Using A Path Matching Algorithm to Detect Inter-Component Leaks in Android Apps

Li Li, Alexandre Bartel, Jacques Klein, Yves le Traon
University of Luxembourg - SnT, Luxembourg
{li.li, alexandre.bartel, jacques.klein, yves.letraon}@uni.lu

I. Introduction

Android has become the most popular mobile phone operating system over the last three years. There are thousands of applications emerging every day. As of May 2013, 48 billion apps have been installed from the Google Play store, and as of September 3, 2013, 1 billion Android devices have been activated[1]. Meanwhile, the Android operating system also becomes a worthwhile target for security and privacy attacks. A major problem in Android is private data leaks. A lot of data leaks have been reported these years, such as passive content leaks [**?**] which cause affected applications to passively disclose in-application data and capability leaks which analyze the reachability of a dangerous permission from a public and unguarded interface.

Many privacy leaks present in Android are the result of interactions among application components which are the basic units to build Android applications. However, on Android no direct code connection exists between two components. To bridge this gap, we present a tool named IccMatcher which uses path matching algorithm to detect inter-component communication (ICC) based leaks. IccMatcher is built on top of Flowdroid [3], a tool performing single component static taint analysis and Epicc [4] , a tool for finding ICC links among components. Both Epicc and Flowdroid leverage the Soot framework [5] which uses the Dexpler plugin [6] to convert Android Dalvik bytecode to Soot's internal representation called *Jimple*.

Flowdroid uses a static taint analysis, a kind of data flow analysis, to keep track values derived from sensitive data. It first labels the private data that we call *source* (for instance a method returning GPS coordinate), and then track the data by statically analyzing the code. If the private data reaches a method that sends it outside the app, also called *sink* method, we identify this as a private data leak and we tag the path from the source to the sink as a detected tainted path.

II. ICC Problem

Some specific Android system methods are used to trigger component communication. We call them ICC methods. The most used ICC method is the `startActivity` method for starting a new Activity.

There are four types of components: a) Activity, representing the user interface; b) Service, executing tasks in background; c) Broadcast Receiver, receiving messages from other components or the system; and d) Content Provider, acting as the standard interface to share structured data between applications. Components use Intent to communicate between one another. All ICC methods take at least one Intent as their parameter. Intents can also encapsulate data and thus transfer data between two components.

Let us consider Listing 1 as an example. Two activities `FirstActivity` and `SecondActivity` are defined and they use the `startActivity` ICC method to communicate. `FirstActivity` contains one *source* method, `getDeviceId`, which returns the unique device ID (e.g., the IMEI for GSM and the MEID or ESN for CSMA phones). We consider the device id as sensitive data. `SecondActivity` contains one *sink* method, `Log.i`, which logs data to disk. Neither `FirstActivity` nor `SecondActivity` contains a tainted path. However, it does exist one data leak from *source* method `getDeviceId` in `FirstActivity` to *sink* method `Log.i` in `SecondActivity`.

```
class FirstActivity {
 void onCreate(Bundle state) {
  String id = telMnger.getDeviceId();
  Intent i = new Intent(this,
    SecondActivity.class);
  i.putExtra("sensitive", id);
  this.startActivity(i); }}
class SecondActivity {
 void onCreate(Bundle state) {
  Intent i = getIntent();
  String s = i.getStringExtra("sensitive");
  Log.i("GRSRD2014", s); }}
```

Listing 1. An example code about crossing component data leaks

Static analyses usually rely on call graphs. However, in Android applications the mechanism of components makes that no direct code connection exists between two components[1]. This means one component cannot be reached from another component in the call graph. We refer to this as the *ICC Problem*.

III. Path Matching Algorithm

In this section, we present a path matching algorithm able to detect ICC based privacy leaks between two components. In order to solve the ICC problem, we define four varieties of sources or sinks:

- *real-sources* are methods that return sensitive data of the application (or Android System).
- *real-sinks* are methods that send at least one sensitive data outside the application.

[1] http://en.wikipedia.org/wiki/Android_(operating_system)

- *bridge-sources* are entry point methods of a component that can be started and receive data from another component.
- *bridge-sinks* are ICC methods which are able to start and send data to another component, e.g., `startService`.

Algorithm 1 presents our matching algorithm to detect ICC based privacy leaks. The inputs are 1) ICC links, a set of ICC links which connect two components, computed by Epicc; 2) IPC (Inter-Procedure Communication) paths, a set of IPC paths which starts with a *source* method and ends with a *sink* method within a component. 3) SourceAndSink containing sets for *real-sources*, *real-sinks*, *bridge-sources* and *bridge-sinks* methods. The algorithm first builds an ICC graph (line 7). Then it checks all the IPC paths to mark the corresponding component node in the ICC graph with *start* if the path ends with a *bridge sink* or *end* if the path starts with a *bridge source* (lines 8-15). Finally, It traverses the marked ICC graph to detect ICC based paths (lines 16-23). For all edges in the ICC graph, if the source node is marked as *start* and the destination node is marked as *end*, an ICC based privacy leak is detected.

