PITracker: Detecting Android PendingIntent Vulnerabilities through Intent Flow Analysis

Chennan Zhang[1,2], Shuang Li[1,2], Wenrui Diao[1,2(✉)], and Shanqing Guo[1,2]
[1]School of Cyber Science and Technology, Shandong University
{zcn,lishuang128}@mail.sdu.edu.cn, {diaowenrui,guoshanqing}@sdu.edu.cn,
[2]Key Laboratory of Cryptologic Technology and Information Security of Ministry of Education, Shandong University

ABSTRACT

Intent is an essential inter-component communication mechanism of Android OS, which can be used to request an action from another app component. The security of its design and implementation attracts lots of attention. However, the security of PendingIntent, a kind of delayed-triggered Intent, was neglected by most previous research, and the related analysis techniques are still imperfect. In this paper, we design a novel automated tool, PITracker, to detect the PendingIntent vulnerabilities in Android apps. It achieves the Intent flow tracking technique proposed by us, figuring out how an Intent is created and where it goes. In the real-world evaluations, PITracker discovered 2,939 potential threats in 10,000 third-party apps and 214 in 1,412 pre-installed apps. Among them, 11 exploitable vulnerabilities have been confirmed and acknowledged by the corresponding vendors.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Android; PendingIntent; Vulnerability Detection

ACM Reference Format:
Chennan Zhang, Shuang Li, Wenrui Diao, and Shanqing Guo. 2022. PITracker: Detecting Android PendingIntent Vulnerabilities through Intent Flow Analysis. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '22), May 16–19, 2022, San Antonio, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3507657.3528555

1 INTRODUCTION

The Intent mechanism is a significant ICC (Inter-component Communication) approach of Android OS, which can be used to request an action from another app component, like starting an Activity, starting a Service, or delivering a Broadcast. Given the importance of Intent, several previous works [14, 15, 18] have studied the security of its design and deployments. However, the security of an atypical ICC mechanism was neglected by most of the previous research – *PendingIntent*, which is a kind of delayed-triggered Intent. An app app-A can construct a PendingIntent and pass it to another app app-B. When a specific action is triggered, the app-B can trigger this PendingIntent to launch the component of the app-A. As mentioned in the official documents, *"by giving a PendingIntent to another app, you are granting it the right to perform the operation you have specified as if the other app was yourself (with the same permissions and identity)"* [9]. Therefore, app-B has the right to start app-A's components.

Unfortunately, there is a possibility of permission leakage in the use of PendingIntent. The malware can use PendingIntent to execute malicious actions with the permission of the PendingIntent's owner. In 2020 and 2021, 72 PendingIntent related vulnerabilities are recorded in the CVE database [5]. These vulnerabilities exist in Android 10, Android 11, and pre-installed apps of Samsung productions. In fact, Google has noticed this security issue, and they rejected the update requests of the apps with PendingIntent vulnerabilities in Google Play [10]. In academia, Groß et al. [12] conducted the first study on the PendingIntent vulnerability and designed an analysis tool – PIAnalyzer. However, they did not investigate the principles of the PendingIntent mechanism, leading to ignoring some important factors of vulnerability identification. (More details are discussed in Section 2.3.)

Our Work. To achieve more accurate PendingIntent vulnerability detection, we propose a new automated analysis tool named PITracker based on Intent flow analysis. Specifically, it can analyze each PendingIntent object of an app to figure out how it is created (origin) and where it finally goes (destination). With the above information, PITracker can judge whether a PendingIntent object is vulnerable. To demonstrate the effectiveness of PITracker, we evaluated it on 10,000 randomly selected apps from five third-party app markets and 1,412 pre-installed apps of 6 phones from different vendors. We identified 2,939 potential threats in third-party apps and 214 in pre-installed apps. To pre-installed apps, we reported 11 exploitable vulnerabilities to the corresponding vendors (2 for vivo, 1 for OPPO, 3 for Xiaomi, 1 for Lenovo, and 4 for MEIZU), and all of them had been confirmed. Besides, we manually verified the findings of 50 apps output by the tool. The results show that the tool has high precision.

