PIAnalyzer: A precise approach for PendingIntent vulnerability analysis

Sascha Groß, Abhishek Tiwari, Christian Hammer

University of Potsdam, Potsdam, Germany
{saschagross,tiwari}@uni-potsdam.de
hammer@cs.uni-potsdam.de

Abstract. PendingIntents are a powerful and universal feature of Android for inter-component communication. A PendingIntent holds a base intent to be executed by another application with the creator's permissions and identity without the creator necessarily residing in memory. While PendingIntents are useful for many scenarios, e.g., for setting an alarm or getting notified at some point in the future, insecure usage of PendingIntents causes severe security threats in the form of denial-of-service, identity theft, and privilege escalation attacks. An attacker may gain up to SYSTEM privileges to perform the most sensitive operations, e.g., deleting user's data on the device. However, so far no tool can detect these PendingIntent vulnerabilities.
In this work we propose PIAnalyzer, a novel approach to analyze PendingIntent related vulnerabilities. We empirically evaluate PIAnalyzer on a set of 1000 randomly selected applications from the Google Play Store and find 1358 insecure usages of PendingIntents, including 70 severe vulnerabilities. We manually inspected ten reported vulnerabilities out of which nine correctly reported vulnerabilities, indicating a high precision. The evaluation shows that PIAnalyzer is efficient with an average execution time of 13 seconds per application.

Keywords: Android · Intent analysis · information flow control · static analysis.

1 Introduction

The usage of mobile devices is rapidly growing with Android being the most prevalent mobile operating system (global market share of 74.39% as of January 2018 [22]). Android phones are used for a plenitude of highly security critical tasks, and a lot of sensitive information—including session tokens of online services—are saved on these devices. As the official Google Play Store and other alternative Android app marketplaces are not strongly regulated, the main defense against malware that aims to steal sensitive information is the Android sandbox and permission system.

In our study we discover that the Android permission system can be circumvented in many cases in the form of denial-of-service, identity theft, and privilege

escalation attacks. By exploiting vulnerable but benign applications that are insecurely using PendingIntents, a malicious application *without any permissions* can perform many critical operations, such as sending text messages (SMS) to a premium number. PendingIntents are a widespread Android callback mechanism and reference token. While the concept of PendingIntents is flexible and powerful, insecure usage can lead to severe vulnerabilities. Yu et al. [24] report a PendingIntent vulnerability in Android's official Settings app, which made a privilege escalation attack up to SYSTEM privileges possible for every installed application. Thus, given the severe security implications, the official Android documentation on PendingIntents [11] now warns against insecure usage. However, to the best of our knowledge, to-date no analysis tool detects the described PendingIntent vulnerabilities. Thus, an automated analysis tool is envisioned that scales to a large number of applications.

In this work we propose a novel approach to detect PendingIntent related vulnerabilities in Android applications. We implemented our approach in a tool called PIAnalyzer. In multiple analysis steps, PIAnalyzer computes the relevant information of the potentially vulnerable code based on program slicing [26]. PIAnalyzer is fully automated and does not require the source code of the application under inspection. PIAnalyzer assists human analysts by computing and presenting vulnerability details in easily understandable log files. We evaluated PIAnalyzer on 1000 randomly selected applications from the Google Play Store. We discover 435 applications that wrap at least one implicit base intent with a PendingIntent object, out of which 1358 insecure usages of PendingIntents arise. These include 70 PendingIntent vulnerabilities leading up to the execution of critical operations from unprivileged applications. We manually investigate multiple findings and inspect reports on examples known to be vulnerable. Our investigation show that PIAnalyzer is highly precise and sound. Technically, we provide the following contributions:

- *PendingIntent analysis.* We propose a novel method based on program slicing for the detection of PendingIntent related vulnerabilities.
- *Implementation.* We implemented a program slicer for SMALI intermediate code and the proposed PendingIntent analysis in a tool called *PIAnalyzer*.
- *Evaluation of PIAnalyzer.* We empirically evaluate PIAnalyzer on a set of 1000 randomly selected applications from the Google Play Store and find 1358 insecure usages of PendingIntents. These include 70 severe vulnerabilities. We find critical vulnerabilities in widely used libraries such as, *TapJoy* and *Google Cloud Messaging*. PIAnalyzer is efficient and only takes 13 seconds per application on average.
- *Validation of PIAnalyzer.* We manually validated multiple reports of PIAnalyzer. Our validation confirms PIAnalyzer's high precision and recall.

2 Background

Android applications are written in Kotlin [3], Java and C++. From an architectural point of view they may consist of four types of components: Activities,

Services, Broadcast Receivers, and Content Providers. Each of these components acts as an entry point through which a user or the system can interact. Activities are a single screen with a user interface, e.g., the login screen in a banking application. Services run in the background without a user interface, and are intended to perform long-term operations, e.g., Internet downloads. Broadcast receivers are components that receive system or application events to be notified, e.g., when an external power source has been connected. Finally, Content Providers dispense data to applications via various storage mechanisms.

Each component in an Android application is defined in a mandatory manifest file, *AndroidManifest.xml*. The manifest file proclaims essential information about the application, e.g., permissions that are required by the application.

Android applications are compiled from source code to Dalvik bytecode [10], which is specially designed for Android. Finally, the compiled classes along with additional metadata are compressed into an *Android Package* (APK). APKs are made available in different marketplaces such as the official Google Play Store.

