

Static and Dynamic Software Security Analysis: Analysing the KoopaApp

Fares KECHID and Mashal ZAINAB

1 TOOLS INVESTIGATION

In terms of the tools that we chose in order to do this project, we selected: Java Soot and FlowDroid. The reason why we selected these two tools, is that they have all the features needed in order to address all the questions of that project, and even in terms of efficiency, it seems that they are always one of the best tools that one may use while working with similar projects. On top of that, the two tools are compatible with each other, in a way that it enhances the developer's quality of experience, and it gives higher performance in terms of programmability.

The Android app components are:

- Activities: They dictate the UI and handle the user interaction to the smartphone screen.
- Services: They handle background processing associated with an application.
- Broadcast Receivers: They handle communication between Android OS and applications.
- Content Providers: They handle data and database management issues.

There are other additional components, which are:

- Fragments: Represents a portion of the user interface in an Activity.
- Views UI: elements that are drawn on-screen including buttons, lists forms etc.
- Layouts View: hierarchies that control screen format and appearance of the views.
- Intents: Messages wiring components together.
- Resources: External elements, such as strings, constants and drawable pictures.
- Manifest: Configuration file for the application.

Some of these components do have a life cycle, such as Activities, Services and Fragments, and some do not have a life cycle or a short one like, Broadcast Receivers, Content Providers and Widgets.

The activity lifecycle in Android is a core set of six callbacks: onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy(). Understanding and implementing these callbacks is important for managing the behavior of the app as users interact with it.

A Fragment has a lifecycle that is similar to an Activity, but it also includes extra events for attaching and detaching from the Activity. Fragments have methods like onAttach(), onCreate(), onCreateView(), onActivityCreated(), onStart(), onResume(), onPause(), onStop(), onDestroyView(), onDestroy(), and onDetach().

2 INSTRUMENTING THE APP

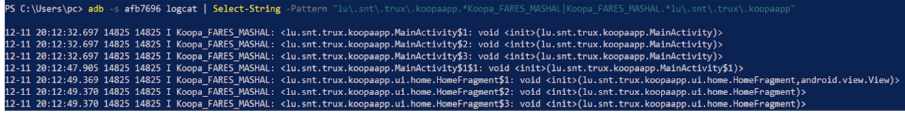
Before starting the instrumentation, we have to understand it:

Android Instrumentation is a set of APIs that are used for testing and controlling an Android application's behavior. It provides hooks into the Android system, which you can use to control aspects of your application's execution. In our case we are going to modify the files of the Koopaapp.apk in such a way that whenever an Activity or a Fragment is executed it logs it.

The source code used to perform this instrumentation using Soot, is sent with this report, under the name of Main.java.

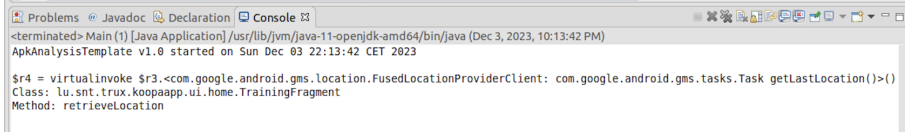
Once the mobile app is instrumented, we need to sign it, so that we should be able to install it later on. The following are the commands that we used to do that:

```
keytool -genkey -v -keystore my-release-key.keystore -alias alias_name -keyalg  
RSA -keysize 2048 -validity 10000
```



```
PS C:\Users\pc> adb -s afb7696 logcat | Select-String -Pattern "lu\.snt\.trux\.koopapp.*Koopa_FARES_MASHAL.*Koopa_FARES_MASHAL.*lu\.snt\.trux\.koopapp.*"
12-11 20:12:32.697 14825 I Koopa_FARES_MASHAL: <lu.snt.trux.koopapp.MainActivity$1: void <init>(lu.snt.trux.koopapp.MainActivity)>
12-11 20:12:32.697 14825 I Koopa_FARES_MASHAL: <lu.snt.trux.koopapp.MainActivity$2: void <init>(lu.snt.trux.koopapp.MainActivity)>
12-11 20:12:32.697 14825 I Koopa_FARES_MASHAL: <lu.snt.trux.koopapp.MainActivity$3: void <init>(lu.snt.trux.koopapp.MainActivity)>
12-11 20:12:47.985 14825 I Koopa_FARES_MASHAL: <lu.snt.trux.koopapp.MainActivity$1$1: void <init>(lu.snt.trux.koopapp.MainActivity$1)>
12-11 20:12:49.369 14825 I Koopa_FARES_MASHAL: <lu.snt.trux.koopapp.ui.home.HomeFragment$1: void <init>(lu.snt.trux.koopapp.ui.home.HomeFragment,android.view.View)>
12-11 20:12:49.370 14825 I Koopa_FARES_MASHAL: <lu.snt.trux.koopapp.ui.home.HomeFragment$2: void <init>(lu.snt.trux.koopapp.ui.home.HomeFragment)>
12-11 20:12:49.370 14825 I Koopa_FARES_MASHAL: <lu.snt.trux.koopapp.ui.home.HomeFragment$3: void <init>(lu.snt.trux.koopapp.ui.home.HomeFragment)>
```

