ANDROID COLLAPSES INTO FRAGMENTS

Roee Hay
IBM Security Systems
roeeh@il.ibm.com

***Abstract*—We present a newly found vulnerability in the Android Framework which breaks its sandbox environment. This vulnerability affects many Android apps including ones which are bundled with every Android device. The vulnerability has been patched in Android KitKat.**

I. ANDROID BASICS

A. Threat model

Android applications are executed in a sandbox environment to ensure that no application can access sensitive information held by another without adequate privileges. For example, Android's browser application holds sensitive information such as cookies, cache and history which cannot be accessed by third-party apps. An android app may request specific privileges (permissions) during its installation; if granted by the user, the app's capabilities are extended. Permissions are defined under the application's manifest file (`AndroidManifest.xml`).

B. Activities and Fragments

Android apps are composed of application components of different types including activities. An Activity, implemented by the `android.content.Activity` class [1], defines a single UI, e.g. A browsing window or a preferences screen. Activities can contain fragments (introduced in Android 3.0 [2]). A Fragment, implemented by the `android.app.Fragment` class [3], provides a piece of UI. While activities enable application reuse across the system, fragments provide a greater granularity and enable UI reuse within the same app (see Figure 1).

C. Inter-App Communication and Intents

Android applications make heavy use of Inter-App Communication. This is achieved by Intents. These are messaging objects which contain several attributes such as an action, data, category, target and extras. The data attribute is a URI which identifies the intent (e.g. tel:0422123). Each

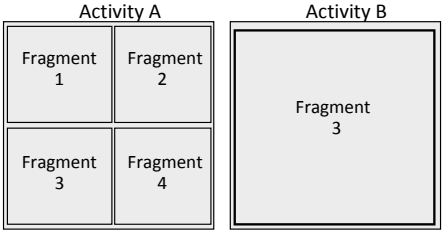

Figure 1. The relation between activities and fragments

Intent can also contain extra data fields (Intent 'extras') which reside inside a bundle (implemented by the `android.os.Bundle` class [4]). These extra fields can be set by using the `android.content.Intent.putExtra` API or by manipulating the extras bundle directly. It is important to emphasize that intents provide a channel for a malicious app to inject malicious data into a target, potentially vulnerable app. Intents can be sent anonymously (implicit intents, i.e. target is not specified) and non-anonymously (explicit intents, target is specified). Intents can be broadcast, passed to the `startActivity` call (when an application starts another activity), or passed to the `startService` call (when an application starts a service). Under the application's manifest file, an application component may claim whether it can be invoked externally using an Intent, and if so which set of permissions is required.

II. PREFERENCE ACTIVITIES AND DYNAMIC FRAGMENT LOADING

The Android Framework provides an abstract activity class, `android.preference.PreferenceActivity` [5] which presents a hierarchy of preferences. An App which wants to show preferences to the user can extend this activity to derive its functionality. The base activity class examines several extra data fields in the input Intent, among them two are interesting: `PreferenceActivity.EXTRA_SHOW_FRAGMENT` (`':android:show_fragment'`) and `Preference-`

`Activity.EXTRA_SHOW_FRAGMENT_ARGUMENTS` (`':android:show_fragment_arguments'`). The first extra field contains a `Fragment` class name and causes a `PreferenceActivity` activity to dynamically display it upon creation. The latter contains the `Fragment` input bundle. A loaded `Fragment` can also receive input arguments by accessing its host activity (and therefore its input `Intent`) using the `Fragment.getActivity` API.

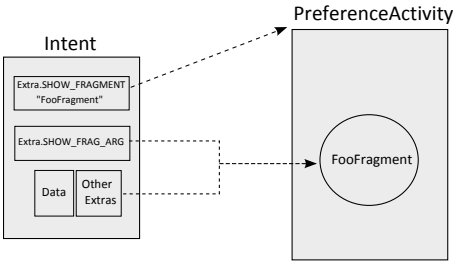

Figure 2. Dynamic Fragment loading

Code under `PreferenceActivity` calls a dynamic `Fragment` loading function, `Fragment.instantiate`. This function loads the `Fragment` using reflection, and then casts it into a `Fragment` object (see Figure 3)

```
577  public static Fragment instantiate
     (Context context, String fname, Bundle args)
     {
578    try {
579      Class<?> clazz = sClassMap.get(fname);
580        if (clazz == null) {
582          clazz = context.getClassLoader().
                     loadClass(fname);
583          sClassMap.put(fname, clazz);
584        }
585        Fragment f = (Fragment)clazz.
                                 newInstance();
586        if (args != null) {
587          args.setClassLoader(f.getClass().
                              getClassLoader());
588          f.mArguments = args;
589        }
590        return f;
591
...
604  }
```

Figure 3. `Fragment.instantiate` (as of Android 4.3.1_r1)

III. Vulnerability

Any app which implements and exports an activity that extends a `PreferenceActivity` class can be subverted to load an arbitrary class by exploiting the dynamic fragment loading process. A malicious app can simply invoke the target activity using an `Intent` object with an `':android:show_fragment'` extra field containing the arbitrary class name, and provide it arguments using the `':android:show_fragment_arguments'` extra or by other intent fields. In the context of `PreferenceActivity`, the class loader which is used is `dalvik.system.PathClassLoader` [6] which enables it to load classes belonging the the vulnerable app, Android or Java frameworks. The loaded class will run in the context of the vulnerable app, will have the same privileges of it and have access to its private data.

