Static Taint Analysis Method for Intent Injection Vulnerability in Android Applications

Bin Xiong[1], Guangli Xiang[1](✉), Tianyu Du[2], Jing (Selena) He[3], and Shouling Ji[2]

[1] College of Computer Science and Technology, Wuhan University of Technology, Wuhan, China
13072799038@163.com, glxiang@whut.edu.cn

[2] College of Computer Science and Technology, Zhejiang University, Hangzhou, China
tydusky@gmail.com, sji@zju.edu.cn

[3] Department of Computer Science, Kennesaw State University, Marietta, GA 30060, USA
jhe4@kennesaw.edu

Abstract. In the component communication of Android application, the risk that Intent can be constructed by attackers may result in malicious component injection. To solve this problem, we develop IntentSoot, a prototype for detecting Intent injection vulnerability in both public components and private components for Android applications based on static taint analysis. It first builds call graph and control flow graph of Android application, and then tracks the taint propagation within a component, between components and during the reflection call to detect the potential Intent injection vulnerability. Experimental results validate the effectiveness of IntentSoot in various kinds of applications.

Keywords: Static taint analysis · Call graph · Control flow graph · Intent injection vulnerability

1 Introduction

With the growing momentum of the Android operating system, thousands of applications (also called apps) emerge every day on the official Android market (Google Play) and other alternative markets. In the second quarter of 2016, Android devices occupied 86.2% of total sales on mobile operating system, and 2600 thousand apps have been installed from Google Play store in December 2016. Due to the fact that many application developers have poor security awareness and application markets do not take comprehensive security testing measures, there are various security vulnerabilities in a large number of applications, which brings security risks to users.

In order to enhance application security, each application is assigned a user ID (UID) and runs in an isolated virtual machine. However, applications need to share data with others in a collaborative environment, so Android provides a flexible communication mechanism named Inter-Process Communication (IPC), which takes place in Binder or Intent [11]. Among them, Intent can transfer messages between the same or different application components as a message container (including action, data, type, extras and other properties), which achieves Inter Component Communication (ICC). A large number of studies have shown that the introduction of ICC mechanism will expose applications to a variety of external stressful conditions, and therefore applications is likely to receive a large number of untrusted data, which causes security vulnerabilities. Intent injection is an ICC application vulnerability. Generally, the application with an intent injection contains a public component. The application can receive a malicious intent message. Intent injection vulnerability is a relatively common ICC vulnerability. Applications with this type of vulnerability typically contain a public component. After the application receives a maliciously constructed Intent message, it does not perform valid security authentication and directly parses out the parameters for some security-sensitive operations, resulting in malicious behaviors such as privilege elevation, information leakage, and even remote code execution.

Related Work. Methods for detecting Intent injection vulnerability can be roughly divided into two kinds, including dynamic analysis and static analysis [14]. Dynamic analysis sends random input to the target application to identify potential vulnerabilities by monitoring software anomalies. JarJarBink modified the ICC parameters according to a custom variation strategy and sent them to the target application component to evaluate the ICC [7]. Intent Fuzzer used a new method combined with static analysis and test case generation [10]. For all the public components of the application, the data structure of Intent is obtained by static analysis, and then the corresponding fields are randomly changed. Compared with JarJarBinks, Intent Fuzzer can detect exceptions from deeper logic of code segments. However, it is difficult for the Fuzzing method to find available vulnerabilities along with a lot of empty pointer exceptions. Without running the practical application, static analysis directly analyzes Dalvik bytecode of the application and extracts potential malicious behaviors. ComDroid [1] used static analysis to detect ICC problems, but they believed that as long as the component is exposed and there is no privilege protection, there are Intent injection attacks. This coarse-grained analysis method will lead to a quantity of false positives. CHEX found data flow between application components by connecting all the code segments from the entry point to detect component hijacking vulnerabilities [6]. However, due to the absence of data flow analysis between components, they cannot detect the injection of private components, leading to false negatives. Epicc [8] transformed the ICC issue to an IDE problem [9], but there are still a lot of false positives. At the Blackhat conference in 2014, Daniele proposed a static detection approach for Intent message vulnerabilities in Android applications [7], but it only detects vulnerabilities in a single Activity component, which exists false negatives. Recently, IccTA proposed a static detection method for ICC problems [4,5], which can be modified to directly connect components but cannot detect the vulnerability during reflection calls. Since component exposure does not imply the presence of Intent injection and Intent injection exists not only in public components, the above-mentioned studies is common in false positives or false negatives and cannot effectively analyze the dynamic loading as well as reflection mechanism in Android.