Fig. 1. Result of Logs



```
<terminated> Main (1) [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Dec 3, 2023, 10:13:42 PM)
ApkAnalysisTemplate v1.0 started on Sun Dec 03 22:13:42 CET 2023

$#4 = virtualinvoke $#3.<com.google.android.gms.location.FusedLocationProviderClient: com.google.android.gms.tasks.Task getLastLocation()>()
Class: lu.snt.trux.koopapp.ui.home.TrainingFragment
Method: retrieveLocation
```

Fig. 2. Output of the findLocationMethod()

apksigner sign --ks my-release-key.keystore koopaapp.apk

In order to see the result of this instrumentation, we run the instrumented app on an android device, and we used Android Debug Bridge (adb), with Logcat to see the results:

```
adb -s afb7696 logcat | findstr "Koopa\_FARES\_MASHAL"
```

```
adb -s afb7696 logcat | Select-String -Pattern "lu\.snt\.trux\.koopapp.*
Koopa_FARES_MASHAL|Koopa_FARES_MASHAL.*lu\.snt\.trux\.koopapp"
```

The previous two commands show how to check the log from the powershell, where we see that we can add an argument to the command to search for what we are looking for. Figure 1, represents the a result of running the second command:

While exploring the instrumented app on the device, we notice that the activities and logs appear, for instance MainActivity, HomeActivity, RegistrationActivity and others activities that are APIs related. In terms of Fragments we notice the existence of HomeFragment, TrainingFragment, MyAccountFragment and other Fragments which APIs related, such those used by google APIs.

3 LOCATION SOURCE IDENTIFICATION

In order to identify the method or function that retrieves the device's location, we use the following approach. We write a findLocationMethod() method which iterates over all the all the classes in the Scene, and filters them to find the classes only belonging to the "lu.snt.trux.koopapp" package. Then for each method in these packages, we look for method names that contain the word or substring "location" in their name. If it is present, then it gets the body of that method and iterates over each statement from the body. It checks if the statement contains an invoke statement and if that invoke statement is invoking the "getLastLocation" method which is the way to get device's location in android. Then we print the method name, its corresponding class and the statement which is identified, in this case, the Fragment. The method's name is "retrieveLocation()" and the corresponding fragments is "lu.snt.trux.koopapp.ui.home.TrainingFragment" as shown in figure 2.