As Dalvik bytecode is complex and non-human-readable, a widely used intermediate representation has become the de-facto standard to analyze Android applications: Smali [8] improves code readability and eases the analysis.

2.1 Intents

Android promotes communication between different components via a message passing mechanism: Intents are messages sent from one component to another to invoke a certain functionality, e.g., to start an Activity of another component. Intents can be sent from the system to applications or vice versa, from one application to another (inter-app communication) or even from one component to another within the same application (intra-app-communication) [9].

Central pieces of information associated with intents are a *target component*, an *intent action* and *extra data*. The intent action represents the general action to be performed by the receiving component, e.g., *ACTION_MAIN* is used to launch the home screen. The extra data contains additional information similar to parameters, which can be used by the receiving component, such as passing user input data from one component to another.

Depending on the target component or action, one distinguishes *explicit* from *implicit* intents. An *explicit* intent defines a target component and thus is only delivered to the specified component. Conversely, an *implicit* intent can be delivered to all components that register a matching intent filter in their manifest file. An implicit intent gives an user flexibility to choose among different supported components, e.g., users can opt among different browsers to open a specific webpage. In the event of multiple components registering the same intent filter, the intent resolution asks the user to select, e.g., between multiple browsers to open a particular webpage. In contrast, a *broadcast intent* is broadcast to every registered component instead of only one. Lastly, Android offers *PendingIntent*s. A PendingIntent is intended for another application to perform a certain action in the context of the sending application. The usage and security implications of PendingIntents are discussed in the following.

Listing 1.1. A simple PendingIntent Usage

```
//Component A: Create the base Intent with a target component
  Intent baseIntent = new Intent("TARGET_COMPONENT");
//Create a PendingIntent object wrapping the base Intent
  PendingIntent pendingIntent = PendingIntent.getActivity(this, 1,
     baseIntent, PendingIntent.FLAG_UPDATE_CURRENT);
//Component B (may be in another application or within a system manager):
//Execute the PendingIntent (internally launches the base intent)
 try {
    pendingIntent.send();
} catch (PendingIntent.CanceledException e) {}
```

2.2 PendingIntent

A PendingIntent is a special kind of intent which stores a base intent that is to be executed at some later point in time by another component/application, but with the original app's identity and permissions. The point is that the original app is not required to be in memory or active at that point of time, as the receiver will execute it as if executed by the original application. Thus PendingIntent is applicable in cases where normal intents are not. "A PendingIntent itself is simply a reference to a token maintained by the system describing the original data used to retrieve it. This means that even if its owning application's process is killed, the PendingIntent itself will remain usable from other processes that have been given it" [11]. A possible usage scenario for PendingIntent is a notification. If an application wishes to get notified by the system at a later point of time, it can create a PendingIntent and pass this PendingIntent to the Notification Manager. The Notification Manager will trigger this PendingIntent in the future, and so a predefined component of the application will be notified and gets executed.

Programmatically, the usage of a PendingIntent is a three step process (Listing 1.1). First, the so called *base intent* is created. The base intent is an ordinary intent which defines the action to be performed on the execution of the PendingIntent. The PendingIntent object wraps the base intent using the factory methods *getActivity()*, *getActivities()*, *getBroadcast()* or *getService()*. These factory methods define the nature of the base intent, e.g., *PendingIntent.getBroadcast()* will launch the base intent as a broadcast intent. The PendingIntent object returned by these methods can be passed to another application or system component, e.g., it can be embedded in another intent object (the wrapping intent) as extra data to make it available to other applications. It is also common to pass a PendingIntent object to a system component, e.g., the AlarmManager, for callback purposes.

Security Implications: Whenever a PendingIntent is triggered, the associated base intent is executed in the context (with the same privileges and name) of the application that created it. However, the three main pieces of data of the base intent may be changed even after the PendingIntent has been handed to

Listing 1.2. App A - Vulnerable Activity

```
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main_vuln);
  Intent baseIntent = new Intent();
  PendingIntent pendingIntent = PendingIntent.getActivity(this, 1,
    baseIntent, PendingIntent.FLAG_UPDATE_CURRENT);
  Intent implicitWrappingIntent = new Intent(Intent.ACTION_SEND);
  implicitWrappingIntent.putExtra("vulnPI", pendingIntent);
  sendBroadcast(implicitWrappingIntent);
}
```

another component, which may alter the semantics of the base intent that is to be executed with the original app's identity and permissions. While the Target Component or Action of the base Intent cannot be overridden by an attacker if already defined by the sender, an undefined Action or Target Component may be defined after handing it to the receiver. Finally, extra data, which is effectively a key-value store, can always be added after the fact. The implications include that an implicit intent (with no target component defined) can be altered by the receiving app to target any component it desires (and with the original app's permission support), including system features like wiping the phone.

As a consequence, the Android documentation of PendingIntent [11] explicitly warns about potential vulnerabilities caused by misusage: "By giving a PendingIntent to another application, you are granting it the right to perform the operation you have specified as if the other application was yourself (with the same permissions and identity) (but just for a predefined piece of code). As such, you should be careful about how you build the PendingIntent: almost always, for example, the base Intent you supply should have the component name explicitly set to one of your own components, to ensure it is ultimately sent there and nowhere else." In fact, if a malicious application can retrieve a PendingIntent from another application with an implicit base intent, it may perform a restricted form of *arbitrary code execution* in the context of the application that created the PendingIntent object: As many (but not all) permission-clad functionalities are accessible via intents, the attacker can reroute the base intent to such functions. The next section will exemplify the attack opportunities via PendingIntents.