Our Contribution. In this paper, we present IntentSoot, a prototype to effectively detect the Intent injection vulnerability in Android applications based on static taint analysis. By defining the Intent message as a taint source and tracking the propagation of taint data within a component, IntentSoot can effectively reduce the number of false positives and false negatives between components and during reflection calls.

Roadmap. The rest of this paper is organized as follows. Section 2 introduces the communication process security analysis in Android applications. Section 3 presents the overall structure of IntentSoot. Section 4 details the principles of IntentSoot. Section 5 shows the datasets and experimental results. Section 6 concludes the paper.

2 Communication Process Security Analysis in Android Application

2.1 Component Communication Mechanism

Android applications have four types of components, including Activity, Service, BroadcastReceiver and ContentProvider. Except ContentProvider, the communication between the other three components requires an Intent message. Intent is a media for ICC process to convey messages. Intent contains both the explicit and the implicit, among which explicit Intent specifies the receiving component by name, so the Intent is sent to a particular application component. However, implicit Intent's receiver can be all components that meet the conditions, and the Android system determines which application can receive the Intent. Intent-filter tag can be added to a component in AndroidManifest file, thus receiving different Intent by defining action, category and data. In addition, all components that have the intent-filter tag and do not have the attribute "exported" valued false are public by default, and other applications can send Intent messages to public components.

2.2 Intent Injection Vulnerability in Application Communication Process

The Intent injection vulnerability was proposed in literature [2] at the first time, and an application with such vulnerability typically has a public component that can receive external Intent messages. The logic of internal processing in application directly uses the parameter information parsed from the Intent message for some security-sensitive operations without validation. If the Intent message is carefully constructed by the attacker, it will cause Intent injection. The threat model of Intent injection contains two roles, including attackers and victims, as shown in Fig. 1.

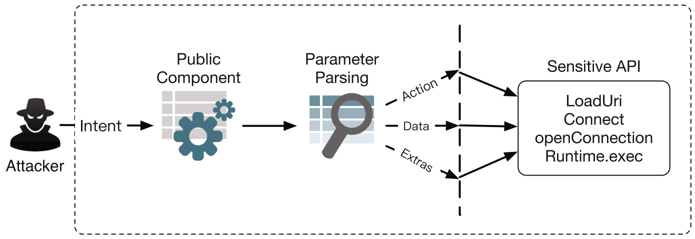

Fig. 1. Threat model for Intent injection attack

2.3 Intent Injection Example

Intent injection vulnerability means that, the taint data may not only flow within a component and between components in the process of taint data propagation, but also flow during the refection calls, as shown in Listing 1.1. Assuming that doSomethingBad1 and doSomethingBad2 are two security-sensitive API functions, then there are two Intent injection vulnerabilities in this example. The first vulnerability exists in the public component Activity01, where the parameter s1 is passed directly to the method doSomethingBad1 without any security check. The second vulnerability exists in the private component Activity02, where the parameter s4 is passed to the method sendMSG in the class SendMessage, and then passed to the method doSomethingBad2 without any security check. The Next-Intent attack described in [13] utilized the Dropbox application's private components. The presence of the Intent injection vulnerability in the VideoPlayerActivity caused the leak of token information of a large quantity of Drophox accounts. However, previous studies [1,3,6,8] were unable to detect Intent injection vulnerabilities in private components. Therefore, the purpose of this paper is to develop a static analysis tool which can detect Intent injection vulnerability in both public components and private components for Android applications through the static taint analysis method.

3 Overall Design for IntentSoot

This section systematically introduces our tool named IntentSoot, whose overall structure and flow chart is shown in Fig. 2. We make some definitions first to facilitate further description.

– **Source method:** The method where the caller is located, is represented by Sm. As shown in Listing 1.1, the onCreate method in Activity02 is a source method.
– **Source class:** The class where the source function is located, is represented by Sc.
– **Target method:** The called method, is represented by Tm. As shown in Listing 1.1, the sendMSG method in SendMessage is a target method.

```
 1 //Activity01
 2 onCreate(Bunlde ...){...
 3   Intent i1 = this.getIntent();
 4   String s1 = i1.getData().getQueryParameter("key1"); //source
 5   String s2 = i1.getStringExtra("key2"); //source
 6   ...
 7   doSomethingWith(s1);
 8   Intent i2 = new Intent();
 9   i2.setClass(..., BroadcastReceiver01.class);
10   i2.putExtra("key2", s2);
11   sendBroadcast(i2);
12 }
13 Public void doSomethingWith(String value){
14   doSomethingBad1(value);//sink
15 }
16 //BroadcastReceiver01
17 OnReceive(Context c, Intent i3){...
18   Intent i4 = new Intent(c, Activity02.class);
19   String s3 = i3.getStringExtra("key2");
20   I4.putExtra("key3",s3);
21   C.startActivity(i4);
22 }
23 //Activity02
24 onCreate(Bunlde){...
25   Intent i5 = getIntent();
26   String s4 = i5.getStringExtra("key3");
27   File file = new File("/adcard/DynamicLoadClient.apk");
28   if (file.exist()){
29     DexClassLoader cl = new DexClassLoader(file.toString(), ...);
30     try{
31         Class<?> myClass = cl.loadClass("com.dynamicloadclient.SendMessage");
32         Constructor<?> constructor = myClass.getConstructor(new Class[]{String.class});
33         Object object = constructor.newInstance(new Object[]{"MessageMark"});
34         Method action = myClass.getMethod("SendMSG", new Class[]{String.class});
35         action.invoke(object, s4);
36     } catch (Exception e) {
37         e.printStackTrace();
38     }
39   }
40 }
41 //SendMessage
42 public void sendMSG(String content){
43   doSomethingBad2(content);//sink
44 }
```

