Lost in Conversion: Exploit Data Structure Conversion with Attribute Loss to Break Android Systems

Rui Li, *School of Cyber Science and Technology, Shandong University; Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, SDU; The Chinese University of Hong Kong;* Wenrui Diao and Shishuai Yang, *School of Cyber Science and Technology, Shandong University; Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, SDU;* Xiangyu Liu, *Alibaba Group;* Shanqing Guo, *School of Cyber Science and Technology, Shandong University; Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, SDU;* Kehuan Zhang, *The Chinese University of Hong Kong*

https://www.usenix.org/conference/usenixsecurity23/presentation/li-rui

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Lost in Conversion: Exploit Data Structure Conversion with Attribute Loss to Break Android Systems

Rui Li*†‡, Wenrui Diao*†(✉), Shishuai Yang*†, Xiangyu Liu§, Shanqing Guo*†, and Kehuan Zhang‡
* *School of Cyber Science and Technology, Shandong University*
leiry@mail.sdu.edu.cn, diaowenrui@link.cuhk.edu.hk
† *Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, SDU*
‡ *The Chinese University of Hong Kong* § *Alibaba Group*

Abstract

Inside the operating system, the processing of configuration files tends to be complicated and involves various data operation procedures. On Android, the processing of manifest files (the principal configuration files of Android apps) correlates to multiple core system mechanisms, such as permission and component management. It is widely recognized that improperly configured manifest files can put apps at risk. Even worse, we find that vulnerable configuration data processing can be exploited by crafted manifest files to break the Android system mechanisms, even achieving privilege escalation.

In this work, we systematically studied the Android manifest processing procedures and discovered a new category of vulnerabilities called the Evil Twins flaw. In brief, during the processing of twin manifest elements (with the same name but different attributes), the ill-considered data structure conversion (e.g., from `List` to `Map` without considering the duplication issue) merges them into one item with attribute loss, further resulting in system configuration inconsistency, i.e., potential security risks. To detect the Evil Twins flaw lying in the Android OS, we designed an automated analysis tool, TwinDroid, to identify the data structure conversions with attribute loss and then manually confirm the vulnerabilities. With TwinDroid, we assessed the code of AOSP Android 11 & 12. Finally, 47 suspicious methods were reported, and four vulnerabilities were identified, which can be exploited to achieve permission escalation and revoking prevention. All discovered vulnerabilities have been acknowledged by Google, and three CVE IDs have been assigned.

1 Introduction

The configuration files are critical for computer applications, which describe how the system should enforce the management policies. On the other hand, inside the operating system, the processing of configuration files tends to be complicated and involves various data operation procedures. Thus, for security, we need to ensure the correctness of both configuration files and their subsequent processing.

On Android – the most popular smartphone platform, as the principal app configuration file, the manifest file (i.e., `AndroidManifest.xml`) describes the essential information about an app, such as contained components and requested permissions [2]. The previous research has demonstrated that app developers may misconfigure the manifest files of their apps, for example, through declaring duplicate components [25, 31] or misplacing attributes [31, 38]. Such misconfiguration may make their apps insecure, facing the risks of component protection bypassing, app defrauding, and secret data leakage. These studies give us a clue to consider the manifest's security impact on the Android OS level. On the other hand, manifest files are constructed by app developers. Also, it is closely related to multiple core Android mechanisms, like permission registration and component management. Manifest file bridges apps and the system, and the vulnerable configuration data processing procedures against it could be exploited to break the Android systems. However, the correctness of manifest parsing and processing was neglected by most previous research.