3 Motivation

3.1 A Potential Vulnerable Example

We demonstrate PendingIntent-related vulnerabilities and exploitation via a simplified example (Listings 1.2 and 1.3). In this example, the vulnerable application has the permission to perform phone calls, while the malicious application does not. In listing 1.2, the vulnerable application creates an empty base intent

Listing 1.3. App B - Malicious Activity

```
1 public void onReceive(Context context, Intent intent) {
2   Bundle extras = intent.getExtras();
3   PendingIntent pendingIntent = (PendingIntent) extras.get("vulnPI");
4   Intent vunlnIntent = new Intent(Intent.ACTION_CALL, Uri.parse("tel:" +
      "0900123456789"));
5   try {
6     pendingIntent.send(context, 2, vunlnIntent, null, null);
7   } catch (PendingIntent.CanceledException e) { e.printStackTrace(); }
8 }
```

(line 4), wraps it into a PendingIntent (line 5), and sends it as an extra of the broadcast intent *implicitWrappingIntent* (line 6, 7, 8). Any application that defines a corresponding intent filter in their manifest file can receive *implicitWrappingIntent*. In listing 1.3, a malicious application which is capable of receiving *implicitWrappingIntent*, extracts the PendingIntent (line 3) and creates a new intent (with the motivation to manipulate the base intent) (line 4) such that it triggers a phone call to some arbitrary number, e.g., to a premium number. On line 6, an invocation of the *send* method of this PendingIntent object causes the execution of the base intent but with all empty properties updated to the values specified in *vulnIntent*, which results in calling the premium number.

While this example is simplified for better understanding, PendingIntent vulnerabilities can occur in various forms and lead to different types of severe security implications. In our study we find that at least 435 out of 1000 applications wrap at least one implicit base intent into a PendingIntent object. The vulnerable application can accidentally send the PendingIntent object in numerous ways. Instead of broadcasting, it can also be sent out via an implicit wrapping intent. If more than one application has a matching intent filter, the user will be asked to choose a destination. This case can be abused by an intent phishing application. In the majority of the cases the PendingIntent is not sent out by wrapping it into another intent, but by passing it to system components such as the AlarmManager or the NotificationManager. These components will eventually call the *send* method of the PendingIntent object, which triggers the base intent. A malicious app can register a component to retrieve the base intent to perform a denial of service attack, as these intents are then not passed to the intended component.

This situation becomes even more critical when the described PendingIntent vulnerability occurs in system components. Tao et al. [24] found this type of vulnerability in the Android Settings application. In the following subsection we elaborate on the details of this vulnerability.

3.2 A Real-world PendingIntent Vulnerability

For all subversions of Android 4, the *Settings* application triggered a PendingIntent with an empty base intent [24]. In February 2018, 17.4% of all Android

Listing 1.4. Android Settings: AddAccountSettings.java

```
1  private void addAccount(String accountType) {
2    Bundle addAccountOptions=new Bundle();
3    mPendingIntent=PendingIntent.getBroadcast(this, 0, new Intent(), 0);
4    addAccountOptions.putParcelable(KEY_CALLER_IDENTITY, mPendingIntent);
5    addAccountOptions.putBoolean(EXTRA_HAS_MULTIPLE_USERS,
       Utils.hasMultipleUsers(this));
6    AccountManager.get(this).addAccount( accountType, null,
7      /* authTokenType */ null, /* requiredFeatures */ addAccountOptions,
8      null, mCallback, null /* handler */);
9    mAddAccountCalled = true;
10 }
```

Listing 1.5. Malicious Application A: Activity 1

```
1    Intent intent = new Intent();
2    intent.setComponent(new ComponentName("com.android.settings",
       "com.android.settings.accounts.AddAccountSettings"));
3    intent.setAction(Intent.ACTION_RUN);
4    intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
5    String authTypes[] = {AccountGeneral.ACCOUNT_TYPE};
6    intent.putExtra("account_types", authTypes);
7    startActivity(intent);
```

devices still run such an Android version [12], rendering them vulnerable to a privilege escalation attack up to system privileges. This vulnerability occurred due to unawareness of the PendingIntent security implications. The fix in Android version 5.0 makes the base intent explicit.

Listing 1.4 shows the code snippet of the corresponding vulnerable method *addAccount*. In this method a PendingIntent object, *mPendingIntent*, is created (line 3) with an empty base intent. Whenever an application requests to add an account of the requested (custom) type, the *addAccount* method gets invoked and the vulnerable PendingIntent (*mPendingIntent*) is returned to this application if it registers to receive `android.accounts.AccountAuthenticator` intents (see Listing 1.6). As this application executes *mPendingIntent* in the context of the *Settings* application (with *SYSTEM* level permissions), it can maliciously overwrite the (empty) action and extra data in the base intent.

Listing 1.5 and 1.6 describe the code snippets of a malicious application *A*, targeting the vulnerability of the *Settings* application. In listing 1.5, *A* initiates an intent to add an account type (line 7). Upon reception of this intent, the *Settings* application invokes the *addAccount* method (cf. Listing 1.4) and sends *mPendingIntent* out. As *A* has registered as *AccountAuthenticator*, it receives this PendingIntent (line 2 of Listing 1.6). On line 3, it creates an intent *vunlnIntent* to perform a Factory Reset[1]. Later it triggers the PendingIntent

[1] A factory reset resets the device to its factory setting, i.e., deletes all data.