Contributions. Here we list the main contributions of this paper:

- **New Technique.** We propose the Intent flow analysis technique to detect the PendingIntent vulnerabilities. Based on this technique, we designed a new tool – PITracker.
- **Evaluations in the Wild.** With PITracker, in experiments, we identified 2,939 potential threats in third-party apps and 214 in pre-installed apps.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WiSec '22, May 16–19, 2022, San Antonio, TX, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9216-7/22/05…$15.00
https://doi.org/10.1145/3507657.3528555

https://t.me/learningnets

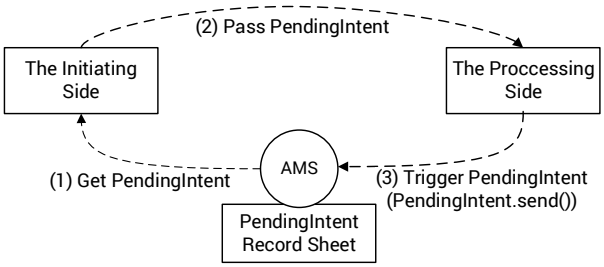

Figure 1: Life cycle of PendingIntent.

- **Real-world Vulnerabilities.** In total, 11 vulnerabilities discovered by PITracker have been confirmed and acknowledged by 5 mainstream phone vendors.

2 BACKGROUND AND RELATED WORK

In this section, we provide the necessary background of the Android PendingIntent mechanism. Also, related works are reviewed.

2.1 Android PendingIntent

As mentioned in Section 1, Intent is important for Android ICC. PendingIntent is a wrapper for Intent. It provides an atypical way to perform ICC. The most common application scenario of PendingIntent is notification. For example, a weather app pops up a notification reminding the user to check tomorrow's weather forecast. She clicks this notification, and then the phone's UI can immediately jump to the weather app and display the content of tomorrow's weather forecast. During the internal implementation, when the user clicks this notification, a PendingIntent is triggered. It performs the action of launching the weather app's Activity.

The life cycle of a PendingIntent is illustrated in Figure 1. A component of an app (i.e., the initiating side) first creates a base Intent. It specifies what it wants to do with this Intent through the Intent APIs. The next step is wrapping this base Intent into a PendingIntent. To do this, the following methods are provided for getting a PendingIntent object in Android and frequently used by developers (We omitted the parameters of these methods and each of them takes an Intent as an argument): `PendingIntent.getActivity()`, `PendingIntent.getBroadcast()`, `PendingIntent.getService()` and `PendingIntent.getForegroundService()` [9]. These different APIs represent different components to be launched. AMS (Activity Manager Service) will look this PendingIntent up in a record sheet. If this PendingIntent already exists, AMS will give the proxy object of it to the initiating side. If AMS cannot find this PendingIntent, it will create a new PendingIntent. After recording this new PendingIntent in the sheet, the proxy object of it will be passed to the initiating side (Step 1). Then the PendingIntent object will be handed over to another component (Step 2), i.e., the processing side. When a specific action is triggered, the processing side can use `PendingIntent.send()` to trigger this PendingIntent (Step 3). AMS will find this PendingIntent in the sheet and trigger it. It is noteworthy that when this PendingIntent is triggered, the base Intent in it is executed in the context (with the same privileges and name) of the initiating side, and this is the key point of PendingIntent vulnerability.

2.2 PendingIntent Vulnerability

As Groß et al. [12] pointed out that when the processing side triggers the PendingIntent, the semantics of the base Intent that is to be executed with the original app's identity and permissions can be changed. The processing side can create a new Intent and make it as the third argument of an overloaded method of `PendingIntent.send()`:

```
send(Context context, int code, Intent intent,
    PendingIntent.OnFinished onFinished,
    Handler handler, String requiredPermission,
     Bundle options)
```

Listing 1: `PendingIntent.send()`.