Algorithm 1 ICC based privacy leak detection algorithm

```
1: procedure DETECTICCBASEDPRIVACYLEAKS
2:     links ← ICClinks
3:     paths ← IPCpaths
4:     sas ← SourceAndSink
5:     iccPaths ← empthSet
6:     start ← start_marker, end ← end_marker
7:     graph ← buildICCGraph(links)
8:     for path in paths do
9:         if path.first ∈ sas.bridgeSource then
10:            markICCPath(graph, path, end)
11:        end if
12:        if path.last ∈ sas.bridgeSink then
13:            markICCPath(graph, path, start)
14:        end if
15:    end for
16:    for node in graph do
17:        children ← getChildNodes(graph, node)
18:        for childNode in children do
19:            if node.hasStart & childNode.hasEnd then
20:                iccPaths.add(node.start, childNode.end)
21:            end if
22:        end for
23:    end for
24:    return iccPaths
25: end procedure
```

Figure 1 shows the result of applying our path matching algorithm on the code of Listing 1. First, Epicc computes an ICC link from `FirstActivity` to `SecondActivity`. Then, Flowdroid computes an IPC path from `getDeviceId` method to `startActivity` method in `FirstActivity` and a IPC path from `getIntent` method to `Log.i` method in `SecondActivity`. Finally, we run our matching algorithm to build the ICC graph. The left node represents `FirstActivity` and we mark it as *start* since it contains an IPC path ending with a *bridge-sink* method. The right node represents `SecondActivity` and we mark it as *end* since it contains an IPC path starting with a *bridge-source* method. The two nodes' marker match so there is one ICC based privacy leak between `FirstActivity` and `SecondActivity`.

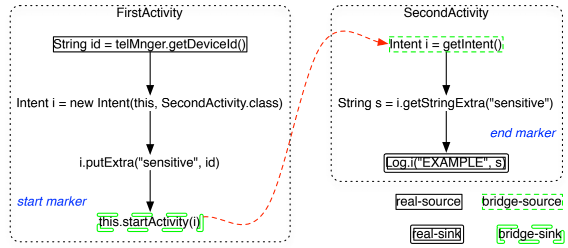

Fig. 1. Marked ICC graph of our motivating example listed in Listing 1

One special ICC method, `startActivityForResult`, exists in Android system. It starts another component and then waits for it to finish. Once the other component finishes, it continues running with the result returned from the other component. Because it has more complicated semantics compared to common ICC methods that only trigger one-way communication between components, we handle them specifically.

IV. Case Study

InsecureBank[2] is a vulnerable Android application created by Paladion Inc. specifically for the purpose of evaluating privacy leak detection tools. Flowdroid finds all seven data leaks (within component) without any false positives nor false negatives. We ran our path matching algorithm based tool IccMatcher on InsecureBank, and we find a privacy leak crossing two components from `com.android.insecurebank.LoginScreen` to `com.android.insecurebank.PostLogin`. In `LoginScreen`, a password is obtained from `EditText` and is stored into an `Intent` which is sent to `PostLogin` by the ICC method `startActivity`. In `PostLogin`, the password is sent outside InsecureBank through the `postHttpContent` method.

Acknowledgement: this work has been done in collaboration with the team of Prof. Eric Bodden (TU Darmstadt) and the team of Prof. Patrick McDaniel (Penn State University).

References

[1] Li Li et al. Detecting privacy leaks in Android Apps. In: International Symposium on Engineering Secure Software and Systems - Doctoral Symposium (ESSoS-DS2014). 2014

[2] Michael Grace et al. Systematic detection of capability leaks in stock Android smartphones. In: Proceedings of the 19th Annual Symposium on Net- work and Distributed System Security. 2012

[3] Steven Arzt et al. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In: the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2014). 2014

[4] Damien Octeau et al. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In: Proceedings of the 22nd USENIX Security Symposium. 2013.

[5] Patrick Lam et al. The Soot framework for Java program analysis: a retrospective. In: Cetus Users and Compiler Infastructure Workshop (CETUS 2011). 2011

[6] Alexandre Bartel et al. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In: ACM Sigplan International Work- shop on the State Of The Art in Java Program Analysis. Beijing, China, 2012

[2] http://www.paladion.net/downloadapp.html