013.

[7] Dropbox, "Core API Development kits and documentation (version 1.5.4)," https://www.dropbox.com/developers/core/sdk, 2013.

[8] Microsoft, "The Live SDK for Android library (version 5.0)," https://github.com/liveservices/LiveSDK-for-Android, 2013.

[9] Box, "Box SDK for Android (version 2.0)," https://github.com/box/box-android-sdk, 2013.

[10] Google, "Google+ Platform for Android (version 1.3.0)," https://developers.google.com/+/mobile/android/getting-started, 2013.

[11] Instgram, "Instagram client for Android (version 1.86)," https://github.com/markchang/android-instagram/, 2013.

[12] Linkedin, "A java wrapper for linkedin API (version 1.0.429)," http://code.google.com/p/linkedin-j/, 2013.

[13] Open Source, "A java flickr API library (version 2.0.0)," http://code.google.com/p/flickrj-android/wiki/HowToGuide4Android/, 2013.

[14] Facebook, "The Facebook SDK for iOS (version 3.5.1)," https://developers.facebook.com/ios/, 2013.

[15] Twitter-iOS, "Twitter iOS Integration (version 1.1)," https://dev.twitter.com/docs/ios/, 2013.

[16] Microsoft, "The Live SDK for iOS library (version 5.0)," https://github.com/liveservices/LiveSDK-for-ios, 2013.

[17] Box, "Box SDK for iOS (version 1.0)," https://github.com/box/box-ios-sdk, 2013.

[18] Google, "Google+ Platform for iOS (version 1.3.0)," https://developers.google.com/+/mobile/ios/, 2013.

[19] Instagram, "Instagram iOS Authentication (version 1)," http://instagram.com/developer/authentication/#/, 2013.

[20] Open Source, "API Kits (version 1.1)," http://www.whitneyland.com/2011/03/iphone-oauth.html/, 2013.

[21] ——, "A java flickr API library (version 2.0)," https://github.com/lukhnos/objectiveflickr/, 2013.

[22] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003.

[23] 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*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 378–390.

[24] R. Paul, "Compromising Twitter's OAuth security system," http://arstechnica.com/security/2010/09/twitter-a-case-study-on-how-to-do-oauth-wrong/.

[25] R. Wang, Y. Zhou,S. Chen, S. Qadeer, D. Evans, Y. Gurevich, "Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization ," http://research.microsoft.com/pubs/188979/ExplicatingSDKs-TR.pdf/, 2013.

[26] M. McGloin and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations," http://tools.ietf.org/id/draft-ietf-oauth-v2-threatmodel-00.txt/, 2011.

[27] A. Wulf, "Stealing Passwords is Easy in Native Mobile Apps Despite OAuth," http://goo.gl/QskLq, accessed: 03/12/2013.

[28] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on webview in the android system," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11. New York, NY, USA: ACM, 2011.

[29] W. Enck and P. Gilbert and B. Chun and L. Cox and J. Jung and P. McDeniel and A. Sheth , "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," 2010, pp. 1–6.

[30] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting Privacy Leaks in iOS Applications," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2011.

[31] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: separating smartphone advertising from applications," in *Proceedings of the 21st USENIX conference on Security symposium*, ser. Security'12, Berkeley, CA, USA, 2012.

[32] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "AdDroid: Privilege separation for applications and advertisers in Android," in *Proceedings of AsiaCCS*, May 2012.

[33] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo, "Don'T Kill My Ads!: Balancing Privacy in an Ad-supported Mobile Application Market," in *Proceedings of the Workshop on Mobile Computing Systems & Applications*, 2012.