Remote Exploitation of the Dropbox SDK for Android

Roee Hay and Or Peles
IBM Security Systems
{roeeh,orpeles}@il.ibm.com

Abstract

In today's world personal data is on the cloud. Services, such as photo-hosting or general-purpose storage , should be accessible not only for the user, but also for apps that need access to the data on the user's behalf. Interoperability has always been challenging in many aspects, including access control. In order to combat the latter, authorization protocols, such as OAuth 1 & 2, can securely grant apps access to personal data found in a target service, without disclosing the user's credentials. In order to ease the development lifecycle, such services oftentimes provide a framework, or an SDK, that apps can utilize in order to communicate with the given service. These frameworks are very appealing for app developers since they abstract away the internals, and give the developers a simple client-side API they can use. From a security perspective, the frameworks themselves provide an extremely attractive attack surface for malicious attacks as an exploit of the framework could potentially affect numerous applications making use of it.

In this paper we present a severe vulnerability (identified as `CVE-2014-8889`) within the Dropbox SDK for Android versions 1.5.4-1.6.1. This vulnerability can expose applications using the Dropbox SDK to severe local and remote attacks. As a Proof of Concept, we developed a remote drive-by attack which works against real world apps, including *Microsoft Office Mobile* and *1Password*. We had responsibly reported the vulnerability to Dropbox which promptly provided a patched SDK (version 1.6.2). Developers are strongly encouraged to download it and update their apps.

1 Introduction

The Dropbox SDK is a library that developers can download and add to their products. This library provides easy access to Dropbox features, such as downloading and uploading files, via a simple set of APIs.

AppBrain provides statistics as to the prevalence of the use of the Dropbox SDK on Android [1]. According to these statistics, 0.31% of all applications use the Dropbox SDK. Of the top 500 apps in the Google Play Store, 1.41% use the Dropbox SDK. Interestingly, 1.32% of total app installations and 3.93% of app installations of the top 500 apps use the Dropbox SDK, respectively.

While it is not a highly prevalent library, some extremely popular Android apps that may hold sensitive information use the Dropbox SDK, including Microsoft Office Mobile with over 10,000,000 downloads[1] and AgileBits 1Password with over 100,000 downloads[2].

The vulnerability that we discovered may affect any Android app that uses the Dropbox SDK versions 1.5.4-1.6.1. We examined 41 apps that use the Dropbox SDK for Android, out of which 31 apps (76%) were vulnerable to our attack (i.e. they used version 1.5.4-1.6.1). It's noteworthy that the rest of the apps were vulnerable to a much simpler attack with the same consequences, but had been fixed by Dropbox with the 1.5.4 version of the SDK which they did not care to upgrade to.

This paper is organized as follows. Section 2 gives a background on Inter-App Communication (IAC) in Android. Section 3 shows how IAC can be exploited in general locally by malware and remotely using drive-by techniques. Section 4 describes how the Dropbox SDK for Android uses OAuth for app authorization. In

[1] https://play.google.com/store/apps/details?id=com.microsoft.office.officehub
[2] https://play.google.com/store/apps/details?id=com.agilebits.onepassword

section 5 we deep-dive into the vulnerability we found within the Dropbox SDK for Android OAuth code. Section 6 presents a real attack, dubbed DroppedIn, that exploits the vulnerability. In section 7, we show that the threat is real by presenting case studies. We end with section 8 that presents a mitigation for the vulnerability.

2 Inter-App Communication (IAC) in Android

Android applications are executed in a sandbox environment. The sandbox ensures data confidentiality and integrity as no application can access sensitive information held by another application without proper privileges. For example, Android's stock browser application holds sensitive information such as cookies, cache and history which shouldn't be accessed by third-party apps. The sandbox relies on several techniques including per-package Linux user-id assignment. Thus, resources, such as files, owned by one app cannot be accessed by default by other apps. While sandboxing is great for security, it may diminish interoperability as apps sometimes would like to talk to each other. Going back to the browser example, the browser would want to invoke the Google Play app when a user browsed to the Google Play website. In order to support this kind of functionality, Android provides high-level Inter-App Communication (IAC) mechanisms. This communication is usually done using special messages called `Intent`s, which hold both the payload and the target application component. `Intent`s can be sent explicitly, where the target application component is specified, or implicitly, where the target is left unspecified and is determined by Android according to other `Intent` parameters such as its *URI scheme*, *action* or *category*.