Listing 1.6. Malicious Application A: Activity 2

```
1 public Bundle addAccount(AccountAuthenticatorResponse response, String
      accountType, String authTokenType, String[] requiredFeatures, Bundle
      options) throws NetworkErrorException {
2 PendingIntent pi = (PendingIntent)options.getParcelable("pendingIntent");
3 Intent vunlnIntent = new Intent("android.intent.action.MASTER_CLEAR");
4 try {
5   pi.send(mContext, 0, vunlnIntent, null, null);
6 } catch (CanceledException e) { e.printStackTrace(); }
```

with *vunlnIntent* as the updated base intent (line 5). As *A* executes the PendingIntent in the same context as the *Settings* application (with *SYSTEM* level permissions), a Factory Reset is performed.

As previously described, the key cause of this type of vulnerability is the usage of implicit base intents for PendingIntents. Therefore, in this work we provide a novel analysis mechanism which detects implicit base intents in PendingIntents, analyzes their usage and gives a security warning in case of an actual vulnerability.

4 Methodology

4.1 SMALI and SMALI Slicing

PIAnalyzer analyzes the SMALI intermediate representation (IR) of the Dex bytecode extracted from an APK. SMALI is an intermediate representation of Dalvik bytecode that improves readability and analyzability. As background information, Listing A.1 (in the appendix) shows a simplified example of the creation of an Intent object in SMALI code. Similar to Dalvik bytecode, SMALI is register based. As known from assembly languages, registers are universally used for holding values. For example, on line 15 the register *v3* is used to store a String variable, while on line 18 an Intent object is saved in the register *v0*. Please consider the comments in the listing for a more detailed explanation of the code.

PIAnalyzer transforms the bytecode of an APK to its SMALI IR using APKTool [18]. The core of the analysis of PIAnalyzer is performed through program slicing [26] the SMALI representation. Conceptually, a slice is a list of statements that influence a statement (backward slice), or get influenced by a statement (forward slice). For this purpose we design a SMALI slicer. Our SMALI slicer can create both forward and backward slices that are required for the analysis of PIAnalyzer. As registers are SMALI's universal storage mechanism for holding any kind of values, our SMALI slicer is register based. The SMALI slicer is initialized with an arbitrary start position in the code as well as with a set of relevant registers. After completion it returns a set of influencing statements. For example, in Listing A.1 the backward slice of the registers *v0* and *v3*, starting from line 21 will return the statements on lines 15, 18 and 21 as backward

Fig. 1. The workflow of PIAnalyzer

slice. We would like to stress that the PendingIntent analysis described in the following is just one usage of our developed slicer. In fact, our slicer is universal and can be used for various program analysis purposes. The software architecture of PIAnalyzer is designed in a modular way that facilitates the extension by further analysis approaches. Similar to the analysis of PendingIntents, these approaches can easily make use of our generic SMALI slicer.

4.2 PendingIntent Analysis

PIAnalyzer is designed for the efficient analysis of a large number of APKs and therefore accepts as input an arbitrarily large set of APKs. Figure 1 depicts the workflow of PIAnalyzer per APK. The analysis of PIAnalyzer consists of the following steps.

PendingIntent extraction. PIAnalyzer decompiles the DEX bytecode of a given APK to the SMALI IR using APKTool [18]. It then parses the content of each SMALI file together with the application's manifest. PendingIntents can only be created by four methods: *getActivity()*, *getActivities()*, *getBroadcast()* and *getService()* [11]. PIAnalyzer searches in the parsed SMALI files for calls to these methods, leading to a complete list of all PendingIntent creations in the application.

Base Intent analysis. In the next step PIAnalyzer extracts the base Intent object used for creating the PendingIntent. It builds the backward slice from the PendingIntent creation site to the creation site(s) of the base Intent leveraging our universal SMALI slicer. Based on this backward slice, PIAnalyzer determines whether the base Intent is potentially implicit, meaning no target component was definitely set. For determining whether an Intent may be implicit, PIAnalyzer first confirms that an implicit constructor, i.e., a constructor without a specified target component was used to create the base Intent. It then examines whether an explicit transformation method was invoked on the base Intent object. Explicit transformation methods set the target component of an Intent object after it has been constructed, transforming an implicit Intent into an explicit one. To the best of our knowledge only five explicit transformation functions exist at the time of this writing: *setClass()*, *setClassName()*, *setComponent()* *setPackage()* and *setSelector()*. If an Intent has been created by an implicit constructor and no explicit transformation has definitely been invoked on the Intent, it is considered implicit. In the following steps, PIAnalyzer only considers occurrences of PendingIntents with implicit base Intents, as only these can lead to the described security issues (see discussion in section 2.2).

PendingIntent analysis. The severity of the vulnerability depends on the usage of the PendingIntent. Concretely, it depends on the sink functions to which the PendingIntent object is passed. A PendingIntent can either be sent to a trusted system component, e.g. Alarm manager, or wrapped into another Intent. PIAnalyzer therefore computes the forward slice from the creation of the PendingIntent object to either of the mentioned APIs, using our universal SMALI slicer.

WrappingIntent analysis. The most dangerous class of attacks can occur if the PendingIntent object itself is intercepted by a malicious application. This can happen if the PendingIntent is wrapped in another intent (referred to in the sequel as *wrapping Intent*) as Intent extra data. If the wrapping Intent is implicit it can be received by a malicious application to extract its wrapped PendingIntent and manipulate the base Intent. To detect this particularly dangerous class of vulnerabilities, PIAnalyzer examines all wrapping Intents whether they are implicit, as only in this case they can be received by a malicious application. To that end, PIAnalyzer creates the backward slice for all wrapping Intents using our universal SMALI slicer. From the resulting slice it determines whether the wrapping Intent is implicit (in analogy to the base Intent analysis phase), in which case it reports a vulnerability.