After a series of function invocations, this Intent will be passed to the `fillIn()` method of the Intent class [7]. As shown in Listing 2, the new incoming Intent (i.e., `other` at line 1) is analyzed in this method, and when its original action field (i.e., `mAction` at line 4) is empty, or the `FILL_IN_ACTION` flag is set, the original action will be changed to the new one (i.e., `other.mAction`). In addition, the following fields can also be modified in the same way (we call these fields *modifiable fields*): *package*, *category*, *data*, *flag*, *clipdata* and *extra*. However, the component field (i.e., `mComponent` at line 8 and line 9) of the original Intent can only be changed when the flag `FILL_IN_COMPONENT` is set regardless of whether the initial filed is empty or not. The *selector* field can also be changed in this way. We call them *explicit modifiable fields*.

```
1  public int fillIn(NonNull Intent other,
       FillInFlags int flags) {
2      ...
3      if (other.mAction != null && (mAction ==
       null || (flags&FILL_IN_ACTION) != 0)) {
4          mAction = other.mAction;
5          changes |= FILL_IN_ACTION;
6      }
7      ...
8      if (other.mComponent != null && (flags&
       FILL_IN_COMPONENT) != 0) {
9          mComponent = other.mComponent;
10         changes |= FILL_IN_COMPONENT;
11     }}
```

Listing 2: `fillIn()` **of** `Intent.class`.

However, we found few apps set the above flags when using PendingIntent in practical analysis. Therefore, the modifiable fields are considered changeable when they are initially empty, and the explicit modifiable fields are basically cannot be changed. Therefore, when the modifiable fields of the original base Intent are not set and the wrapping PendingIntent is obtained by the malware, it can modify these fields at will. Then it can trigger this PendingIntent with the identity of the initiating side. In other words, the malware can do something bad as the initiating side. By refining different empty modifiable fields, the malware can perform different attacks. For example, in CVE-2014-8609 [4], the Setting app, which requests lots of dangerous permissions, leaks a PendingIntent with no modifiable fields set in its base intent. The malware can utilize it to create a phishing SMS by refining the `action` and `extra` fields or force the phone to factory reset by refining the `action` field.

https://t.me/learningnets

It is worth mentioning that Android has added a new flag `FLAG_IMMUTABLE` in API level 23 [9]. If this flag is used when creating the PendingIntent, the semantics of the base Intent cannot be changed. Therefore, this vulnerability can only exist when the PendingIntent object is created without setting this flag.

As a consequence, we can conclude **three significant conditions** for the PendingIntent vulnerability:

(1) The modifiable fields of base Intent have not been set.
(2) The PendingIntent object can be obtained by other apps.
(3) The `FLAG_IMMUTABLE` flag has not been set.

2.3 Related Work

Samhi et al. [16] pointed out that the Android framework provides atypical ways of performing ICC, such as PendingIntent and IntentSender. They found 111 AICC methods in total. We filtered out the API calls of the system managers (such as AlarmManager, LocationManager, NotificationCompat$Builder, etc.) that took the PendingIntent (Intentsender) object as a parameter in these methods. The PendingIntents going to these methods, except for the methods related to Notification, cannot be easily obtained by malware.

As the first study on the PendingIntent vulnerability, Groß et al. [12] used the smali-based program slicing for data flow analysis to detect vulnerabilities. However, they only considered the PendingIntent created with an implicit base Intent (i.e., no target component was set) and ignored other fields. They assumed that the PendingIntent passed to system managers could not cause harm. In fact, we found that when the PendingIntent is passed to the Notification components, the malware can also obtain it. Besides, they did not evaluate the pre-installed apps with dangerous permissions.