3 General Exploitation via Inter-App Communication

The attack surface is greatly increased if the attacker can directly invoke application components, controlling the `Intent`'s payload. This is the case with exported application components. Such components can be attacked locally by malware. Activities, Android application components responsible for UI screens, can also be attacked remotely using drive-by exploitation techniques as shown by [2, 3].

In the local attack, illustrated by Figure 3.1, malware invokes the exported target application component with a malicious `Intent` (i.e. one that contains malicious data) by simply calling APIs such as `Context.startActivity(Intent)`.

In the case of remote *drive-by exploitation*, illustrated by Figure 3.2, a user is lured into browsing a malicious website. This site serves a web page that causes the browser to invoke the target activity with the malicious `Intent`.

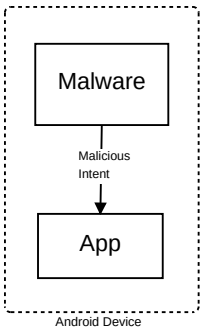

Figure 3.1: Local Attack by Malware

4 OAuth & Dropbox

The Dropbox SDK uses *OAuth* in order to authorize the app on a given Dropbox account. This process begins by an out-of-band registration of the app on the Dropbox website. The app then receives from Dropbox an *app key* and *app secret* which are saved - hard-coded - in the application.