Evil Twins Flaw. In this work, we systematically studied the Android manifest processing procedures and discovered a new category of vulnerabilities – the Evil Twins flaw. *That is, when the system processes the twin manifest elements (with the same name but different attributes), the defective data structure conversion procedure converts (merges) them into one item without considering the duplication issue. Meanwhile, indispensable element attributes are lost, resulting in system configuration inconsistency, i.e., potential vulnerabilities.*

For instance, while processing the app's permission declarations in Android 11, the system parses the twin `<permission>` elements and stores them in a `List<ParsedPermission>` structure for recording the app's own permission configurations. Then, the system further converts its twin `ParsedPermission` members and puts them into an `ArrayMap<permission-name, BasePermission>` structure for permission registration. Note that the `List` structure supports duplicate items. However, the `Key` part of `ArrayMap` must be unique. Therefore, during `List` to `ArrayMap`, the twin `<permission>` declarations are inconsiderately merged into one item, and some essential attributes are lost, such as permission protection level. As a result, the integrity of registered permission configurations is broken, and the app can stealthily obtain `dangerous` system permissions.

The fundamental cause of the Evil Twins flaw is that the ill-considered data structure conversion procedure with attribute loss leads to system configuration inconsistency. To exploit it, the attacker just needs to construct an app with a crafted manifest file, which is a pretty low bar. However, its security implications are severe, even leading to privilege escalation.

Flaw Detection. The Evil Twins flaw is a new category of logic vulnerabilities, not a single or random bug. Our preliminary investigation shows that the design of Android OS does not consider the correctness of manifest processing procedures comprehensively, especially the data structure conversions. Also, the Android internal implementations of manifest processing are quite complicated. Therefore, to detect the vulnerabilities related to the Evil Twins flaw, we designed an automated static data-flow analysis tool – TWINDROID, against the Android system code. Its high-level idea is to analyze the processing procedures of manifest files and identify the data structure conversion procedures with attribute loss.

After solving a series of technical challenges (e.g., state space explosion and abstract method implementation), we implemented a prototype of TWINDROID[1] and deployed it on the Android 11 &12 images of Pixel 3a (AOSP). Finally, it identified 47 suspicious processing methods associated with ill-considered data structure conversions. Our further manual checking and testing confirmed four vulnerabilities. Also, following the responsible disclosure policy, we reported our discoveries to the Android security team, and all of them have been confirmed and acknowledged. The consequences of these vulnerabilities are severe, which can lead to permission escalation (**Vul#1**: `CVE-2021-39695`, **Vul#2**: `CVE-2022-20392`, and **Vul#4**: `CVE-2023-20971`) and permission revoking prevention (**Vul#3**: `Android-ID-227340775`).

Since our discovered vulnerabilities exist in AOSP, they affect all downstream phone vendors. Especially, **Vul#1** and **Vul#2** have been acknowledged by Samsung [19], Huawei [9, 10], Honor [7,8], realme [20], and LG [21]. According to their security bulletins, around 271 models[2] are affected (150 for Samsung, 41 for Huawei, 20 for Honor, and 60 for realme).

The PoC attack demos of the above vulnerabilities can be found at `https://sites.google.com/view/eviltwins`.

Contributions. The main contributions of this paper are:

- *New security flaw.* We discovered a new category of vulnerabilities lying in the Android manifest processing procedures – the Evil Twins flaw. That is, the ill-considered data structure conversion procedure with attribute loss leads to system configuration inconsistency.

[1]Source code: `https://github.com/little-leiry/TwinDroid`.
[2]LG did not provide the affected device list on the security bulletin.

- *New analysis tool.* We designed a static data-flow analysis tool, TWINDROID, to discover the Evil Twins flaw. It can automatically analyze the manifest processing procedures of the Android OS to identify suspicious data structure conversions with attribute loss.

- *Real-world vulnerabilities.* With TWINDROID, we identified four vulnerabilities that can result in permission escalation and revoking prevention. All of them have been confirmed by Google, and CVE IDs were assigned.