Listing 1.1. The Intent injection example for apps without reflection

– **Target class:** The class where the called function is located, is represented by Tc. As shown in Listing 1.1, the SendMessage class is a target class.
– **Reflection method:** The method used by the caller to implement the reflection mechanism, including the newInstance and invoke methods, is represented by Rm. As shown in Listing 1.1, the newInstance method in line 33 and the invoke method in line 35 are reflection methods.

IntentSoot mainly consists of two parts: dynamic execution module (IntentSoot-D) and static analysis module(IntentSoot-S).

IntenSoot first runs IntentSoot-D, installing the app into the Android device. After installing the app, IntentSoot runs the app file and cyclically reads the system's log output to get information about the dynamic loading and reflection calls. When capturing the dynamic loading behaviors, IntentSoot sends an adb downloading command to the Android device, and then downloads the loaded

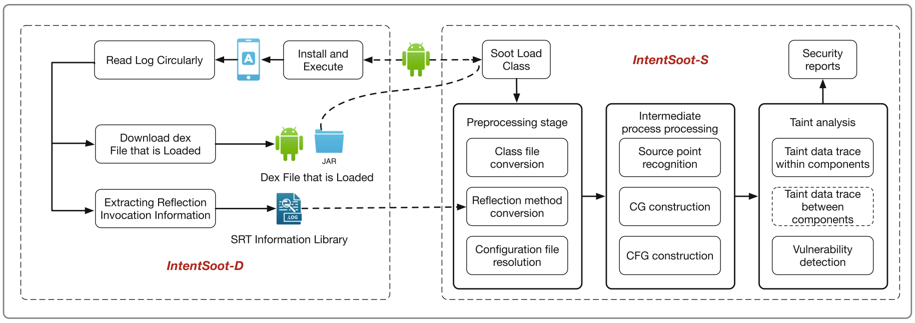

Fig. 2. Overall structure and flow chart of IntentSoot

dex file to the local computer. When capturing the reflection call behaviors, IntentSoot extracts the information such as Sm, Rm and Tm corresponding to the reflection call from the log, and stores the information in the form of a triple group of <Sm, Rm, Tm> in the local computer's file called SRT library. The downloaded dex file and SRT library will be used for subsequent static taint analysis.

When the dynamic execution module is finished, IntentSoot runs IntentSoot-S. IntentSoot-S is actually an improvement version of Soot [12]. It makes up Soot's defects by adding the dynamic execution module outputs, namely dex file and the SRT library, to static taint analysis. Meanwhile, the mapping table between ICC method and lifecycle method of target component is supplied to make up the defect that Soot cannot handle component communication. IntentSoot-S consists of three stages, including preprocessing stage, intermediate processing stage and taint analysis stage.

4 Principle Introduction for IntentSoot

4.1 Principle Introduction for IntentSoot-D

IntentSoot-D includes running the app, downloading the dex file to the local computer and collecting a triple group <Sm, Rm, Tm> of reflection call information into the SRT library. The downloaded dex files and the SRT library will be used for subsequent static taint analysis.

4.1.1 The Modifications for Android Source Code

We modify the Android source code according to the literature [15], and the modified Android version is 4.4. The main modifications include:

– **The modification of elements stored in the method call stack.** The method call stack in the source code only stores the class name and method name. In order to get complete information, we modified it to store parameters and return values of the method.

- **Increase the log output for dynamic loading.** Modify the openDexFile method in the DexFile.java file. Therefore, when dynamic loading behavior occurs in the system, the system will output the path of the loaded dex file, so that the loaded dex file can be downloaded by IntentSoot-D in time.
- **Increase the log output of the newInstance method call.** Modify the newInstance method in the Constructor.java file. Therefore, when newInstance method call occurs in the system, it will output class name of the instantiated target class, instantiated method name and instantiated parameters.
- **Increase the log output of the invoke method call.** Modify the invoke method in the Method.java file. Therefore, when invoke method call occurs in the system, it will output class name of the target class, method name and parameters of the target method.