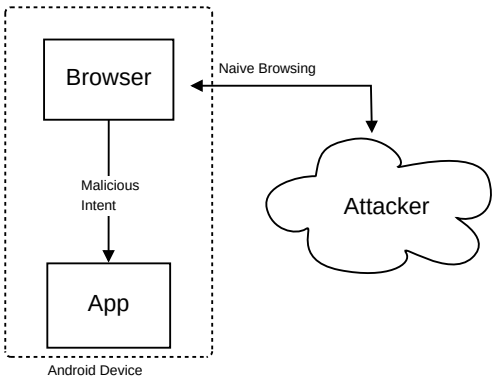

Figure 3.2: Remote Drive-By Attack

The app then exports the Dropbox-provided `AuthActivity` in the *Android Manifest file* as follows:

```
<activity android:name="com.dropbox.client2.android.AuthActivity" ...>
  <intent-filter>
    <data android:scheme="db-<APP_KEY>" />
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.BROWSABLE" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

The players of the Dropbox OAuth dance and their communication channels are depicted by Figure 4.1. The process starts by app code calling the Dropbox's SDK's static method `start{OAuth2}Authentication` of `AndroidAuthSession` with the needed data (i.e. app key and secret). These methods invoke `AuthActivity` using an `Intent`. The `AuthActivity` then generates a *nonce* and either invokes, again by using an `Intent`, the browser or the Dropbox app (if it is installed) in order to authenticate the user and authorize the app. This invocation contains the previously generated *nonce*. The browser or the Dropbox app return the access token, secret and *nonce* with additional data (e.g. uid) back to the app using an implicit Intent targeting the app's unique URI scheme (`db-<APP_KEY>`). This causes `AuthActivity`'s `onNewIntent` method to be called. The latter checks if the incoming *nonce* matches the outgoing one. If it does, it accepts the token and saves it in its `result` static member. The token can then be saved on the Dropbox session for upcoming Dropbox SDK calls.

There are two main threats in this process. The first one is stealing the returned OAuth access token. This would allow the attacker to access the authorized Dropbox resources. This attack could have been done by malware impersonating the App by registering a similar intent filter. However, the Dropbox SDK checks if there is another application with the same intent filter, so this threat is quite mitigated. The other threat is injecting an access token pertaining to the attacker. This will link the App with the attacker's account instead of the victim's, which may then either upload, without consent, sensitive information to the attacker, or download, again unknowingly, malicious data that may enable other attacks. This threat should have been mitigated by the *nonce* parameter (which was added to the SDK in version 1.5.4), however, as we show up next, an implementation-specific vulnerability exists that allows the attacker to actively cause the Dropbox SDK to leak that *nonce* to the attacker's server.

5 Vulnerability

We present a severe vulnerability, identified as `CVE-2014-8889`, that allows the adversary to insert an arbitrary access token into the Dropbox SDK `AuthActivity`, completely bypassing the *nonce* protection.

`AuthActivity` consumes various `Intent` *extra* parameters. Since it is an *exported* and *browsable* activity (and it must be, as per section 4), it can be invoked by both malware and malicious websites with arbitrary `Intent` extras (see section 3). Therefore special care must be taken when consuming these intent extras.

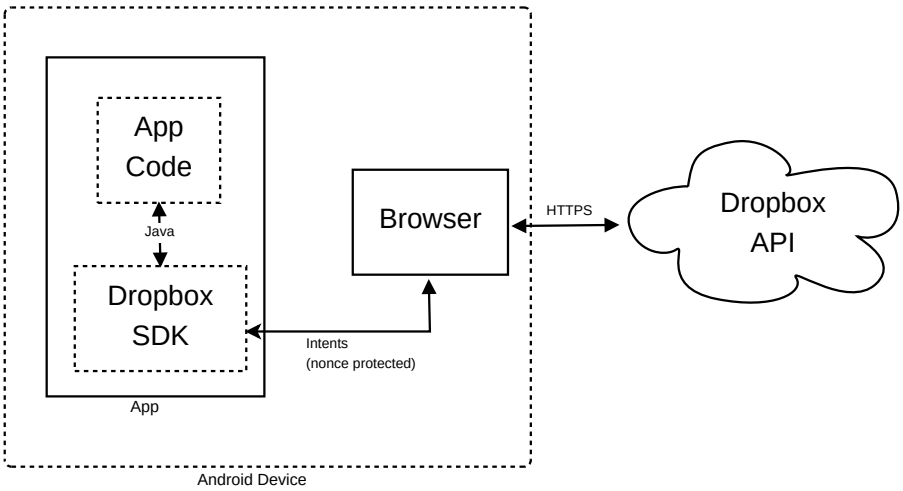

Figure 4.1: Dropbox OAuth Dance

Consumption of a particular Intent extra parameter, named `INTERNAL_WEB_HOST`, has devastating results since it can be controlled by the attacker. When the browser is used to authenticate the user and authorize the app (i.e. when the Dropbox app is not installed), this parameter eventually controls the host that the browser surfs to, as we can see on Appendix A. The method `startWebAuth` is called by the `AuthActivity`'s `onResume` (which is also called after an `Intent` invokes the `Activity`). Thus if the attacker can generate an `Intent` targeting the activity, with `INTERNAL_WEB_HOST` pointing to his server, the authentication process will begin automatically, with the *nonce* being sent to the attacker's server!

6 The DroppedIn Attack

We created both local and remote (drive-by) end-to-end attacks. Both attacks require that the Dropbox app is not installed on the attacked device. Both attacks begin with by the adversary obtaining, out-of-band, an access token (`ACCESS_TOKEN_KEY, ACCESS_TOKEN_SECRET`) and uid pertained to his account and the attacked App. This is specially easy since the attacker can simply download the App to his device, authorize it on his Dropbox account and record the returned access token pair. The following four steps describe the remote attack which is also illustrated by Figure 6.1. The local attack is similar and requires malware running on the device.

6.1 Naive Browsing

In this step the victim naively browses to a malicious website. This can be a website that is completely controlled by the attacker or a website that has been injected with malicious code, using other vulnerabilities, such as Cross-Site Scripting (XSS).

6.2 Active Nonce Leaking

The malicious code automatically causes the victim's browser to invoke the attacked App with an `Intent` that causes it to start the OAuth dance with the adversary instead of `https://www.dropbox.com`. This step allows the attacker to obtain the *nonce*. This invocation can be done by a simple HTTP redirect to