Roadmap. The rest of this paper is organized as follows. Section 2 provides the necessary background of Android manifest processing. In Section 3, we give a motivation case and summarize the Evil Twins flaw. Section 4 introduces the design of our analysis tool – TWINDROID, and Section 5 summarizes the detection results. Section 6 analyzes the discovered vulnerabilities in depth. Section 7 discusses other relevant bugs and possible mitigation measures. Section 8 reviews related work, and Section 9 concludes this paper.

2 Background

Here we provide the necessary background of manifest file composition and processing procedures. Besides, we introduce the threat model used in this paper.

2.1 Manifest File and Elements

Every app must have a manifest file (`AndroidManifest.xml`). This file describes the essential configurations of the app to the Android build tools, Android OS, and Google Play, such as requested permissions, contained components, and required hardware & software features [2]. Such configurations are highly relevant to multiple Android critical mechanisms, such as permission and inter-component communication.

A manifest file is composed of a set of `XML` elements, and an element contains multiple attributes. For example, in Listing 1, it declares an activity that implements part of the app's visual user interface [12]. For this `<activity>` element, the `android:name` attribute can be treated as the *identifier* of it. In general, Android Studio does not allow app developers to use two same element types with the same name to avoid ambiguity. According to the Android development documentation [16], other identifiers include `android:tag` and `android:process` attributes.

```
<manifest ... >
  <activity
   android:name="com.example.TestActivity"
   android:exported="false">
  </activity>
...
</manifest>
```

Listing 1: Example of an element in a manifest file.

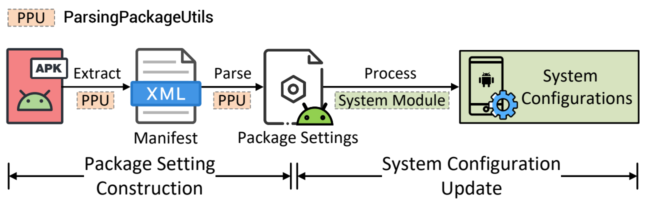


Figure 1: Manifest processing procedure.

Among all manifest elements, permission-related ones are closely related to security. On Android, sensitive resources are protected by permissions, which can be defined by the system (system permission) or third-party apps (custom permission) through the `<permission>` elements [5,23]. Permissions are divided into three primary protection levels: `normal`, `signature`, and `dangerous`. The system grants a `normal` permission to an app automatically. For the `signature` permission, the system grants it to the app that is signed by the same certificates as the app defining it. Both of these two permission types are granted when installing an app. Thus, they are also called install-time permissions. On the other hand, `dangerous` permissions are granted by users at runtime, so they are called runtime permissions. If an app requests a runtime permission, the system will present a dialog asking the user to decide, say allow or deny this request. Permissions can belong to a permission group. Runtime permissions are managed on a group basis. If an app has been granted a runtime permission, it can get other runtime permissions belonging to the same group (if requested) without user consent [15].

2.2 Manifest Processing Procedures

Here we overview the manifest file processing procedures of the Android OS. As illustrated in Figure 1, while installing a new app, the ParsingPackageUtils module (PPU for short) extracts the manifest file from the app package and parses each defined element to construct this package's settings. Further, based on these settings, other system modules update corresponding system configurations.

Package Setting Construction. The configurations defined in an app's manifest file are first extracted to construct the package settings. Specifically, for each manifest element, PPU invokes the corresponding method to parse it and stores the parsed data in the corresponding field of the `ParsingPackageImpl` class. Take Listing 1 as an example. During processing the `<activity>` element, PPU invokes the `parseActivityOrReceiver` method to obtain the corresponding `ParsedActivity` instance and add it to the `activities` field of `ParsingPackageImpl`. Such a field keeps the activity declarations of this app.

System Configuration Update. Next, other system modules further process the data stored in package settings and update corresponding system configurations. For example, based on the above package setting `activities`, ComponentResolver updates the system configuration `mActivities`, which maintains the activity registration information of the system.