The items 2-4 above corresponds to the log output, which includes not only the descripted log output but also the additional output, including log ID, uid number, operation number and call stack information, etc. The log ID is used to identify which log information need to be parsed by IntentSoot-D. Through the uid number, IntentSoot-D can identify which log output is generated by the application. The operation number is used to identify the type of monitored program behavior. According to the difference of called method, we set the newInstance, invoke, openDexFile method corresponding to the number 1, 2, 3. In addition, the Android system outputs not only the log information of the three items above but also the information of the Android method call stack in the current thread, which will assist the IntentSoot-D to get the information of source method and reflection method. The details will be described in Sect. 4.2.

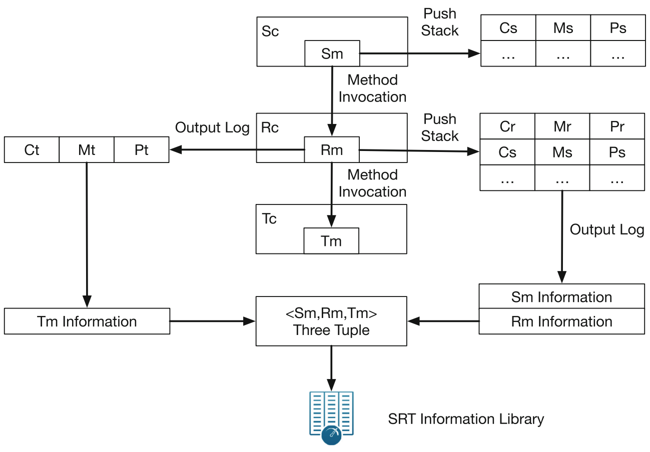

Fig. 3. Schematic diagram of log analysis

4.1.2 Log Analysis

After installing and running the application, IntentSoot can get the app's uid number. The app's running requires actual use, as well as triggering dynamic loading and reflection call as much as possible. IntentSoot-D will cyclically read the phone's log information, extracting the log informations of the app according to the uid. When reading a record of dynamically loading dex files, IntentSoot-D downloads the dex file to the specified folder in the local computer. When capturing the output information of the newInstance method or the invoke method, the IntentSoot extracts the information of the corresponding source method Sm, the reflection method Rm and the target method Tm, which will be stored in <Sm, Rm, Tm>.

The extraction principle of the information is described in Fig. 3. We use Cx, Mx, and Px to represent class name, method name and method parameter name, respectively. X can be a value of s, r or t, representing the source, the reflection and the target. For example, Cs represents the class name of the source class. According to the modification in Sect. 4.1.1, we can get the information of Tm directly from log when newInstance or invoke method is called. Similarly, we can get the information about Sm and Rm through the Android method call stack in current thread. The Android method call stack stores the method call sequence from the current method call in chronological order. The method information currently being called is stored on the top of the stack, and the last method call from the current method call is stored on the second unit. Thus, when a reflection method call occurs, the method at the top of the stack is usually the information of the reflection method Rm, and the second unit in the stack is the information of the source method Sm. According to the modification of the elements stored in the method call stack of Android system in Sect. 4.1.1, we can obtain the information of Sm, including Cs, Ms, Ps and the information of Rm, including Cr, Mr, Pr through the output stack information when the reflection method call occurs. Finally, IntentSoot-D will store Sm, Rm and Tm to the SRT library in the form of a triple group.

4.2 Principle Introduction for IntentSoot-S

When the dynamic execution module is finished, IntentSoot runs IntentSoot-S. First, the necessary Java class files in the apk file and in the dex file are loaded into memory and converted into Soot's internal representation–Jimple. Then the reflection method is converted according to SRT library while the source point is identified. Finally, IntentSoot builds correct CG and CFG and starts static taint analysis.