Call Graph generation. PIAnalyzer is designed to facilitate the analysis of human security experts. PIAnalyzer assists human investigation of a reported vulnerability via a generated the call graph, which leads to the method in which it has been detected. Thus human experts may determine the events that lead to the execution of the vulnerable code spot. The generated call graph can track control flow between the main application and its used libraries. Additionally, it handles recursive functions.

Reporting. In the last phase, PIAnalyzer logs the results of the analysis. PIAnalyzer creates two types of log files: For each detected vulnerability, it creates a vulnerability log file that reports details of that vulnerability. Additionally, it creates a summary log file that summarizes the findings in the whole APK batch and gives general statistics:

- *Vulnerability Log File* This file contains the slice from the creation of the base intent, over the creation of the PendingIntent object, to the final sink function. Additionally, each vulnerability log file contains the slice from the base Intent to the PendingIntent, as well as the PendingIntent forward slice. Finally, the call graph to the method containing the vulnerability is logged.
- *Summary Log File* For each batch of APKs one summary log file is created. Apart from some hardware specifications, this summary log file contains the total number of warnings and vulnerabilities, as well as some statistics over the batch of APKs.

4.3 Vulnerability severity levels

PIAnalyzer distinguishes the following levels of severity (in increasing order):

Table 1. Distribution of vulnerabilities and warnings

# vuln.	0	1	2	4
# apps	938	56	5	1

# warn.	0	1	2	3	4	5	6	7	8	9	10	13
# apps	565	104	101	96	61	33	16	16	2	2	3	1

- *Secure* PendingIntents with explicit base Intents are considered secure as a known and apparently trusted component is invoked. We respect this trust relation and create no report for these cases.
- *Warning* If a PendingIntent with an implicit base Intent is created, but this PendingIntent is only passed to System managers that are supposed to be benign, it is considered a *Warning*. As a System manager will not redefine the base Intent of the PendingIntent, the only possible attack scenario in this case is a denial of service attack if a malicious application catches the implicit base Intent after the System manager has triggered the *send()* method of the PendingIntent.
- *Vulnerability* PIAnalyzer reports a *Vulnerability* if a PendingIntent has been created with an implicit base Intent and the PendingIntent has been wrapped in another implicit WrappingIntent. In this scenario a malicious application can receive the PendingIntent, and redefine its base Intent resulting in a privilege escalation attack.

5 Evaluation

We applied PIAnalyzer to 1000 randomly selected applications from the Google Play Store. All experiments were performed on a MacBook Pro with MacOS High Sierra 10.13.3 installed, a 2,9 GHz Intel Core i7 processor and 16 GB DDR3 RAM.

PIAnalyzer reports 70 PendingIntent vulnerabilities and 1288 PendingIntent warnings[2]. We statistically analyzed the distribution of vulnerabilities and warnings among the inspected applications. Table 1 depicts the distribution ratios. In the vast majority of the cases a vulnerability does not occur more than once per application. However, the situation is different for warnings. Our findings show that it is likely for an application to include more than one warning. PendingIntents are thus more likely to be delivered to system components (e.g., AlarmManager).

Additionally, we analyzed the proportion of vulnerabilities and warnings that were contained in third party libraries. Remarkably, we find that 80% of the reported vulnerabilities and 98% of the reported warnings occur in third party libraries. Third party libraries thereby act as a multiplier for vulnerabilities, as they are used by a large number of applications. We therefore would like to stress the importance of PIAnalyzer for library developers. Table 2 provides a list of these libraries along with their contribution to the number of vulnerable apps. Libraries are included as the dependencies in the *build.gradle* file [3]. As

[2] For explanations of the severity levels please refer to section 4.3

[3] https://developer.android.com/studio/build/index.html

Table 2. Libraries contribution to number of vulnerabilities

Library	Description	app vuln.	Year
Google Messaging Library	Cloud Messaging	39	2017
Cloud to Device Messaging	Cloud Messaging	8	2016
TapJoy	Marketing and Automation	3	2016
MixPlane	Push Notification & In App Messaging	4	2016
LeanPlum	Messaging, Variable, Analytics & Testing	2	2017

this file is not compiled into the APKs, we could not find the exact version of the library. A tedious way to find the exact version of the library is to match the app's intermediate code with the intermediate code of the each version of the library. We find that these versions of libraries are still in use in the recent versions of applications. Thus, instead of providing the exact version of libraries we provide their year of appearance in an application (in 1000 applications from our experiment).

As mentioned, an attacker can escalate a PendingIntent vulnerability into a privilege escalation attack and leverage the permissions of the vulnerable applications. We therefore analyzed the permissions of the applications for which PIAnalyzer reported vulnerabilities. We find that 279 dangerous permissions [13] and 273 normal permissions are used by these vulnerable applications. As dangerous permissions are required for performing critical operations on the device, an attacker may act maliciously in many of these instances, e.g., call a premium number.

Table 3 provides a list of ten vulnerable applications (randomly selected) along with their category and used dangerous permission groups. The permission groups contain permissions organized into a device's capabilities or features, e.g., *PHONE* group includes the *CALL_PHONE* permission. 35% of the vulnerable applications belong to the *Business, Entertainment, or Education* category.