Furthermore, to calculate the shortest path from the entry point, the onCreate() method of the MainActivity, to the retrieveLocation() method we use the Breadth First Search (BFS) Algorithm. We set the entry point as the onCreate() method as it is the first method that gets called when an android application starts and the MainActivity is launched. Moreover, we use the BFS algorithm, as it is a graph traversal algorithm used to explore nodes level by level and ensures an efficient exploration of the graph's breath. It can be adapted to find the shortest path between two methods using the control flow graph created with soot and flowdroid. The methods are treated as nodes and

```

Number of nodes in the call graph: 2299176
Time taken: 137.399 seconds.
The shortest path from <lu.snt.trux.koopaapp.MainActivity: void onCreate(android.os.Bundle)> to <lu.snt.trux.koopaapp.ui.home.TrainingFragment: void retrieveLocation()> is:
<lu.snt.trux.koopaapp.MainActivity: void onCreate(android.os.Bundle)>
<androidx.fragment.app.FragmentActivity: void onCreate(android.os.Bundle)>
<androidx.fragment.app.FragmentController: void dispatchCreate()>
<androidx.fragment.app.FragmentManager: void dispatchCreate()>
<androidx.fragment.app.FragmentManager: void dispatchStateChange(int)>
<androidx.fragment.app.FragmentManager: void moveToState(int,boolean)>
<androidx.fragment.app.FragmentManager: void moveToExpectedState()>
<androidx.fragment.app.FragmentManager: void moveToExpectedState()>
<androidx.fragment.app.FragmentManager: void moveToExpectedState()>
<androidx.fragment.app.Fragment: void performCreateView(android.view.LayoutInflater,android.view.ViewGroup,android.os.Bundle)>
<lu.snt.trux.koopaapp.ui.home.TrainingFragment: android.view.View onCreateView(android.view.LayoutInflater,android.view.ViewGroup,android.os.Bundle)>
<lu.snt.trux.koopaapp.ui.home.TrainingFragment: void retrieveLocation()>
The length of the shortest path is: 12

```

Fig. 3. Output of BFS to find the shortest path

edges are based on method calls. We create a CFG using the two and the CHA algorithm and pass it to a method implementing the BFS algorithm called `findShortestPath()` which takes as input the call graph of the apk and the two methods to find the shortest path between. The method uses a queue to perform a systematic exploration, maintains a parent map, and marks the visited methods. In this way it constructs the shortest path when the destination method is reached. And then returns that path. The length of the shortest path in our case is 12, and the path can be seen in the figure 3.

4 DATA LEAK DETECTION

We conduct a taint analysis, which is used to track the flow of sensitive information (taint) through an application, to find data leaks originating from the `retrieveLocation()` method. To do this, we use the flowdroid's `inflow` command line. We provide it with an input `SourcesAndSinks.txt` file that is present in flowdroid by default but we edit the file to have the `retrieveLocation()` method as a source. In terms of a taint analysis, sources are points in the program where sensitive data originates. And sinks are points where sensitive data should not reach without proper validation. We run the following command, generate an xml file about the results and using this we find two leaks originating from the `retrieveLocation()` method. The xml file which was generated is provided with this report, its name is `Leak.xml`. The sink methods corresponding to these leaks are:

- (1) `onCreateView()`
- (2) `onRequestPermissionsResult()`

The command used to find these is:

```

java -jar FlowDroid.jar -a koopaApp.apk -p /home/miccispa/Desktop/Android-
platforms/jars/stub -s SourcesAndSinks.txt -d -cg CHA -t
EasyTaintWrapperSource.txt -o analysis-results.xml

```

5 LLM EXPERIMENTS

For the LLM experiments, i.e., trying to find data leaks from the bytecode using a large language model, we tried two LLMs, Perplexity and ChatGPT. Perplexity can take as input files, so we uploaded the xml file generated from the apk of the koopaApp. It provides us with a very generic result saying, the XML files reveal the Android app's code structure, but without method implementations, identifying data leaks is inconclusive. The app uses Google Protocol Buffers for data serialization, a common practice for communication and storage, yet its usage doesn't necessarily imply data leaks. And for ChatGPT, as it is not possible to upload files through chat, we wrote a python script which uses the openai API to send requests to ChatGPT and data from files can be read through the script as well. For this one, we used dex files. But the api needs a specific api token which has a free limit, so before we can read any notable results, it did not allow us to send anymore requests. However, the python notebook is given with this report.