```
Intent:#Intent;scheme=db-<APP_KEY>;
   S.EXTRA_INTERNAL_WEB_HOST=<ATTACKER-SERVER>;
   S.EXTRA_INTERNAL_APP_KEY=foo;
   S.EXTRA_INTERNAL_APP_SECRET=bar;
end
```

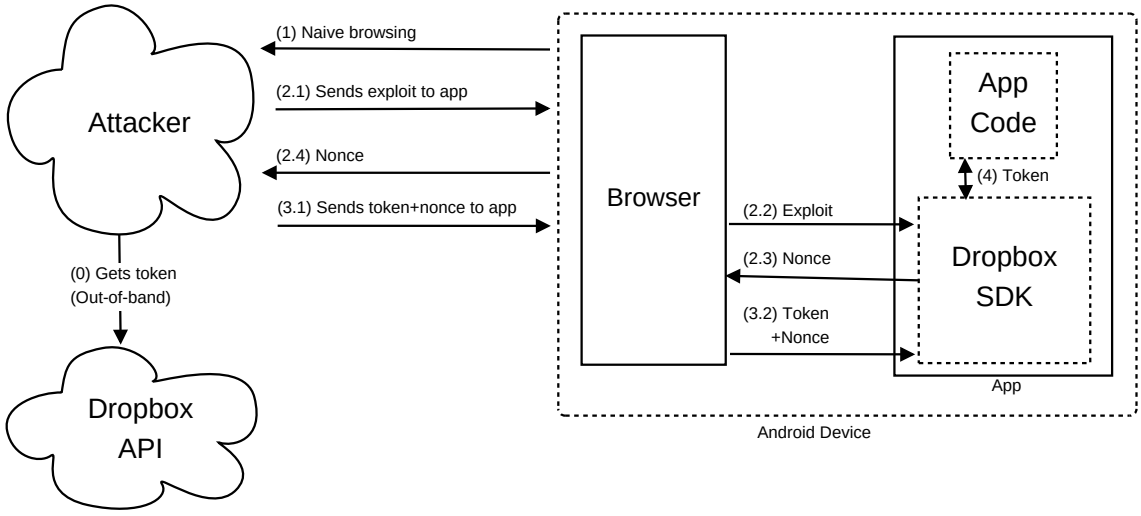

Figure 6.1: The DroppedIn Attack

At time of writing, most of the popular browsers support invoking implicit `Intent`s via the `Intent` URI scheme.

6.3 OAuth Access Token Injection

The above `Intent` will eventually cause the browser to request `https://attacker:443/1/connect` (with the *nonce* as a GET parameter). This implies that succeeding in the last step requires that the attacker owns an SSL certificate of his own domain, but this is rather easy. Learning the *nonce* allows the attacker to simply inject his pre-generated access token into the App by another HTTP redirect to:

```
db-<APP_KEY>://1/connect?oauth_token_secret=<ACCESS_TOKEN_SECRET>
                        &oauth_token=<ACCESS_TOKEN_KEY>
                        &uid=<UID>
                        &state=<NONCE>
```

If the access token injection is successful, it will be saved under `AuthActivity.result`.

6.4 OAuth Access Token Consumption by App

At this point, `AuthActivity`'s `result` static member contains the attacker's token. The missing piece of the puzzle is how the App consumes this data, which is up to the developer's decision.

In general, although other cases may exist, the client-side (App) code will initiate authentication on one of its `Activities`' `onCreate` method, or when clicking on some button. It will then check for successful authentication and move the token to the Dropbox session on its `onResume` method.

The key observation that enables the attack is the fact that `onCreate` and `onResume` are successively called (according to [4]). This implies that if the attacker injects his access token before the App's activity is displayed, the access token will be consumed before the user enters his/her own credentials - see Section 7 for a deep-dive into specific cases.

7 Case Studies

7.1 *Microsoft Office Mobile*: Inject and Add New Link

The *Microsoft Office Mobile* app allows uploading the user's documents to the cloud. This app supports multiple Dropbox accounts.

In this case, the following normal chain of events takes place:

1. User tries to add a Dropbox account. This invokes the activity responsible for the Dropbox authentication.
2. The activity's `onCreate` method will automatically call `startOAuth2Authentication` of the SDK's `AndroidAuthSession`.
3. The activity's `onResume` method is called in a successive manner. It checks if the authentication has succeeded via `AndroidAuthSession`'s `authenticationSuccessful`. The latter returns a negative result.
4. User, via the browser, authenticates on Dropbox and authorizes the app.
5. The activity's `onResume` method is called again, this time `authenticationSuccessful` returns a positive result. The token is copied from `AuthActivity` to the session object and is consumed by the app. The activity is destroyed by calling `Activity.finish`
6. The account is added to the app.

This flow can be attacked as follows - before step 1, the adversary injects his/her token by exploiting the vulnerability. The user then decides to add a new Dropbox account. He will be forwarded to the Dropbox website as usual, however, this time, on step 3 which happens in the background without the user's consent, the `authenticationSuccessful` will succeed. The attacker's token will be moved to the session object and the activity will be destroyed, thus step 5 will not even occur. Therefore, even if the user entered his own credentials, the adversary's token will be used instead, creating a seamless attack.

7.2 *1Password*: Inject Before First Link

The 1Password app falls under the category of password management apps (such as KeyPass), and uses the Dropbox SDK to synchronize the user's vault (key store) with Dropbox. This app supports the use of a single Dropbox account and is used for syncing the local vault with Dropbox. Uploading it to a malicious Dropbox account has devastating results - the attacker can then crack it offline, and if a weak master password is used (which is still a prevalent problem [5]), it can be performed in a feasible time. Alternatively, the attacker can perform a Phishing attack in order to find the master password.

A chain of events similar to 7.1 occurs when the user decides to sync.

1. User clicks on the 'Sync with Dropbox' button. The button invokes the activity responsible for the Dropbox authentication.
2. The activity's `onCreate` method will check if it is linked via `AndroidAuthSession.isLinked()`. If it's not, it will invoke `startAuthentication` of `AndroidAuthSession`.
3. The activity's `onResume` method is called in a successive manner. It will again call `AndroidAuthSession.isLinked()`. If it returns *false*, it will check if the authentication has succeeded via `AndroidAuthSession.authenticationSuccessful`(). The latter returns a negative result.
4. User, via the browser, authenticates on Dropbox and authorizes the app.
5. The activity's `onResume` method is called. Again, `AndroidAuthSession.isLinked()` returns *false*. However, this time `authenticationSuccessful()` returns a positive result. The app then calls `finishAuthentication` which copies the token to the session object (which causes its `isLinked` method to return *true*). The token is then consumed by the app.
