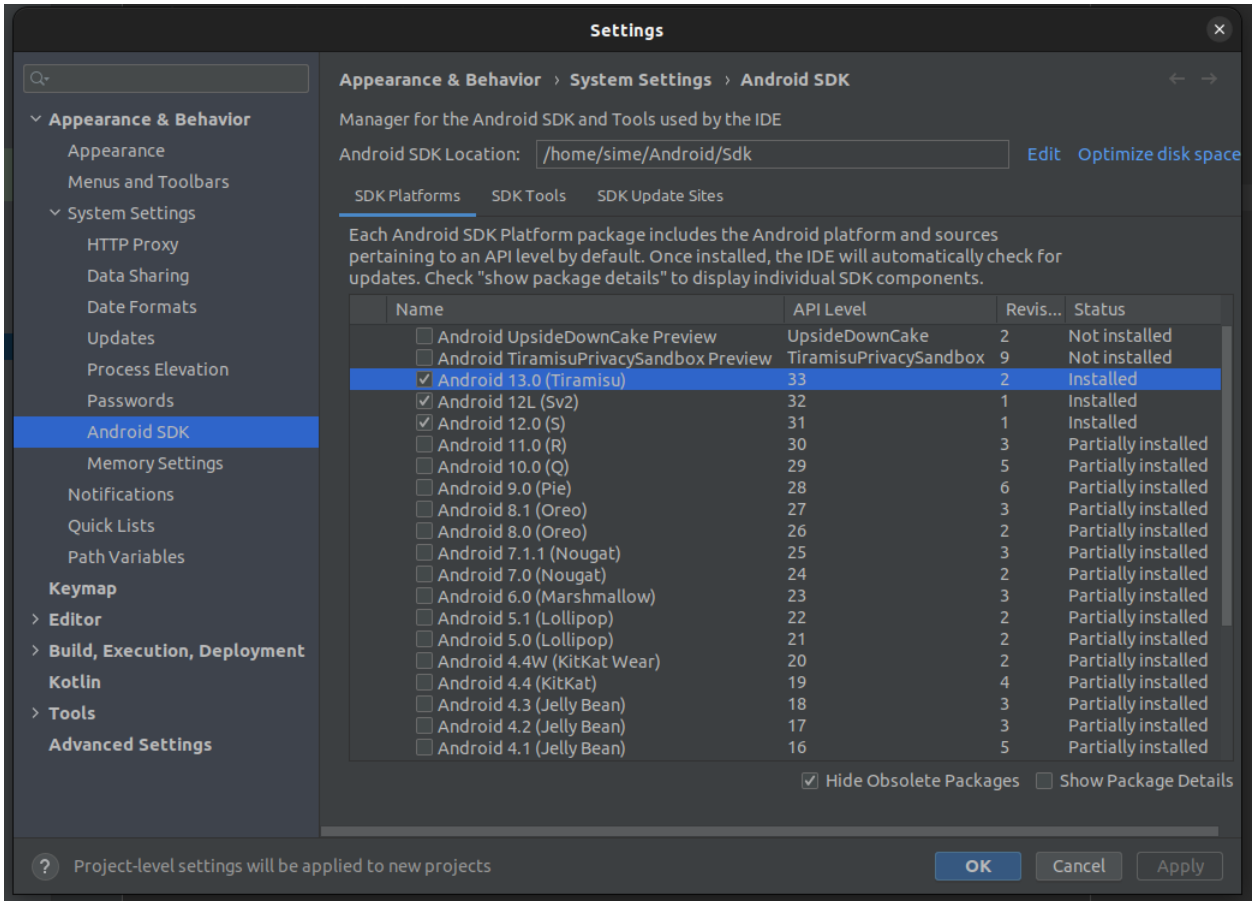
FLOWDROID DEMO

First, let's install flowdroid. We need to install the platforms from the SDK manager of Android studio. We need to use the android.jar file from the same version as the app we are going to analyze, which in this case is android-33.



We also need to download the FlowDroid jar, which can be found at https://repo1.maven.org/maven2/de/fraunhofer/sit/sse/flowdroid/soot-infoflow-cmd/2.11.1/soot-infoflow-cmd-2.11.1-jar-with-dependencies.jar

Finally we need to create the configuration file, and define sources and sinks. However, let's first have a look at the sample application, in order to get an idea of what sources and what sinks we may want to define.

First, we can look at the manifest file of the application. We can see that the app requires the “READ_PHONE_NUMBERS” permission, which is a dangerous permission. We can also see that the app exports two activities, the MainActivity and the ExportedActivity. We can look at these two activities afterwards.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    <uses-permission android:name="android.permission.READ_PHONE_NUMBERS"/>
    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="LeakyApp"
        android:supportsRtl="true"
        android:theme="@style/Theme.LeakyApp"
        tools:targetApi="31">
        <activity
            android:name=".ExportedActivity"
            android:exported="true" >
            <intent-filter>
                <action android:name="ADD_ITEM"/>
                <data android:mimeType="text/plain"/>
            </intent-filter>
        </activity>
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