4.2.1 Class Loading

Soot's static analysis method is based on Soot's Jimple language. Soot loads all the classes contained in the apk file into memory, then builds the main method and constructs the function call graph and the control flow graph. In order to reduce the burden of memory, by using on-demand loading way, IntentSoot-S

```
1 $r14 = new array(java.lang.Object)[1]
2 $r14[0] = "MessageMark";
3 $r15 = virtualinvoke $r13.<java.lang.reflect.Constructor:java.lang.Object newInstance(
      java.lang.Object[])>($r14);
4 ...
5 $r14 = new array(java.lang.Object)[1];
6 $r14[0] = $r7
7 $r5 = virtualinvoke $r15.<java.lang.reflect.Method:java.lang.Object invoke(java.lang.
      Object, java.lang.Object)>($r5, $r14);
```

Listing 1.2. The Jimple code before the transformation of newInstance and invoke

loads the corresponding class in the loaded dex file. In other words, IntentSoot-S only loads the source classes and the target classes in the SRT library. When constructing the function call graph, Soot automatically loads the extra necessary classes according to the extension of the function call graph. The target method may exist either in the loaded dex file or in the class of the app or in the Android system library named android.jar, and these possible files will serve as the base class for Soot.

4.2.2 Modification Algorithm for Source Method

We first give the following definitions to facilitate the description of our algorithm.

- **Target reflection object.** The object of instantiating the target class through reflection mechanism, is represented by Fo, such as the "object" object that is shown in the 33rd line of Listing 1.1.
- **Target object.** The object of directly instantiating target class, is represented by To.
- **Target reflection parameter.** The parameter that is passed to the target method through the reflection mechanism, is represented by Fp, such as the $r14 array shown in the third line and seventh line of Listing 1.2.
- **Reflection object.** The object of instantiating reflection class, is represented by Ro, such as the "constructor" object in 32nd line and the "action" object in the 34th line of Listing 1.1.

An algorithm flow chart for specific source method modification is shown in Fig. 4. For SRT mapping, IntentSoot-S first judges whether the reflection method is newInstance or invoke method. For the invoke method, IntentSoot-S first transforms the target reflection parameter Fp into the target parameter Tp, then determines whether the target reflection object is empty. If it is empty, IntentSoot constructs a static target method call. If it is not empty, IntentSoot-S defines the object To of the target class type, and casts the target reflection object Fo to To, finally executes a non-static method call. The newInstance method contains two types of parameters and no parameters. For the newInstance method with parameters, IntentSoot-S first defines and creates a target object To, then casts the target reflection parameter Fp to the target parameter Tp, invokes the target method Tm (init ()) and instantiates it, finally reassigns the To object to the original target reflection object Fo to ensure that Fo is

```
$r14 = new array(java.lang.Object)[1]
$r14[0] = "MessageMark";
$i1 = 0;
if $i1==0 goto label08;
$r18 = new com.dynamicloadclient.SendMessage;
special invoke $r18.<com.dynamicloadclient.SendMessage:void<init>(java.lang.String)>("
      SendMessage");
$r5 = $r18;
goto label09;
label08:
$r5 = virtualinvoke $r13.<java.lang.reflect.Constructor:java.lang.Object newInstance(
      java.lang.Object[])>($r14);
label09:
...
$r14 = new array(java.lang.Object)[1];
$r14[0] = $r7
$i0 = 0;
if $i0==0 goto label14;
$r17 = (com.dynamicloadclient.SendMessage)$r5;
$r5 = virtualinvoke $r17.<java.lang.reflect.Method:java.lang.Object invoke(java.lang.
      Object, java.lang.Object)>($r5, $r14);
goto label15;
label14:
$r5 = virualinvoke $r15.<java.lang.reflect. Method:java.lang.Object invoke(java.lang.
      Object, java.lang.Object)>($r5, $r14);
label15:
```

Listing 1.3. The Jimple code after the transformation of newInstance and invoke

passed normally in the program. The source code of the 33rd and 35th line in Listing 1.1 corresponds to the Jimple code in Listing 1.2. Listing 1.3 shows the result of the modified source method in Listing 1.2.

4.2.3 Source and Sink Definitions

IntentSoot uses the static taint analysis method to detect the sensitive operation (Sink) by marking the untrusted input data (Source) and statically tracking the propagation path of the taint data. The Source and Sink are defined as follows.

- **Source method:** the GET method of the Intent object that is used to extract information about the Data and Extras fields, such as getStringExtra(), getData() and so on.
- **Sink method:** some security-sensitive operations that are related to the API, such as the page loading-loadUrl(), the command execution-Runtime.exec(), the database query-query() and so on.

4.2.4 Build CG and CFG

CG is a directed graph, of which the nodes represent the functions and the edges represent call points. In order to build a function call graph, IntentSoot needs to determine the entry function of the program first. However, in the lifecycle management of the components in Android, one of the lifecycle methods (e.g., onCreate) is likely to be the entry function of the program.