IV. Exploitation techniques

A. Actions in constructors

As explained above, the attacker can load any class under the application's package or under the Android/Java framework. Any class which does not extends `Fragment` will cause a `java.lang.CastException` exception (line 585 under `Fragment.instantiate`, see Figure 3) and crash. However, before the casting exception is thrown, two events take place. First, the static initializer of the class is run (if it hasn't run before). Second, its empty constructor is executed. The attacker can abuse this behavior and search for a class which does actions in its constructors. Attractive Java/Android classes would be ones that require privileges that are available to the vulnerable app but not to the malicious app. Application classes provide the same benefit of Java/Android classes (except for the fact that they should be chosen specifically for each vulnerable app), but in addition to that, they are more likely to access sensitive information which is private to the vulnerable app and not otherwise accessible to the attacker. For example, a vulnerable app may have a few exported activities, and some private ones which are only invoked in a particular state (e.g. after login). Normally, the attacker is able to invoke the exported activity classes by intents, but cannot easily invoke the state-dependent classes (accessing them usually requires a user interaction), however by exploiting the vulnerability he is able to instantiate the private activities which possibly perform some actions.

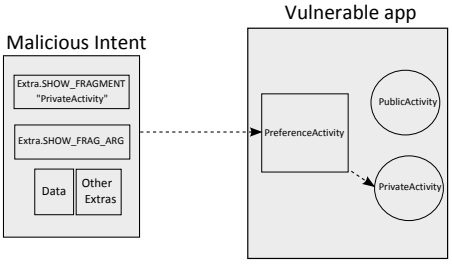

Figure 4. Exploitation by constructors

B. Fragments manipulation

Another opportunity is to find a `Fragment` class (again under the application package or Android/Java frameworks). In contrast to the previous technique, loading a fragment would not cause a `CastException` exception. It allows the attacker to feed the fragment with malicious data by using the input channels described in Section II. Usually the fragment is expected to be loaded by a non-exported (private) activity class thus it will trust the input data and consider them genuine. In addition to the static initializer and default constructor, the fragment lifecycle takes place. This means that methods such as `Fragment.onCreate` will be automatically invoked thus it is more likely that sensitive actions will occur. Even if no sensitive action is performed in automatically invoked methods, the attacker can be the device owner himself or a thief (thus he can control the UI and cause the loaded fragment to perform some action), and use this technique in order to attack system applications and bypass restrictions (see Section V for an example of attacking the Settings app).

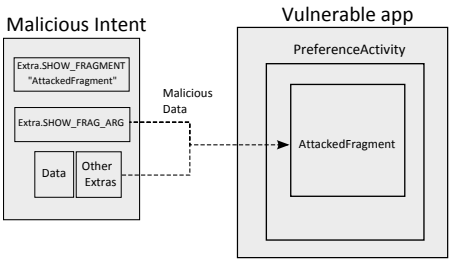

Figure 5. Exploitation by Fragments

V. Real World Example: Android Settings

The Settings app's main activity (which is of course public), `com.android.settings.Settings`, extends `PreferenceActivity`. Therefore it is vulnerable. The package contains many fragments, one of them is `ChooseLockPassword$ChooseLockPasswordFragment`. This fragment is expected to be loaded under the `ChooseLockPassword` class (also extends `PreferenceSettings`) which is not exported according to the manifest file:

```
<activity
  android:name="ChooseLockPassword"
  android:exported="false" ... />
```

The vulnerability allows an external malicious app to load this fragment and provide it data despite its normal instantiation under a non-exported activity. This fragment does indeed consume data from its host activity under its `onCreate` method. For example, it reads the extra value '`confirm_credentials`' which indicates whether the user has entered the correct PIN code. The default value for this attribute is *true* (see Figure 6).

```
213  final boolean confirmCredentials =
         intent.getBooleanExtra("
             confirm_credentials", true);
```

Figure 6. `confirmCredentials` initialization under `ChooseLockPassword$ChooseLockPasswordFragment`

Normally this extra value is provided to the host activity (`ChooseLockPassword`) by `ChooseLockPasswordGeneric` (under the function `updateUnlockMethodAndFinish`) and should be set to *false* if and only if the user has confirmed his password. The conclusion is that the `ChooseLockPassword` activity cannot be invoked with a `confirm_extra` set to *false* unless the user has actually confirmed the credentials (as shown in Figure 7).