2.3 Threat Model

In this study, we adopt a common threat model like previous Android system vulnerability research [24, 33, 37, 38]. An attacker builds a malicious app with the crafted manifest file and releases it on various app markets. Note that defining elements with the same type and name is not allowed by Android Studio – the official IDE for Android app development, and the compilation will report errors. However, it still can be achieved through APK repackaging easily [32].

The victim user may install this "apparently harmless" app on her phone. After running, this app exploits the system vulnerabilities (discovered by us) to conduct malicious actions, such as privilege escalation and data theft.

3 The Evil Twins Flaw

As mentioned in Section 1, the previous research [25, 31, 38] noticed the issue of developers' misconfigurations on manifest files, which will put apps at risk. Differently, our study does not focus on the developers' unsafe practices but on the correctness of manifest processing procedures of Android OS. Our investigation shows that the security implications of manifest files can affect the OS level. Exploiting a crafted manifest file, a malicious app can conduct various harmful actions, even stealthily obtaining `dangerous` system permissions. In this section, we present a concrete case discovered by us on Android 11. Further, we discuss the cause behind – the Evil Twins flaw.

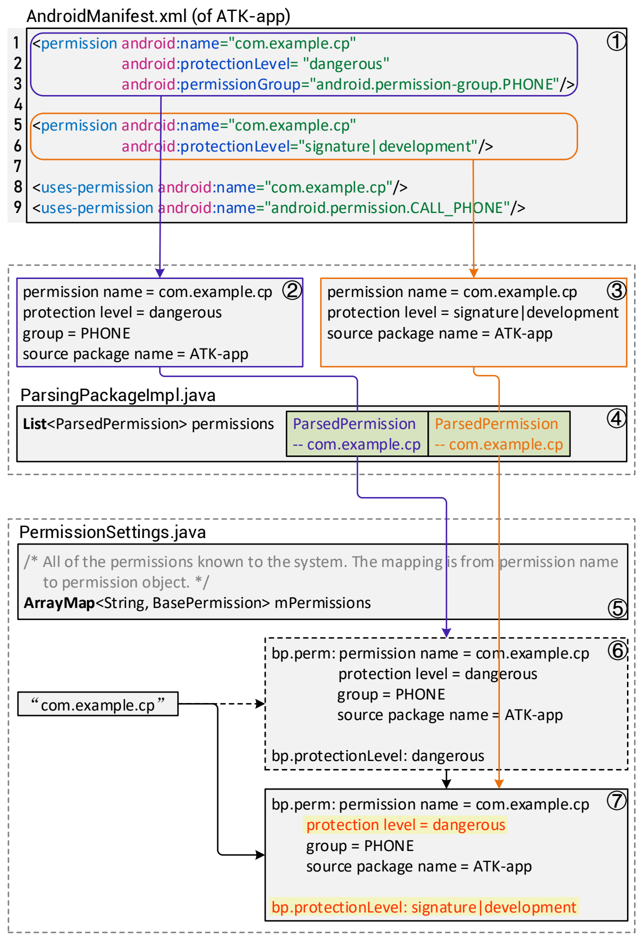
3.1 Motivation Case

Here we demonstrate a concrete case in which a malicious app with the crafted manifest file can achieve permission escalation. This exploiting case has been confirmed and acknowledged by the Android security team. They rated it as **High severity** (`Android-ID-209607944`) and assigned `CVE-2021-39695`.

Specifically, a malicious app, `ATK-app`, has a carefully constructed manifest file, as listed in Block ① of Figure 2. In this manifest, two custom permissions are defined with the same name (`com.example.cp`), say Lines 1 & 5, respectively. Their other attributes are different, including protection levels (`dangerous` v.s. `signature|development`[3]) and groups (`PHONE` v.s. no group). Besides, `ATK-app` requests the `com.example.cp` and `CALL_PHONE`[4] permissions, say Lines 8 & 9, respectively. At this moment, we create the status of ***twin***

[3]Appendix A.1 explains the reason for using `signature|development`.