He et al. [13] developed a Soot-based tool [6] to detect the PendingIntent vulnerabilities. They still did not conduct a further inspection on the entire Intent flow, say ignoring the destination of PendingIntent. Actually, many PendingIntent objects are not exposed to other apps and cannot be exploited. The efficiency of Soot also brings some performance issues. As discussed in [17], the Soot-based analysis tools are less efficient. They will first perform an overall analysis of the entire APK file and then perform a data flow analysis. For the PendingIntent vulnerability, it is unnecessary to analyze the entire APK file, and we only need to focus on PendingIntent related variables and codes.

3 DESIGN

In this section, we introduce the detailed design of PITracker[1]. The framework of PITracker is illustrated in Figure 2, and it can be divided into three steps: i) *Preparation*, ii) *Key Information Collection*, and iii) *Assessment*.

In the preparation step, PITracker transforms the APKs of apps to smali files and locates the PendingIntent creation sites. Then it uses program slicing to collect key information related to PendingIntent vulnerability in part ii). Finally, the tool gives an assessment of each PendingIntent object.

[1]The source code of PITracker can be found at https://github.com/Sp1keeeee/PItracker.

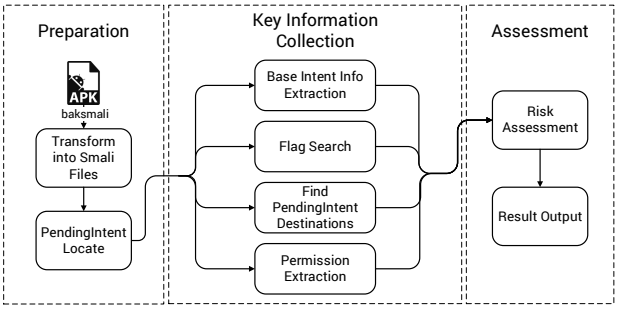

Figure 2: The framework of PITracker.

Table 1: Fields set methods.

Field	Method
action	setAction (String action)
package	setPackage (String packageName)
data	setData (Uri data) setDataAndType (Uri data, String type)
clipdata	setClipData (ClipData clip)

3.1 Preparation

As the first step, we need to decompile the APK files and locate the methods of creating PendingIntent objects in target apps. PITracker extracts the DEX files from apps and then uses baksmali [3] to convert the DEX files into smali files for static analysis. It will search the PendingIntent creation methods mentioned in Section 2 in the smali files. After this, we can get a list of creation sites of all PendingIntents.

3.2 Key Information Collection

We are concerned about the related conditions of PendingIntent vulnerability as mentioned in Section 2. Therefore, we need to collect the information about the base Intent, the creation flag of PendingIntent, and the flow of PendingIntent (especially the origin and destination). Besides this, we need to know what permissions may be leaked by the origin app, so PITracker will extract its requested permissions.

Base Intent Info Extraction. In this process, PITracker collects the information about whether the base Intents have set the modified fields. It uses backward program slicing from the PendingIntent creation methods to check whether the base Intent objects have called modified fields setting methods. The methods for setting related fields are listed in Table 1. The traditional backward slicing iterate over the statements in a function. However, in practical scenarios, the decompiled smali code has many jump and condition instructions. The statements order in smali files may not be the real order. For example, the code snippet of creating a PendingIntent object in a smali file is listed below:

```
1 #PendingIntent creation statements
2 :cond_5
3 const-string v10, "enter_homepage_way"
```