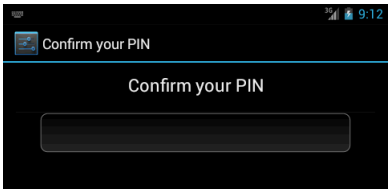

Figure 7. PIN code changing: user must supply old password

A user can bypass this restriction by hosting `ChooseLockPassword$ChooseLockPasswordFragment` inside `Settings` (using the '`:android:show_fragment`' extra parameter, and invoking `Settings` with the '`confirm_credentials`' extra set to `false`, i.e. the second exploitation technique presented above). This allows to user to change the device PIN code without proving that he knows the old one.The attack flow is shown in Figure 9, PoC code provided in figure 10, Result is displayed in Figure 11).

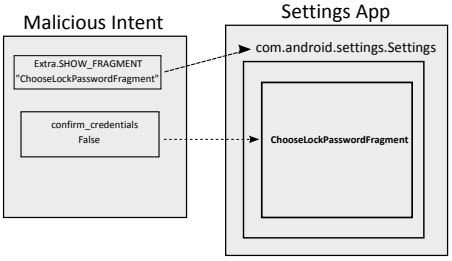

Figure 9. Settings exploit attack flow

```
1176  private void switchToHeaderInner(String fragmentName,
                                       Bundle args,
                                       int direction) {
...
1179      if (!isValidFragment(fragmentName)) {
1180        throw new IllegalArgumentException("Invalid fragment
                                              for this activity: "
1181                  + fragmentName);
1182      }
1183      Fragment f = Fragment.instantiate(this,
                                  fragmentName, args);
...
1888  }
```

Figure 8. Patched Fragment instantiation

```
Intent i = new Intent();

i.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
i.setClassName("com.android.settings", "com.android.settings.Settings");
i.putExtra(":android:show_fragment", "com.android.settings.
    ChooseLockPassword$ChooseLockPasswordFragment");
i.putExtra("confirm_credentials", false);

startActivity(i);
```

Figure 10. Settings exploit code

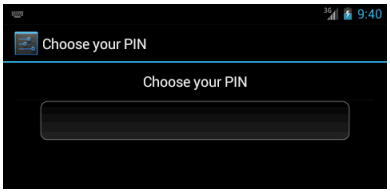

Figure 11. PIN code changing under attack: user does not need to supply old password

VI. The Patch

We reported the security issue to the Android Security Team and a patched `PreferenceActivity` class is provided in Android 4.4 [7]. The patched class contains a new method `protected boolean isValidFragment (String fragmentName).` The new method has been given an appropriate documentation in the Android SDK reference [5]:

> *"Added in API level 19*
> Subclasses should override this method and verify that the given fragment is a valid type to be attached to this activity. The default implementation returns true for apps built for `android:targetSdkVersion` older than `KITKAT`. For later versions, it will throw an exception.
> *Parameters* `fragmentName` the class name of the Fragment about to be attached to this activity.
> *Returns* true if the fragment class name is valid for this Activity and false otherwise which is called before the fragment is instantiated."

The `isValidFragment` method is called before the fragment is instantiated (see Figure 8). It is the responsibility of the developer to override it to provide a white-list of fragments that are allowed to be loaded within a specific activity.

VII. Vulnerable versions

Android 4.3 Jelly Bean [8] and below.

VIII. Non-vulnerable versions

Android 4.4 KitKat [7].

IX. Disclosure Timeline

12/05/2013	Reply from Android Security Team: "Issue is fixed".
12/05/2013	Requested a status update.
11/11/2013	Reply from Android Security Team: "Fix in progress".
10/24/2013	Requested for status update.
07/14/2013	Reply from Android Security Team: "We are now looking into the issue".
07/12/2013	Disclosure to Android Security Team.

X. Appendix: Popular & vulnerable Android Play apps

Vulnerable App	Activities
Google GMail	`GmailPreferenceActivity`
Google Search	`SettingsActivity`
Google Pinyin Input	`AdvancedSettingsActivity`
DropBox	`PrefsActivity`

Evernote	`EvernotePreferenceActivity`
	`AccountInfoPreferenceActivity`
	`SecurityPreferenceActivity`

References

[1] Activity class reference. http://developer.android.com/reference/android/app/Activity.html.

[2] Android 3.0, Honeycomb. http://developer.android.com/about/versions/android-3.0-highlights.html.

[3] Fragment class reference. http://developer.android.com/reference/android/app/Fragment.html.

[4] Bundle class reference. http://developer.android.com/reference/android/os/Bundle.html.

[5] PreferenceActivity class reference. http://developer.android.com/reference/android/preference/PreferenceActivity.html.

[6] PathClassLoader class reference. http://developer.android.com/reference/dalvik/system/PathClassLoader.html.

[7] Android 4.4, KitKat. http://www.android.com/about/kitkat.

[8] Android 4.3, Jelly Bean. http://www.android.com/about/jelly-bean.