[4]A dangerous system permission belonging to the `PHONE` group.


AndroidManifest.xml (of ATK-app)

```
1 <permission android:name="com.example.cp"
2              android:protectionLevel= "dangerous"
3              android:permissionGroup="android.permission-group.PHONE"/>
4
5 <permission android:name="com.example.cp"
6              android:protectionLevel="signature|development"/>
7
8 <uses-permission android:name="com.example.cp"/>
9 <uses-permission android:name="android.permission.CALL_PHONE"/>
```

①
② permission name = com.example.cp
protection level = dangerous
group = PHONE
source package name = ATK-app
③ permission name = com.example.cp
protection level = signature|development
source package name = ATK-app
ParsingPackageImpl.java
④ List<ParsedPermission> permissions
ParsedPermission -- com.example.cp
ParsedPermission -- com.example.cp
PermissionSettings.java
/* All of the permissions known to the system. The mapping is from permission name to permission object. */
ArrayMap<String, BasePermission> mPermissions
⑤
"com.example.cp"
⑥ bp.perm: permission name = com.example.cp
protection level = dangerous
group = PHONE
source package name = ATK-app
bp.protectionLevel: dangerous
⑦ bp.perm: permission name = com.example.cp
protection level = dangerous
group = PHONE
source package name = ATK-app
bp.protectionLevel: signature|development

Figure 2: Motivation case – data structure conversion.

custom permission declarations (i.e., with the same name but different attributes).

Once the victim user installs `ATK-app` on the phone running Android 11, it can stealthily obtain the `CALL_PHONE` permission (`dangerous` system permission) without user consent, say ***permission escalation***.

3.2 Vulnerability Analysis

Based on the source code of Android 11, we further explored the cause behind the above vulnerability and identified a new class of design flaws lying in manifest processing procedures.

Permission Element Parsing. As illustrated in Figure 2, during app installation, ParsingPackageUtils (PPU for short) parses each declared permission in the manifest file and constructs its corresponding `ParsedPermission` instance to store this permission's configuration data, including name, protection level, grouping, and source package name. Then, PPU adds these newly generated `ParsedPermission` items to `[List<ParsedPermission> permissions]` (i.e., Block ④), which stores all declared permission configurations of an app.

Since `List` can hold duplicate objects, in our exploit, the twin custom permission declarations (with the same name) correspond to two `ParsedPermission` instances in `permissions`, as shown in Block ② & ③.

Permission Registering. After that, based on `permissions`, PermissionManagerService (PMS for short) further updates `[ArrayMap<permission-name, BasePermission> mPermissions]` (i.e., Block ⑤), which stores the information of all declared permissions known to the system. This structure is relevant to permission-related operations, such as permission granting and revoking. Due to the uniqueness of the `Key` part of `ArrayMap`, during updating `mPermissions`, PMS will create or update the corresponding `BasePermission` instance `bp` based on the permission name (`Key` of `mPermissions`).

Therefore, in our case, the twin custom permission declarations correspond to one `BasePermission` instance, and some of their attributes have to be eliminated. Specifically, for the first declared `com.example.cp` permission in `permissions`, since such a permission name (i.e., `Key`) cannot be found in `mPermissions`, PMS will create a new `BasePermission` instance `bp` (as shown in Block ⑥) and then add a new item – `{com.example.cp` $\rightarrow$ `bp}` – to `mPermissions`. Next, for the second declared `com.example.cp` permission, since this permission name already exists in `mPermissions`, PMS will update the corresponding `bp`. According to the updating logic, PMS only updates `bp.protectionLevel` to `signature|development`. As a result, ***the protection levels stored in*** `bp.perm` ***and*** `bp.protectionLevel` ***are inconsistent***, as shown in Block ⑦.

Permission Status Update. After that, PMS iterates over the existing packages to update the granting status of their requested permissions. During this process, for each requested permission (name), PMS determines its type based on the value of its corresponding `bp.protectionLevel`.