MainActivity uses the TelephonyManager to get the phone number used by the device. It then puts this number inside an intent as an extra, and uses the intent to send a broadcast. Since the broadcast is implicit, and is sent without any permission, this means that any app can register a broadcast receiver and receive this message, which contains some very sensitive information. Note that to use the TelephonyManager to get the phone number of the device, the application needs to declare the “READ_PHONE_NUMBERS” permission, while apps that receive the

broadcast message do not need to do so. This means that this is a vulnerability that could allow malicious app to bypass permission controls.

```java
package com.example.leakyapp;

import ...

2 usages
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        TelephonyManager tm = getSystemService(TelephonyManager.class);

        @SuppressLint("MissingPermission")
        String number = tm.getLine1Number();
        Intent i = new Intent( action: "BROADCAST_NUMBER");
        i.putExtra( name: "number", number);
        sendBroadcast(i);
    }
}
```

To detect such vulnerability, we can use taint analysis. Thinking about it, the getLine1Number() method could be defined as a source, since it returns the sensitive data, and the sendBroadcast method could be defined as a sink, since it uses the sensitive data and leaks it. This is the configuration file that defines the source and sink. To define a source, we need to write the signature of the method, followed by an arrow and the string _SOURCE_. Similarly, we need to use the signature of the method to define a sink.

```
<android.telephony.TelephonyManager: java.lang.String getLine1Number()> -> _SOURCE_

<android.content.Context: void sendBroadcast(android.content.Intent)> -> _SINK_

```

We can run FlowDroid and verify if it detects the data leak. As you can see, FlowDroid correctly detects that data produced by the getLine1Number source reached the sendBroadcast sink.

```
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - The sink virtualinvoke r0.<com.example.leakyapp.MainActivity: void sendBroadcast(android.content.Intent)>(r2) in method <com.example.leakyapp.MainActivity: void onCreate(android.os.Bundle)> was called with values from the following sources:
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - - $r5 = virtualinvoke r4.<android.telephony.TelephonyManager: java.lang.String getLine1Number()>() in method <com.example.leakyapp.MainActivity: void onCreate(android.os.Bundle)>
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - Data flow solver took 1 seconds. Maximum memory consumption: 634 MB
[main] INFO soot.jimple.infoflow.android.SetupApplication - Found 1 leaks
```

Going back to the application, let's have a look at the other exported activity. In this case, we see that the activity gets two strings from the intent used to launch it, and then uses their value to insert an item inside a Database.

```java
package com.example.leakyapp;

import ...

2 usages
public class ExportedActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_exported);
        Intent i = getIntent();
        try(DBHelper db = new DBHelper( context: this)) {
            String name = i.getStringExtra( name: "name");
            String desc = i.getStringExtra( name: "desc");
            db.addItem(name, desc);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

If we look at the code of the addItem method, we see that the two strings are used inside a SQL query directly, which could result in a SQL injection vulnerability. This situation is different from the previous one, since here we have some untrusted input that reaches a SQL query, However, we can leverage taint analysis to detect this kind of vulnerability as well. In this case, it is the getIntent() method that taints the data, since it returns an intent that could be sent by an external application, and it is the execSQL method that uses the tainted data.

```java
public void addItem(String name, String description) {
    SQLiteDatabase db = getWritableDatabase();
    String query = "INSERT INTO " + TABLE_NAME +" (" + NAME_COL + "," + DESCRIPTION_COL + ")"
            + "VALUES (" + name + ", " + description +")";
    db.execSQL(query);
}
}
```

Defining these two methods respectively as a source and a sink, we obtain this configuration.

```
<android.database.sqlite.SQLiteDatabase: void execSQL(java.lang.String)> -> _SINK_

<android.app.Activity: android.content.Intent getIntent()> -> _SOURCE_

```

We can run FlowDroid with this new configuration to verify if it correctly detects this violation. FlowDroid is able to detect that the data generated by the getIntent() source in the ExportedActivity onCreate method reaches the execSQL sink in the DBHelper.addItem method.

```
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - The sink virtualinvoke $r3.<android.database.sqlite.SQLiteDatabase: void execSQL(java.lang.String)>($r1) in method <com.example.leakyapp.DBHelper: void addItem(java.lang.String,java.lang.String)> was called with values from the following sources:
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - - $r2 = virtualinvoke r0.<com.example.leakyapp.ExportedActivity: android.content.Intent getIntent()>() in method <com.example.leakyapp.ExportedActivity: void onCreate(android.os.Bundle)>
[main] INFO soot.jimple.infoflow.android.SetupApplication$InPlaceInfoflow - Data flow solver took 1 seconds. Maximum memory consumption: 651 MB
[main] INFO soot.jimple.infoflow.android.SetupApplication - Found 1 leaks
```

Note that while in these two examples we only used one source and one sink. There might be multiple sources and multiple sinks: for example, in the first scenario, the sensitive data could be leaked via SMS instead of through a broadcast, so we would need to define additional sinks for methods that can be used to send SMS messages. It could also be leaked by connecting to a remote server, or with BlueTooth, and so on. To detect any possible data leak, we would need to define a list containing many more sinks. Similarly, the phone number is not the only sensitive information that an app could leak, so we would also need to define a list containing many more sources. The sample configurations we used are just simple examples to give you an idea of how sources and sinks are defined in FlowDroid.