In our experiment with 1000 real-world applications, the average execution time of PIAnalyzer is close to 13 seconds with a minimum of 10 seconds and the maximum time of 21 seconds. This time performance strongly demonstrates the efficiency of PIAnalyzer and proves that it can easily be applied to a large number of real-world applications.

To evaluate the precision and the soundness of PIAnalyzer, we manually inspected the reported results of ten applications (out of 70 vulnerable applications). Manual inspection is time consuming as it requires analysis of many SMALI code files. Out of ten applications, we find that nine times PIAnalyzer reports correct vulnerabilities/warnings, indicating a high precision. In one case, the base intent was manipulated dynamically and thus PIAnalyzer conservatively overapproximated it as implicit intent.

In addition, we applied PIAnalyzer to the vulnerability in the *Settings* app (described in the section 3.2) that led to privilege escalation to SYSTEM privileges. PIAnalyzer correctly reports the vulnerability and so PIAnalyzer could have prevented the discussed vulnerability. Finally, we applied PIAnalyzer to

Table 3. Vulnerable applications with dangerous permissions

App Name	App Category	Dangerous Permission Group
SandWipPlus	Communication	Contacts, Phone, Sms, Storage
Reason	News & Magazines	Contacts, Location, Phone, Storage
Santa Dance Man	News & Magazines	Phone, Storage
SmartInput Keyboard	Personalization	Phone
drift15house	Entertainment	Calendar, Contacts, Location, Phone, Storage
Fishermens	Entertainment	Contacts, Camera, Location, Microphone, Phone, Storage
Derek Carroll	Photography	Camera, Location, Microphone, Phone, Photography, Sms, Storage
ElleClub	Business	Camera, Contacts, Location, Microphone, Phone, Sms, Storage
Chat Locator	Productivity	Location, Storage
Deptford Mall	Lifestyle	Calendar, Contacts, Location, Microphone, Phone, Sms, Storage

multiple self-written demo examples that included PendingIntent vulnerabilities. PIAnalyzer correctly reports each of them, indicating high recall.

5.1 Case Study: Vulnerability in the Google Cloud Messaging (GCM) Library

PIAnalyzer finds a vulnerability in an outdated version of the Google Cloud Messaging (GCM) Library, which is part of the Google Messaging Library and still in use by many applications, e.g., Table Tennis 3D [1] or the Android Device Manager. Among 1000 analyzed applications, we find that 37 out of 39 (cf. Table 2) applications still use this version of GCM. The vulnerability exists in the file *GoogleCloudMessaging.java* of the GCM Library. Listing 1.7 shows the code snippet of the vulnerable method *send*. On line 4, an implicit intent named *localIntent* is created. On line 6, *localIntent* is passed to a method *c*. Listing 1.8 shows the code snippet for the method *c*. In this method, a PendingIntent with an empty base intent is created (line 4). On line 5, this PendingIntent is stored as extra data to the input parameter *paramIntent*. Later in method *send* (listing 1.7), *localIntent* is broadcast to all registered receivers. Any Broadcast Receiver, declaring this intent filter (*com.google.android.gcm.intent.SEND*) in its manifest file, can receive this Intent and can easily extract the associated PendingIntent. In this case the permissions of the attacker application are escalated to the permissions of applications that use GCM. In our experiments, we are able to intercept *localIntent* and to extract the associated PendingIntent. As the base intent in the associated PendingIntent is blank, we set any arbitrary action/component and trigger it with the same identity as the vulnerable application. This enables us to perform arbitrary actions with the identity of the vulnerable application, e.g., sending a malicious message to a different compo-

Listing 1.7. Vulnerable Method

```
public void send(String paramString1, String paramString2,
       long paramLong, Bundle paramBundle) {
   // ...
  Intent localIntent = new Intent("com.google.android.gcm.intent.SEND");
  localIntent.putExtras(paramBundle);
  c(localIntent);
  localIntent.putExtra("google.to", paramString1);
  localIntent.putExtra("google.message_id", paramString2);
  localIntent.putExtra("google.ttl", Long.toString(paramLong));
  this.eh.sendOrderedBroadcast(localIntent, null);
}
```

Listing 1.8. PendingIntent with an empty base intent

```
void c(Intent paramIntent) {
  try {
    if (this.xg == null)
      this.xg = PendingIntent.getBroadcast(this.eh, 0, new Intent(), 0);
    paramIntent.putExtra("app", this.xg);
  } finally {}
}
```

nent of the vulnerable application and making it believe it was sent from within the application (i.e. identity theft). In the worst case scenario, if this GCM version were used by a system application with system permissions (GCM is an official Google library), a malicious application could for example factory reset the device (deleting all data). We tested several versions (4–7) of system APKs from Google without such inclusions found. However, due to lacking availability we could not check system APKs from other vendors.

5.2 Discussion

PIAnalyzer is a static analysis tool which shares common limitations with other static analysis approaches. As the program behavior can depend on dynamic input, every static analysis tool cannot be completely sound and precise. The slicing analysis of PIAnalyzer is affected by these limitations. In theory, it is possible to make an Intent implicit or explicit depending on external runtime input. This could happen either by making the constructor used for intent creation or the usage of explicit transformation methods depend on external input. When it is not clear at compile time whether an Intent is implicit or explicit, PIAnalyzer conservatively assumes that it is implicit. Analogously, the slices computed by PIAnalyzer are, by their nature, conservative approximations of the actual control flow at runtime. In theory, it is also possible to make use of Intents in Reflection or native code. PIAnalyzer neither supports reflection, nor native code. We would like to stress that while the above mentioned cases are

possible in theory, they are rare in the real world and we could not observe a single instance of these cases during inspection of many applications. Some of the operations that require permissions cannot be performed directly via Intents to system interfaces, e.g., the retrieval of a precise location. Thus an application with only these permissions may not be vulnerable to PendingIntent related attacks.