In our case, for the request of the custom permission `com.example.cp` in `ATK-app` (Line 8 of Block ①), since its type is `signature|developement` (i.e., install-time permission), PMS will grant it to `ATK-app` automatically. For the request of the system permission `CALL_PHONE` (Line 9 of Block ①), since it is a `dangerous` permission, the system will perform runtime permission granting.

As mentioned in Section 2.1, granting runtime permissions is group-based. The system will first confirm whether `ATK-app` has obtained a granted `dangerous` permission belonging to the same group as the `CALL_PHONE` permission. To achieve this, it needs to construct the permission-group mapping for `ATK-app`'s each requested permission. However, the mapping data is obtained from the `bp.perm` corresponding to each requested permission name. Therefore, the granted `com.example.cp` will be treated as a `dangerous` permission belonging to the `PHONE` group (see `bp.perm` in Block ⑦). Further, since both `com.example.cp` and `CALL_PHONE` permissions are in the `PHONE` group, ***the*** `CALL_PHONE` ***permission will be granted to*** `ATK-app` ***automatically***.

3.3 Flaw Summary and Detection

Evil Twins Flaw. In the above motivation case, Android 11 uses a `List` data structure – `permissions` – for storing the raw permission configurations extracted from a newly installed app and an `ArrayMap` data structure – `mPermissions` – for storing the registered permission information maintained by the system. However, for the twin `<permission>` declarations, the data structure conversion from `List` to `ArrayMap` causes the loss of protection level attributes due to the uniqueness of the `Key` part in `ArrayMap`. It further leads to the inconsistency of permission protection levels and triggers permission escalation. Our further investigation shows that Android 12 does not have this vulnerability. In Android 12, permission registration information is stored in `ArrayMap<permission-name, Permission>`. The `Permission` data structure eliminates inconsistent permission protection levels. Comparing Android 11 with 12, we find that the Android OS design does not comprehensively consider the correctness of manifest processing procedures, especially the correlation between data structures used to store the manifest configurations. For the security-related twin elements, if their essential attributes are lost during data structure conversions, it may break the integrity of system configurations, resulting in an exploitable vulnerability.

To generalize, ***when the system processes the twin manifest elements, the defective data structure conversion procedure converts (merges) them into a single item without considering the duplication issue. As a result, the indispensable element attributes are lost, leading to system configuration inconsistency***. The twin elements mean they have the same identifier (e.g., name) but different attributes. Therefore, we call such a security issue the Evil Twins flaw. Also, it can be exploited by a malicious app with the crafted manifest file and affects various system configurations to compromise the security mechanisms of the Android OS.

Automated Flaw Detection. Through the above analysis, we can find that the Evil Twins flaw is a new category of logic vulnerabilities, not a single or random bug. The app manifest file is a practical attack surface, which (malicious) app developers can control. Besides, it bridges apps and the system, correlating multiple core management mechanisms of the Android OS, as demonstrated in Section 2. Given the significance and complexity of the manifest processing procedures, we believe the discovered motivation case is just the tip of the iceberg. Therefore, we need to design an automated detection tool to detect Evil Twins flaw-related vulnerabilities.

The detection tool should be able to identify the data structure conversions with attribute loss by analyzing Android OS's manifest processing procedure implementations. Particularly, it should cover all processing procedures of all manifest elements with all attributes. Considering the diversity of manifest element types and processing procedures, we decided to adopt the strategy of static analysis for code coverage.