Since Intent injection vulnerability detection focuses primarily on the delivery of external Intent messages, when a public component receives an Intent message, the extraction of Intent object fields information is usually done in a

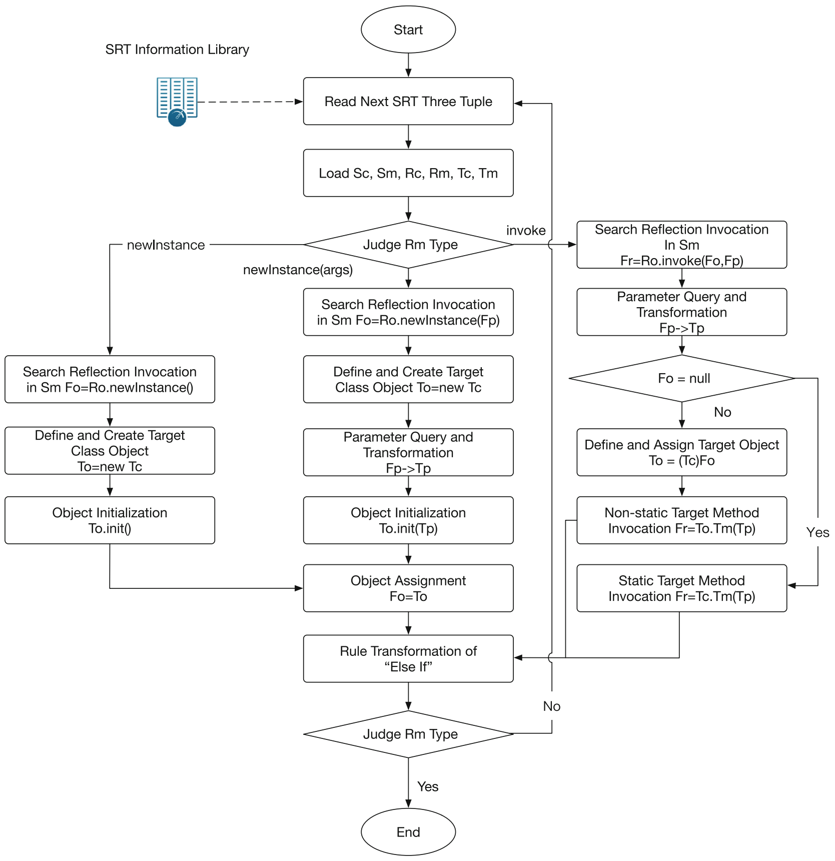
Start
SRT Information Library
Read Next SRT Three Tuple
Load Sc, Sm, Rc, Rm, Tc, Tm
newInstance
Judge Rm Type
invoke
newInstance(args)
Search Reflection Invocation In Sm Fr=Ro.invoke(Fo,Fp)
Search Reflection Invocation in Sm Fo=Ro.newInstance(Fp)
Parameter Query and Transformation Fp->Tp
Search Reflection Invocation in Sm Fo=Ro.newInstance()
Define and Create Target Class Object To=new Tc
Fo = null
No
Define and Create Target Class Object To=new Tc
Parameter Query and Transformation Fp->Tp
Define and Assign Target Object To = (Tc)Fo
Object Initialization To.init()
Object Initialization To.init(Tp)
Non-static Target Method Invocation Fr=To.Tm(Tp)
Yes
Object Assignment Fo=To
Static Target Method Invocation Fr=Tc.Tm(Tp)
Rule Transformation of “Else If”
No
Judge Rm Type
Yes
End

Fig. 4. Algorithm flow chart of modifying source method

specific lifecycle method. According to this feature, we defined the correspondence between different component types and the component entry methods, as listed in Table 1. IntentSoot selects the corresponding component entry method as the entry point of the CG and then analyzes all the function call statements from the CG entry.

CFG is also a directed graph. The nodes in CFG are basic blocks, and the edges in the graph represent the conditional information from one basic block to another basic block. IntentSoot traverses all nodes through Depth-first search algorithm and then generates a CFG for each function in CG.

4.3 Static Taint Analysis

Static taint analysis is based on the constructed CG and CFG, including the taint data trace within a component and between the components.

Table 1. Component entry method

Component type	Component entry method
Activity	void OnCreate(BundlesavedInstanceState);

Service	void OnBind();
	void OnStartCommand();

BroadcastReceiver	void OnReceive();

4.3.1 The Taint Data Trace Within a Component

When the taint data within a component is tracked, IntentSoot identifies the relevant fields information of the Intent object as the taint source, then performs the taint data trace in program analysis to detect whether the taint data flows into Sink method. The taint analysis are analyzed from the entry point of the CG, traversing the CFG of each function. When the method call is encountered, IntentSoot analyzes the taint data propagation in the method call through the binding relationship between formal parameters and actual parameters.