6 Related Work

To the best of our knowledge, there exists no approach that precisely detects the described PendingIntent vulnerabilities at the time of this writing. In a concurrent effort Trummer and Dalvi [25] developed QARK, an Android vulnerability scanner that also detects PendingIntent based vulnerabilities described in this work. However, QARK ignores the flow of PendingIntents with implicit base intents, so even PendingIntents that are never sent anywhere will be reported a vulnerability. In our tool such a case would only be considered a vulnerability if it flows there via another implicit intent. Additionally, they do not consider whether an implicit intent is transformed to an explicit intent via further API calls. In summary this leads to imprecise results as demonstrated in our evaluation, where we observed that these cases are highly relevant and frequently occur in real-world applications. Their work [25] contains no result other than the prototype tool, particularly no evaluation of their technique.

Bugiel et al. [4] proposed XManDroid, a reference monitor to prevent privilege escalation attacks. Their approach is focused on application permissions and policies to model the desired application privileges. In contrast to our approach, XManDroid only regards PendingIntents as vehicle for inter-component communication and does not consider the peculiarities and vulnerabilities of PendingIntents.

SAAF [14], proposed by Hoffmann et al., is a tool to statically analyze SMALI code. It recovers String constants from backward slices of method calls in order to detect suspicious behavior. However, as it could not produce the expected results in our experiments with current APKs we re-implemented a SMALI slicer.

Li et al. [16] analyzed vulnerabilities in Google's GCM and Amazon's ADM mobile-cloud services. They discovered a critical logical flaws, concerning both of these services. Additionally, they discovered a PendingIntent vulnerability in GCM. Unlike our approach, they discovered this vulnerability by manual code analysis and they do not provide an automated approach for discovering PendingIntent vulnerabilities.

Apart from work considering PendingIntents, there is an extensive body of work on general Intent analysis. One line focuses on Intent fuzzers for finding Intent related vulnerabilities. For example, Yang et al. [27] developed an Intent fuzzer for the detection of capability leaking vulnerabilities in Android applications. JarJarBinks, proposed by Maji et al. [17], is a fuzzing tool for Android intents. By sending a large number of requests, the authors found robustness vulnerabilities in the Android runtime environment. Sasnauskas and Regher [21]

created an Intent Fuzzer. Their approach is based on static analysis and generates random test-cases based on the analysis results. In contrast to our approach, their approaches do not consider PendingIntent related vulnerabilities.

Other work focuses on Intent based test case generation. For example, Jha et al. [15] proposed a model that abstracts Android inter-component communication. From this model the authors derived test cases that facilitates the software engineering process. Salva and Zafimiharisoa [20] proposed APSET, a tool that implements a model-based testing approach for Android applications. The proposed approach generates test cases that check for the leakage of sensitive information. In contrast, our approach is focused on the security perspective of PendingIntents.

As Intents are extensively used by malware, some approaches use Intent analysis as a feature for malware detection. Feizollah et al. [7] proposed AndroDialysis, a tool that uses Intents as indicating feature for Android malware. Tam et al. [23] proposed CopperDroid, a monitor system which tracks events via virtual machine introspection. CopperDroid considers PendingIntent as vehicle for Inter Process Communication. In contrast, our approach is intended for finding PendingIntent related vulnerabilities in benign applications.

Several approaches use various static information flow techniques for Intent analysis. Sadeghi et al. [19] proposed COVERT, a static analysis tool for the analysis of Intents. It computes information flows by static taint analysis. Using COVERT, the authors discovered hundreds of vulnerabilities in applications from the Google Play Store and other sources. Yang et al. [28] proposed AppIntent, an analysis tool for finding leakage of sensitive information via Intents. The key idea of their approach is to distinguish intended information leakage from unintended leakage considering user interface actions. The authors leverage the Android execution model to perform an efficient symbolic execution analysis. Unlike ours, both approaches do not consider PendingIntent related vulnerabilities. Arzt et al. [2] proposed FlowDroid, a taint analysis tool for the static analysis of Android applications. It achieves precise and sound results by appropriately modeling the Android lifecycle and maintaining context, flow, field and object-sensitivity. While FlowDroid is intended for detecting unwanted information flows for the sake of confidentiality and integrity, PIAnalyzer focuses on the detection of vulnerabilities that arise by the wrong usage of PendingIntent and that can not be detected by FlowDroid.

Chin et al. [6] proposed ComDroid, a tool for detecting inter-component related vulnerabilities, e.g., Intent spoofing or Service Hijacking. Chan et al. [5] proposed an approach to detect privilege escalation attacks in Android applications. As their approach does not include any kind of information flow control, it overapproximates possible attacks leading to reduced precision. Again, both do not detect security vulnerabilities caused by PendingIntents.