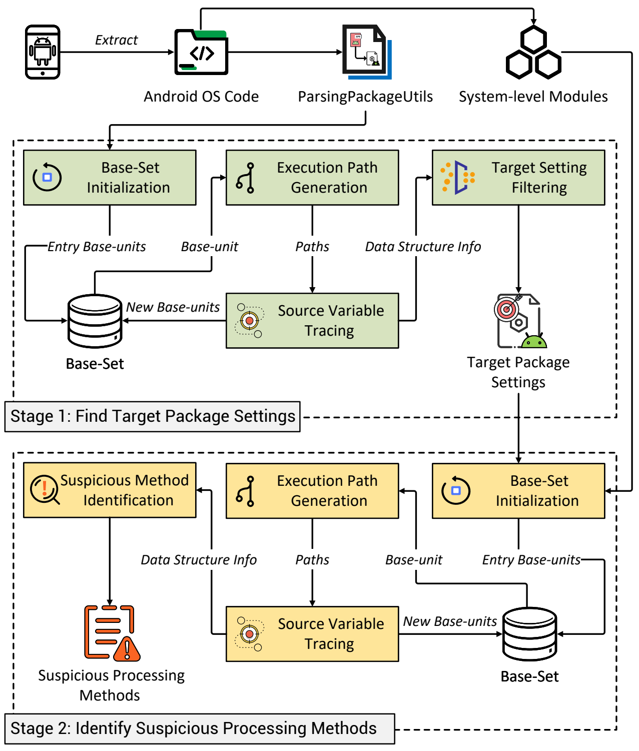


Figure 3: Overview of TWINDROID.

4 Design of TwinDroid

This section introduces the detailed design of our static analysis solution – TWINDROID. Its high-level idea is to analyze the data processing procedures of manifest files and identify the data structure conversion procedures with attribute loss. In practice, since static analysis does not execute the code, it is difficult to determine whether such a conversion can finally lead to exploitable data inconsistency. Therefore, the output of TWINDROID is a candidate set of suspicious methods processing the manifest configuration data. We further need to confirm the Evil Twins flaw manually based on the obtained method information and the Android source code (see Section 5.2). As illustrated in Figure 3, TWINDROID contains the following two main stages.

Stage 1 *Find target package settings.* Firstly, TWINDROID needs to find the package settings supporting duplicate elements, like `[List<ParsedPermission> permissions]` shown in Block ④ of Figure 2. Package settings are constructed based on app manifest files, which malicious developers can control.

Stage 2 *Identify suspicious processing methods.* Next, TWINDROID identifies the suspicious processing methods which access the target settings and cause attribute loss due to data structure conversion.

4.1 Find Target Package Settings

As mentioned in Section 2.2, ParsingPackageUtils (PPU) parses each element declared in an app's manifest file to construct the package settings. As discussed in Section 3.3, the Evil Twins flaw exploits the ill-considered data structure conversions during processing twin elements. Therefore, we need to find the package settings that can hold duplicate elements. To achieve it, TWINDROID performs a path-sensitive data flow analysis against the procedure of parsing app manifest files. The approach is to trace the propagation of configuration data (elements of manifest files) and obtain the corresponding data structure types over the flow. The high-level analysis logic of this stage is illustrated in Figure 3 - Stage 1. Here we explain its core modules.