https://t.me/learningnets

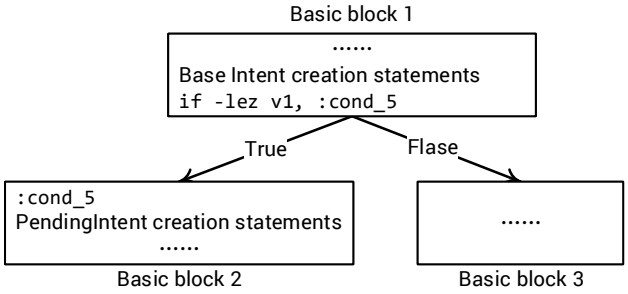

Figure 3: The CFG of Listing 3.

```
4  invoke-virtual {v4, v10, v2}, Landroid/content/
       Intent;->putExtra(Ljava/lang/String;Ljava/
       lang/String;)Landroid/content/Intent;
5  const/high16 v2, 0x40000000
6  invoke-static {v0, v3, v4, v2}, Landroid/app/
       PendingIntent;->getActivity(Landroid/
       content/Context;ILandroid/content/Intent;I)
       Landroid/app/PendingIntent;
7      ...
8  #Base Intent creation statements
9  new-instance v4, Landroid/content/Intent;
10 invoke-direct {v4, v10}, Landroid/content/
       Intent;-><init>(Ljava/lang/String;)V
11     ...
12 if-lez v1, :cond_5
13     ...
```

Listing 3: Example of creating PendingIntent.

We can see that the base Intent creation statements are at Line 9 and Line 10. However, the PendingIntent creation statement is at Line 6. Therefore, we cannot simply iterate over the related statements backwardly in smali files for backward slicing. To solve this problem, PITracker creates a CFG (control flow graph) for each function needing analysis. For example, the CFG of the above code is illustrated in Figure 3.

Based on the CFG, our backward slicing is to iterate over the statements in a basic block backwardly and collect the related information. When all statements in the basic block have been checked, PITracker will go to the block's parents' blocks to do the same thing until finding the end statement. Therefore, PITracker starts backward slicing from the PendingIntent creation statements until it finds the end statement like: `new-instance vx, Landroid/content/Intent` (`vx` is the register of the Intent object). Note that, a basic block may have many parents blocks, and the base Intent may come from different parents blocks. Thus, PITracker uses the DFS (Depth First Search) algorithm to iterate over each parent's blocks until finding all base intents.

Flag Search. The flag for creating the PendingIntent is also very critical. The malware cannot modify the semantic of a PendingIntent object if the flag `FLAG_IMMUTABLE` is set. PITracker uses backward slicing to track the fourth parameter of the PendingIntent creation method until an integer value is set to its register. We are concerned about whether the value of the flag is `0x04000000` (i.e., `FLAG_IMMUTABLE`) in this process.

Find PendingIntent Destinations. After obtaining the information of the base Intent and the flag, we only know whether the semantics of this PendingIntent is changeable. However, another necessary condition of the PendingIntent vulnerability is whether malicious apps can obtain it. Here we summarize two common cases in that third-party (malicious) apps can obtain PendingIntents.

(1) *Implicit Intent.* As proposed by Groß et al. [12], when a PendingIntent is wrapped into an implicit Intent, any app can easily acquire it.
(2) *Notification.* The PendingIntent contained in the notification can be obtained by the service components that inherit from notificationlistenerService [8] and accessibilityService [2]. The malware can construct a service to inherit any of these two services, and then this service can use related methods to get the PendingIntent in Notification.

Therefore, we need to find the locations where the PendingIntent finally goes, and we call these places the PendingIntent destinations. In the practical analysis process, we identified seven types of PendingIntent destinations in the same function.

- **System API.** Chen et al. [16] find 111 atypical methods of performing ICC, which use PendingIntent object or IntendSender object as a parameter. We filtered out the API calls of the system managers (such as AlarmManager, NotificationCompat$Builder, etc.). If the PendingIntent objects are passed to these APIs, we no longer continue slicing forward, and we consider the PendingIntent destination as system API. We focus on whether this system API is related to Notification in this situation.
- **Intent.** The PendingIntent may also be wrapped into an Intent's `extra` field. In this situation, PITracker will continue slicing forward to check whether the Intent calls `setClass()`, `setClassName()`, `setComponent()`, `setPackage()`, and `setSelector()`, which are the APIs that can make it explicit. We are concerned about implicit Intent that can be obtained by other apps.
- **Not Used.** After the PendingIntent object is created, there are no methods to take it as a parameter. Even if the backward slicing determines that it may be dangerous, it is not actually available to other apps.
- **Send.** During our analysis, we found that some PendingIntent objects were triggered immediately in the same function after they were created. Although this is very rare, we need to consider it for accurate analysis.
- **Other Methods.** There are two ways of passing a PendingIntent object to other methods. The first is taking it as a parameter: PITracker will continue going to the target method to trace the relevant statements of the parameter until it goes to the cases above. Another way is taking it as the return value: in this situation, PITracker will search the smali files of the entire app to find the caller of this function, and then continue to trace in these functions until the PendingIntent object goes to the destinations above.
- **IntentSender.** IntentSender is created from PendingIntent by using `getIntentSender()`. It actually has the same function as PendingIntent. Therefore, if a PendingIntent has