7 Conclusion

We described the first approach to analyze and detect PendingIntent-related vulnerabilities. We implemented our approach together with a generic SMALI slicer in a tool called PIAnalyzer. PIAnalyzer is fully automated and does neither require the source code of the applications under inspection, nor any effort by the analyst. We evaluated PIAnalyzer on 1000 randomly selected applications from the Google Play Store to assess the runtime performance, precision and soundness of PIAnalyzer. PIAnalyzer takes on average only approximately 13 seconds per application, which scaled up well to large test sets. PIAnalyzer discovers 1288 warnings and 70 PendingIntent vulnerabilities. We manually investigated some of the reports and elaborated on a privilege escalation vulnerability caused by the usage of a prevalent Google library.

Acknowledgements

This work was supported by the German Federal Ministry of Education and Research (BMBF) through the project SmartPriv (16KIS0760).

References

1. Table tennis 3d. Google Play store (April 2014)
2. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices **49**(6), 259–269 (2014)
3. Brains, J.: Kotlin, `https://kotlinlang.org`
4. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R.: Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technische Universität Darmstadt, Technical Report TR-2011-04 (2011)
5. Chan, P.P., Hui, L.C., Yiu, S.: A privilege escalation vulnerability checking system for android applications. In: Communication Technology (ICCT), 2011 IEEE 13th International Conference on. pp. 681–686. IEEE (2011)
6. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proceedings of the 9th international conference on Mobile systems, applications, and services. pp. 239–252. ACM (2011)
7. Feizollah, A., Anuar, N.B., Salleh, R., Suarez-Tangil, G., Furnell, S.: Androdialysis: Analysis of android intent effectiveness in malware detection. Computers & Security **65**, 121–134 (2017)
8. Freke, J.: Baksmali. https://github.com/JesusFreke/smali
9. Google: Android intent documentation. `https://developer.android.com/reference/android/content/Intent.html`, accessed: May. 2017
10. Google: Dalvik byteycode documentation. `https://source.android.com/devices/tech/dalvik/dalvik-bytecode`, accessed: May. 2017
11. Google: Pending intent documentation. `https://developer.android.com/reference/android/app/PendingIntent.html`

12. Google: Android os statistics (Feb 2018), https://developer.android.com/about/dashboards/index.html#Screens
13. Google: Android permissions (April 2018), https://developer.android.com/guide/topics/permissions/overview.html
14. Hoffmann, J., Ussath, M., Holz, T., Spreitzenbarth, M.: Slicing droids: Program slicing for smali code. In: Proceedings of the ACM Symposium on Applied Computing. SAC, ACM, New York (2013)
15. Jha, A.K., Lee, S., Lee, W.J.: Modeling and test case generation of inter-component communication in android. In: Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems. pp. 113–116. IEEE Press (2015)
16. Li, T., Zhou, X., Xing, L., Lee, Y., Naveed, M., Wang, X., Han, X.: Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 978–989. ACM (2014)
17. Maji, A.K., Arshad, F.A., Bagchi, S., Rellermeyer, J.S.: An empirical study of the robustness of inter-component communication in android. In: Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on. pp. 1–12. IEEE (2012)
18. Ryszard Wiśniewski, C.T.: Apktool. https://ibotpeaches.github.io/Apktool/
19. Sadeghi, A., Bagheri, H., Malek, S.: Analysis of android inter-app security vulnerabilities using covert. In: Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on. vol. 2, pp. 725–728. IEEE (2015)
20. Salva, S., Zafimiharisoa, S.R.: Data vulnerability detection by security testing for android applications. In: Information Security for South Africa, 2013. pp. 1–8. IEEE (2013)
21. Sasnauskas, R., Regehr, J.: Intent fuzzer: crafting intents of death. In: Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA). pp. 1–5. ACM (2014)
22. Statcounter.com: Operating system market share worldwide. http://gs.statcounter.com/os-market-share/mobile/worldwide (January 2018)
23. Tam, K., Khan, S.J., Fattori, A., Cavallaro, L.: Copperdroid: Automatic reconstruction of android malware behaviors. In: NDSS (2015)
24. Tao, W., Zhang, D., Yu, W.: Android settings pendingintent leak. https://packetstormsecurity.com/files/129281/Android-Settings-Pendingintent-Leak.html (November 2014)
25. Trummer, T., Dalvi, T.: Qark: Quick android review kit. DefCon 23 (August 2015), https://github.com/linkedin/qark
26. Weiser, M.: Program slicing. IEEE Trans. Software Eng. **10**(4), 352–357 (Jul 1984)
27. Yang, K., Zhuge, J., Wang, Y., Zhou, L., Duan, H.: Intentfuzzer: detecting capability leaks of android applications. In: Proceedings of the 9th ACM symposium on Information, computer and communications security. pp. 531–536. ACM (2014)
28. Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., Wang, X.S.: Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 1043–1054. ACM (2013)

A Appendices

A.1 Simplified SMALI code example

```
.method protected onCreate(Landroid/os/Bundle;)V
    # 5 local registers are used in this method
    .locals 5

    # Declaration of a parameter register with a given name
    .param p1, "savedInstanceState" # Landroid/os/Bundle;

    # End of the method prologue. Start of the actual code.
    .prologue

    # A call to a super constructor
    invoke-super {p0, p1},
     Landroid/support/v7/app/AppCompatActivity;->onCreate(Landroid/os/Bundle;)V

    # Declaration of a String in register v3
    const-string v3, "android.intent.action.CALL"

    # Creation of an Intent object in register v0
    new-instance v0, Landroid/content/Intent;

    # Invocation of the constructor of the intent object
    invoke-direct {v0, v3},
     Landroid/content/Intent;-><init>(Ljava/lang/String;)V

    # Return of the method with no return value
    return-void

.end method
```