4.3.2 The Taint Data Trace Between the Components

If the method call is an ICC method and its parameter Intent is taint data, it is necessary to track the propagation of the taint information between the components. As shown in the example application of Listing 1.1, when the program is analyzed into the sendBroadcast method in component Activity01, it is necessary to track the propagation of the taint data between components and then continue the taint analysis within the BroadcastReceiver01 component since parameter i2 is the taint data. It is also necessary to track the propagation of taint information between the BroadcastReceiver01 component and the Activity02 component. However, the fact that communication between components is performed by the operating system results in discontinuities of control flow graph between components, as shown in Fig. 5. There are no explicit call process between the sendBroadcast method of the Activity01 component and the onReceive method of the BroadcastReceiver01 component, so static analysis cannot track the data flow information between components.

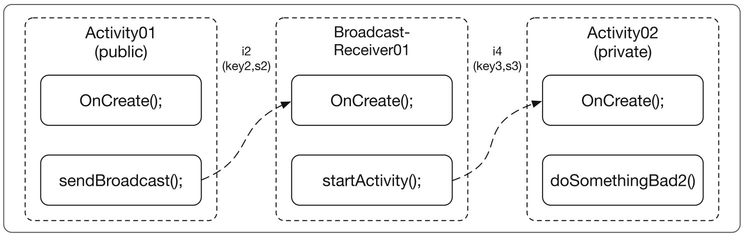

Fig. 5. ICC control flow for example application

To achieve the taint data trace between the components, IntentSoot should first determine the target components. When analyzing the ICC method, we can parse the Intent or the AndroidManifest.xml file to determine the target component. When the target component is determined, IntentSoot achieves the continuity of control flow and data flow between components by constructing a mapping table between the ICC methods and the lifecycle methods, as listed in Table 2. When an ICC method is invoked, it is automatically replaced with the corresponding lifecycle method, and the Intent is passed to the corresponding method for taint data trace.

Table 2. Mapping table between ICC methods and lifecycle methods of target component

Target component type	ICC method	Lifecycle method	Intent information pass

Activity	startActivity(Intent i)
	startActivityForResult(Intent i,· · ·)
		OnCreate(Bundle · · ·)	i→getIntent()

Service	startService(Intent i)
	bindService(Intent i,· · ·)
		OnStartCommand(Intent intent,· · ·)
		OnBind(Intent intent)
			i→Intent
			i→Intent

Broadcast receiver	sendBroadcast(Intent i)
	sendBroadcast(Intent i,· · ·)
	sendOrderedBroadcast(Intent i, · · ·)
	sendOrderedBroadcast(Intent i,· · ·)
	sendStickyBroadcast(Intent i)
	sendStickyOrderedBroadcast(Intent i,· · ·)
		OnReceive(· · · , Intent intent)	i→Intent

4.3.3 Vulnerability Detection

Intent injection vulnerability detection occurs in the process of tracking taint. When IntentSoot analyzes the internal Jimple statement, if the current statement contains a function call, IntentSoot will first determine whether the method is in the defined list of the Sink methods. If so, it further determines whether the relevant parameters are taint data. If so, it means there is a potential Intent injection vulnerability. Then data dependence graph of the parameters is generated, according to which the path of taint trace from the Source to the Sink is recorded.

5 Experimental Results

5.1 Experimental Environment

The experimental platform is 64-bit Ubuntu 14.04 operating system. The number of processor is 6. The frequency of CPU is 2.50 GHZ. The size of physical memory is 6 GB. We selects two sample sets that include the standard test application and the third party application to conduct experiments.

5.2 Contrast Test

In order to evaluate the performance of IntentSoot, we designed five types of standard test applications that contain public component C1, private component C2 and reflection calls, as listed in Table 3.

Table 3. Five types of standard test applications

Application type	Public component C1	Private component C2	Reflection call	Whether exists Intent vulnerability	Intent Soot	Epicc	Average analysis time
A	exist source and sink	-	-	yes	√	√	51/65
B	exist source	sink	-	yes	√	missed positives	52/56
C	exist source	-	sink	yes	√	missed positives	54/61
D	no source	-	-	no	√	false positives	12/45
E	exist source, no sink	-	-	no	√	false positives	46/51

In this experiment, IntentSoot and Epicc were used to analyze the five types of standard applications. Experimental results are shown in Table 3. We can find that Epicc can only detect C-type applications, while IntentSoot can correctly detect all five kinds of applications in terms of detection capability. Since Epicc supposed that there are Intent injection vulnerabilities in all public components, it didn't perform fine-grained analysis for applications. In addition, the average analysis time of IntentSoot is significantly less than Epicc, because IntentSoot analyzes only the public components in the application and the private components on demand, not handling the code of event response function in the process of handling a single component. However, Epicc needs to analyze all the components and all the code within a single component, causing large performance overhead.