https://t.me/learningnets

changed to an IntendSender, PITracker will continue to find its destination in the same way as PendingIntent.

PITracker uses forward slicing from the PendingIntent creation statements, and the end statements is above destinations. It uses the DFS algorithm to iterate over all the children blocks of the CFG for finding the destinations. The only difference between backward slicing and forward slicing is the direction, so we omit the details of the forward slicing.

Permission Extraction. The greatest harm of PendingIntent vulnerability is that the malware can use the initiating side's permission to do something bad without requesting these permissions, so we need to figure out what permissions the owner of the PendingIntent has requested. PITracker uses AAPT2 [1], the official build tool of Android, to extract permissions from apps.

3.3 Assessment

Risk Assessment. After obtaining all necessary information, PITracker will conduct a risk assessment for each PendingIntent object. If any condition of PendingIntent vulnerability is not met, the corresponding PendingIntent is considered secure. Otherwise, PITracker will give a risk label to target PendingIntent depending on the obtained information. There are two kinds of labels:

- **Component Hijacking Risk.** If the base Intent's component has been set, the operation triggered by its PendingIntent can only work in the scope of the target component. The malware can only use this PendingIntent to launch the corresponding components of the original app. Besides, the security threat will be weak if the original app does not request dangerous permissions. However, there is still a possibility of component hijacking which has been studied in previous works [11, 15]. The main reason for this vulnerability is that the developers carelessly expose some of the apps' components to others, so the malware can easily use related invokable interfaces to execute malicious actions. To PendingIntent vulnerability, the malware can arbitrarily launch components of the original app to perform DDoS, data theft, and so on, even if the component is not exposed.
- **Permission Re-delegation Risk.** This risk will happen when the owner of a vulnerable PendingIntent has requested dangerous permissions. The dangerous permissions contain some general runtime permissions like `CALL_PHONE`, `SEND_SMS`, `READ_CONTACTS` and system permissions that cannot request by third-party apps like `SHUTDOWN`, `REBOOT`. Exploiting PendingIntents, the malware can perform malicious actions without requesting the corresponding permissions. We need to point out that the apps that have the permission re-delegation risk must exist the component hijacking risk.

Result Output. PITracker finally outputs all information (i.e., where the base Intent and PendingIntent are created, what modifiable fields of the base Intent have not been set, and what dangerous permissions the apps have requested) to help the security analysts know more about the identified vulnerable PendingIntents.

4 EVALUATIONS

In this section, we summarize the experiment results.

Table 2: Results of apps of third-party app markets (CH – Component Hijacking, PR – Permission Re-delegation).

App Market	Not Used	Secure	CH Risk	PR Risk	Total
Xiaomi	898	512	212	378	2000
HUAWEI	924	700	96	280	2000
PC6	592	724	158	526	2000
Apkpure	70	1030	202	698	2000
Lenovo	1100	511	73	316	2000
Total	3584	3477	741	2198	10000

Table 3: Results of pre-installed apps.