6. The sync process begins.

This flow can be attacked as follows - before step 1, the adversary injects his token by exploiting the vulnerability. The user then decides to sync his account. He will be forwarded to the Dropbox website as usual, however, this time, on step 3 which happens in the background without the user's consent, the `authenticationSuccessful` will succeed. The attacker's token will be moved to the session object, thus step 5's `isLinked` call will return *true*. Therefore, even if the user entered his own credentials, the adversary's token will be used instead, creating a seamless attack.

7.3 *DBRoulette*: Inject and Re-link

The *DBRoulette* app is bundled as a sample app together with the Dropbox SDK for Android. It is a basic app that authenticates the user and then loads a random photo from the user's Dropbox's Photos folder. The main activity of this app is `DBRoulette` whose `onResume` method simply overrides the previously stored credentials. This means that even if a Dropbox account has already been linked with *DBRoulette*, the attacker's account will be used instead. In addition, the *DBRoulette* code that calls the authentication methods of the Dropbox SDK resides in an exported activity which means that the attacker can invoke it to complete a fully automatic attack. Figure 7.1 depicts a successful attack, with one of the authors' thumb fetched from the attacker's dropbox account instead of the victim's.

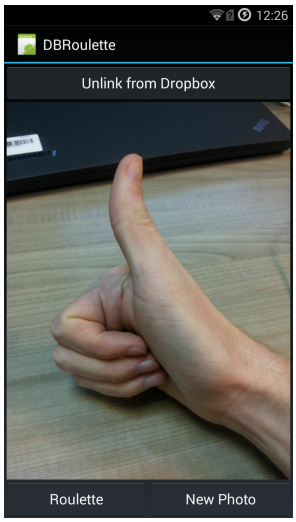

Figure 7.1: Attacked DBRoulette

8 Mitigation

Dropbox SDK for Android version 1.6.2 has been released and incorporates a patch for this security vulnerability. The Dropbox SDK's `AuthActivity` no longer accepts input parameters from the incoming `Intent`'s extras. This makes it impossible for the attacker to control the host the Dropbox SDK communicates with in order to leak the nonce. Developers are strongly encouraged to update their SDK to the latest version. In order to avoid exploitation of slowly-updating apps, end-users (device owners) can install the Dropbox app which makes the attack impossible.

9 Disclosure Timeline

December 1, 2014 - Vulnerabilities disclosed to vendor.
December 2, 2014 - Vendor confirmed the issue, and started working on a patch.
December 5, 2014 - Patch available (Dropbox SDK for Android version 1.6.2).
March 11, 2015 - Public disclosure.

10 Acknowledgments

Dropbox's response to this security threat was particularly noteworthy. We had reported the issue to Dropbox, which acknowledged receipt after a mere 6 minutes. Less than 24 hours after the disclosure Dropbox provided us with a confirmation of the vulnerability and a patch was provided 4 days after the

private disclosure. We would like to thank the Dropbox team for issuing one of quickest patches we have ever witnessed. This with no doubt shows their commitment to the security of their end-users.

References

[1] AppBrain. Dropbox API - Android library statistics. http://www.appbrain.com/stats/libraries/details/dropbox_api/dropbox-api.

[2] Takeshi Terada. Attacking Android browsers via intent scheme URLs. 2014. http://www.mbsd.jp/Whitepaper/IntentScheme.pdf.

[3] Roee Hay & David Kaplan. Remote exploitation of the cordova framework. 2014. http://www.slideshare.net/ibmsecurity/remote-exploitation-of-the-cordova-framework.

[4] Android. Activity. http://developer.android.com/reference/android/app/Activity.html.

[5] Trustwave. 2014 business password analysis, 2014. https://gsr.trustwave.com/topics/business-password-analysis/2014-business-password-analysis/.

A Appendix: Vulnerable Dropbox SDK Code

```
1  protected void onCreate(Bundle savedInstanceState) {
2    ...
3    Intent intent = getIntent();
4    ...
5    webHost = intent.getStringExtra(EXTRA_INTERNAL_WEB_HOST);
6    if (null == webHost) {
7      webHost = DEFAULT_WEB_HOST;
8    }
9    ...
10 }
11
12 protected void onResume() {
13   ...
14   String state = createStateNonce();
15   ...
16   if (hasDropboxApp(officialIntent)) {
17     startActivity(officialIntent);
18   }
19   else {
20     startWebAuth(state);
21   }
22   ...
23   authStateNonce = state;
24 }
25
26 private void startWebAuth(String state)
27 {
28   String path = "/connect";
29   Locale locale = Locale.getDefault();
30
31   String[] params = {
32     "locale", locale.getLanguage()+"_"+locale.getCountry(),
33     "k", appKey,
34     "s", getConsumerSig(),
35     "api", apiType,
36     "state", state};
37   String url = RESTUtility.buildURL(webHost, DropboxAPI.VERSION, path, params);
38   Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse(url));
39   startActivity(intent);
40 }
```