5.3 Function Test

We not only analyzed the standard test applications but also collected 60 applications of 10M from third party application market (HiMarket) as test samples, including security management applications, social applications and mobile shopping applications. We found that seven test samples have the path from the Source to Sink. We manually created an exploit trigger to prove that there are indeed Intent injection vulnerabilities, and the test results are listed in Table 4. From the table, we can find that there are a large number of public components in each application. Furthermore, there are Intent injection vulnerabilities in a public component of sample1. The public component can load any page, and the WebView implements the *addJavaScriptInterface* interface, therefore it can embed the JavaScript code to execute remote command. Sink points in Sample3 occur in private components, so IntentSoot requires taint data trace between components. In addition, the number of public components in Sample6 is 11, and the analysis time of Sample6 is significantly more than any other applications, mainly because the code of each public component in Sample6 is much more complex than any other applications, and there may be codes such as reflection calls.

Table 4. Detection results for third party application vulnerability

Sample	Public component numbers/Total component numbers	Sink type	Within components/Between components	Analysis time (s)
Sample1	7/43	Open the web page (Can execute remote commands)	Within components	93
Sample2	8/63	Execute remote commands	Within components	111
Sample3	13/51	Open the web page	Between components	187
Sample4	8/52	Write the specified file	Within components	109
Sample5	8/49	Modify the database	Within components	55
Sample6	11/63	Send a text message to the specified number	Within components	257
Sample7	13/72	Modify the database	Within components	172

6 Conclusion

We proposed IntentSoot, a prototype for detecting Intent injection vulnerability of Android application based on static taint analysis, which considers the taint data trace within a component, between the components and during reflection calls. Compared with previous detection methods, experimental results shows that IntentSoot can effectively reduce false positives and false negatives while reducing performance overhead through only focusing on code snippets from the outside application's Intent message processing. However, IntentSoot does not consider taint propagations of JNI call in the program code, which we will study in the future.

Acknowledgments. This work was partly supported by NSFC under No. 61772466, the Provincial Key Research and Development Program of Zhejiang, China under No. 2017C01055, the Fundamental Research Funds for the Central Universities, the Alibaba-Zhejiang University Joint Research Institute for Frontier Technologies (A.Z.F.T.) under Program No. XT622017000118, and the CCF-Tencent Open Research Fund under No. AGR20160109.

References

1. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: International Conference on Mobile Systems, pp. 239–252. ACM (2011)
2. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: USENIX Security, vol. 2, p. 21 (2011)
3. Gallingani, D.: Static detection and automatic exploitation of intent message vulnerabilities in android applications (2014)
4. Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P.: IccTA: detecting inter-component privacy leaks in android apps. In: International Conference on Software Engineering, vol. 1, pp. 280–291. IEEE (2015)
5. Li, L., Bartel, A., Klein, J., Traon, Y.L., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., Mcdaniel, P.: I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis. Computer Science (2014)

6. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: Chex: statically vetting android apps for component hijacking vulnerabilities. In: ACM Conference on Computer and Communications Security, pp. 229–240. ACM (2012)
7. Maji, A.K., Arshad, F.A., Bagchi, S., Rellermeyer, J.S.: An empirical study of the robustness of inter-component communication in android. In: The 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 1–12. IEEE (2012)
8. Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., Le Traon, Y.: Effective inter-component communication mapping in android with epicc: an essential step towards holistic security analysis. In: USENIX Security, pp. 543–558 (2013)
9. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theor. Comput. Sci. **167**(1), 131–170 (1996)
10. Sasnauskas, R., Regehr, J.: Intent fuzzer: crafting intents of death. In: Joint International Workshop on Dynamic Analysis, pp. 1–5. ACM (2014)
11. Singapati, S.: Inter process communication in android (2012). https://dspace.cc.tut.fi/dpub/bitstream/handle/123456789/21105/singapati.pdf
12. Soot[eb/ol], G.: https://sable.github.io/soot/
13. Wang, R., Xing, L., Wang, X., Chen, S.: Unauthorized origin crossing on mobile platforms: threats and mitigation. In: ACM SIGSAC Conference on Computer & Communications Security, pp. 635–646. ACM (2013)
14. Yuqing, Z., Zhejun, F., Kai, W., Zhiqiang, W., Hongzhou, Y., Qixu, L.: Survey of android vulnerability detection. Comput. Res. Dev. **52**, 2167–2177 (2015)
15. Zhauniarovich, Y., Ahmad, M., Gadyatskaya, O., Crispo, B., Massacci, F.: Stadyna: addressing the problem of dynamic code updates in the security analysis of android applications. In: ACM Conference on Data and Application Security and Privacy, pp. 37–48. ACM (2015)