Phone Model	Not Used	Secure	CH Risk	PR Risk	Total
IQoo U3x	84	157	9	31	281
OPPO A32	92	170	11	11	284
Redmi K40	101	139	18	33	291
Lenovo Lemon K12	82	90	14	16	202
MEIZU 17	78	78	7	49	212
Coolpad CP30	56	71	7	8	142
Total	493	705	66	148	1412

4.1 Experiment Setup

We randomly selected 10,000 apps from 5 third-party app markets – Xiaomi, HUAWEI, PC6, Apkpure, and Lenovo for evaluation (2,000 for each). In addition, we selected 6 popular models of vivo, OPPO, Xiaomi, Lenovo, MEIZU, and Coolpad to analyze their pre-installed apps. The pre-installed apps usually have a lot of dangerous permissions, especially system permissions.

4.2 Results and Findings

The results are summarized in Table 2 and Table 3. We can see that the potential threat of PendingIntent vulnerability is common in the apps from third-party app markets. More permission re-delegation risks are identified than the component hijacking risk. The reason is that most apps in third-party app markets request the `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE` permissions. The malware can utilize them to get the users' photos, videos, and other files stored in external storage. For pre-installed apps, we can find that the threat of PendingIntent vulnerability also exists, and many of them request one or more dangerous permissions.

We manually inspected the smali code of 50 randomly selected apps, and the information outputted by the tool is in good agreement with the practical case, indicating a high precision. Besides, we manually verified the PendingIntent threats in pre-installed apps and finally discovered 11 exploitable PendingIntent vulnerabilities (2 for vivo, 1 for OPPO, 3 for Xiaomi, 1 for Lenovo, and 4 for MEIZU). We reported them to the corresponding vendors, all of which have been confirmed (also assigned CNVD-2021-102096 and CNVD-2021-100644). Note that, due to code obfuscation and lack of knowledge of PendingIntents usage scenarios, we cannot easily figure out how the apps use these vulnerable PendingIntents and further construct the corresponding PoCs (for vulnerability report). Therefore, we

https://t.me/learningnets

did not verify all discovered potential threats one by one. Even though, these current acknowledgments of vendors have proved the effectiveness of PITracker.

4.3 Case Study

We identified a PendingIntent vulnerability in a pre-installed app `com.android.contacts` of a mainstream phone vendor[2] which can be used by malware to call arbitrary phone numbers and read contacts information without requesting related permissions. The vulnerable code snippet is listed below.

```
1 static Notification
      constructImportFailureNotification(Context
      context, String str, String str2) {
2     Notification.Builder contentIntent = new
      Notification.Builder(context).
      setContentIntent(PendingIntent.getActivity(
      context, 0, new Intent(), 0));
3       ...
4 }
```

Listing 4: The vulnerable code.

The vulnerable PendingIntent is created at Line 2. It is created through the `PendingIntent.getActivity()` method. A base Intent, not set any fields, is used to create this PendingIntent. It is finally wrapped into `Notification.Builder` through the `setContentIntent()` method. This method will be called when this app fails to import contacts from vCard. In addition to that, this app has requested many permissions like: `CALL_PRIVILEGED`, `CALL_PHONE`, `READ_CONTACTS`, and `WRITE_CONTACTS`.

As a result, a malicious app can construct a service extending `accessibilityService` or `notificationlistenerService` for monitoring the notification events. The PoC code can be like below.

```
1 public class NotificationLisner extends
      AccessibilityService {
2 public void onAccessibilityEvent(
      AccessibilityEvent event) {
3     if (event.getParcelableData() != null &&
      event.getParcelableData() instanceof
      Notification) {
4         Notification notification = (
      Notification) event.getParcelableData();
5         PendingIntent pendingIntent =
      notification.contentIntent;
6         Uri uri = Uri.parse("tel:" + "911");
7         Intent mintent = new Intent(Intent.
      ACTION_CALL, uri);//refine the base Intent
8         try {
9             pendingIntent.send(
      NotificationLisner.this,0,mintent,null,null
      );//trigger a new PendingIntent
10        } catch ...}
```

Listing 5: The code of the POC.

In this PoC, We construct a `NotificationListener` class to extend `AccessibilityService` and overwrite the `onAccessibility-Event()` method. A new Intent *mintent* which is to call the number 911 is passed to the `PendingIntent.send()` method. Therefore, when this vulnerable app pops up this notification, the malicious app can use it to call arbitrary phone numbers without requesting the `CALL_PHONE` permission. Besides, this malicious app can arbitrarily read and write the contacts, and we omit it due to space limitations. We have reported this vulnerability to the target vendor. They have acknowledged and rewarded our findings.

[2]We cannot give its name due to this vendor's responsibility disclosure rule.

5 CONCLUSION

This paper proposed the Intent flow analysis technique to detect the PendingIntent vulnerabilities in Android apps. By analyzing over 10,000 third-party apps (from 5 app markets) and 1,412 pre-installed apps (from 6 phones), PITracker uncovered 2,939 and 214 potential threats, respectively. Also, 11 discovered vulnerabilities have been confirmed and acknowledged by the corresponding vendors.

REFERENCES

[1] 2021. *AAPT2.* Retrieved February 2, 2022 from https://developer.android.com/studio/command-line/aapt2?hl=en
[2] 2021. *AccessibilityService.* Retrieved February 2, 2022 from https://developer.android.com/reference/android/accessibilityservice/AccessibilityService
[3] 2021. *baksmali.* Retrieved February 2, 2022 from https://github.com/JesusFreke/smali
[4] 2021. *CVE-2014-8609.* Retrieved February 2, 2022 from https://packetstormsecurity.com/files/129281/Android-Settings-Pendingintent-Leak.html
[5] 2021. *CVEs of PendingIntent.* Retrieved February 2, 2022 from https://developer.android.com/guide/components/intents-filters
[6] 2021. *http://soot-oss.github.io/soot/.* Retrieved February 2, 2022 from http://soot-oss.github.io/soot/
[7] 2021. *Intent.java.* Retrieved February 2, 2022 from https://cs.android.com/android/platform/superproject/+/master:frameworks/base/core/java/android/content/Intent.java
[8] 2021. *NotificationListenerService.* Retrieved February 2, 2022 from https://developer.android.com/reference/android/service/notification/NotificationListenerService
[9] 2021. *PendingIntent.* Retrieved February 2, 2022 from https://developer.android.com/reference/android/app/PendingIntent
[10] 2021. *PendingIntent Remediation.* Retrieved February 2, 2022 from https://developer.android.com/guide/components/intents-filters
[11] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David A. Wagner. 2011. Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys), Bethesda, MD, USA, June 28 - July 01, 2011.*
[12] Sascha Groß, Abhishek Tiwari, and Christian Hammer. 2018. PIAnalyzer: A Precise Approach for PendingIntent Vulnerability Analysis. In *Proceedings of the 23rd European Symposium on Research in Computer Security (ESORICS), Barcelona, Spain, September 3-7, 2018.*
[13] En He, Wenbo Chen, and Daoyuan Wu. 2021. Re-route Your Intent for Privilege Escalation. *BlackHat 2021* (2021).
[14] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proccedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE), Florence, Italy, May 16-24, 2015.*
[15] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS), Raleigh, NC, USA, October 16-18, 2012.*
[16] Jordan Samhi, Alexandre Bartel, Tegawendé F. Bissyandé, and Jacques Klein. 2021. RAICC: Revealing Atypical Inter-Component Communication in Android Apps. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE), Madrid, Spain, May 22-30, 2021.*
[17] Daoyuan Wu, Debin Gao, Robert H. Deng, and Rocky K. C. Chang. 2021. When Program Analysis Meets Bytecode Search: Targeted and Efficient Inter-procedural Analysis of Modern Android Apps in BackDroid. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Taipei, Taiwan, June 21-24, 2021.*
[18] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Hai-Xin Duan. 2014. IntentFuzzer: Detecting Capability Leaks of Android Applications. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (AsiaCCS), Kyoto, Japan, June 3-6, 2014.*

https://t.me/learningnets