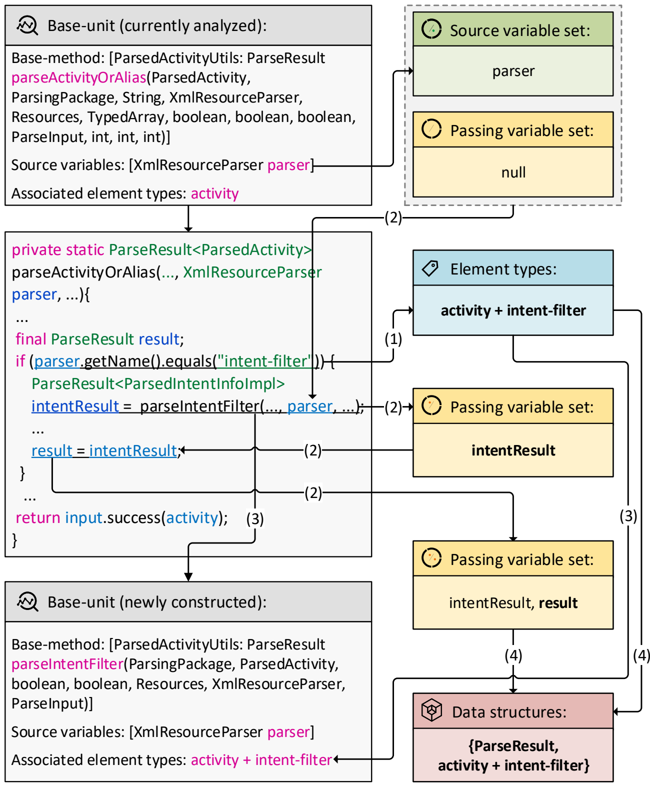
Base-Set Initialization. TWINDROID launches the analysis with the initialization of Base-Set, a set of base-units that need to be analyzed to trace the inter- & intra-procedural variable propagation. Each base-unit is composed of 1) a base-method to be analyzed, 2) a set of source variables to be traced, and 3) element types associated with this base-method. Different element types correspond to different parsing methods. For example, the `<permission>` element corresponds to the `parsePermission` method. Thus, the element type associated with `parsePermission` is "permission". Base-Set is initialized by the entry base-units, which specify the first base-method to be analyzed and the first source variables to be traced. TWINDROID locates entry base-units to activate the whole analysis.

In this stage, the located entry base-units should represent the initial step of parsing a manifest file. Therefore, TWINDROID looks for entry base-units in the PPU module. An entry base-unit contains the following values. Note that, there may be multiple entry base-units meet these requirements.

- Base-method: the method invoking the `openXmlResourceParser` method [13] with `AndroidManifest.xml` as an input parameter.
- Source variables: the `XmlResourceParser` instances returned from the `openXmlResourceParser` invocations.
- Associated element types: `null`. Since the entry base-unit is the beginning of parsing a manifest file, it does not have associated element types.

Execution Path Generation. For the base-method in each base-unit, TWINDROID generates its execution paths for the subsequent analysis. Particularly, TWINDROID first constructs its control flow graph (CFG), and then, performs the depth-first search against this CFG to generate execution paths.

Source Variable Tracing. After that, for each execution path, TWINDROID traces the propagation of source variables. It records the data structures that the manifest data (stored in source variables) finally reaches. As summarized in Figure 4, the tracing procedure contains four main steps, as follows.


Base-unit (currently analyzed):

Base-method: [ParsedActivityUtils: ParseResult parseActivityOrAlias(ParsedActivity, ParsingPackage, String, XmlResourceParser, Resources, TypedArray, boolean, boolean, boolean, ParseInput, int, int, int)]

Source variables: [XmlResourceParser parser]

Associated element types: activity

```
private static ParseResult<ParsedActivity>
parseActivityOrAlias(..., XmlResourceParser
parser, ...){
 ...
 final ParseResult result;
 if (parser.getName().equals("intent-filter")) {
    ParseResult<ParsedIntentInfoImpl>
    intentResult = parseIntentFilter(..., parser, ...);
    ...
    result = intentResult;
 }
  ...
 return input.success(activity);
}
```

Base-unit (newly constructed):

Base-method: [ParsedActivityUtils: ParseResult parseIntentFilter(ParsingPackage, ParsedActivity, boolean, boolean, Resources, XmlResourceParser, ParseInput)]

Source variables: [XmlResourceParser parser]

Associated element types: activity + intent-filter

Figure 4: Diagram of source variable tracing.

(1) *Element type recording.* To construct the correspondence between the element's configuration data and its final reached data structures, TWINDROID keeps a record of the associated element type during tracing source variables. The manifest file is multi-layered, and an element may have sub-elements. Thus, the recorded element type is the combination of the associated element type (of the currently analyzed base-unit) and the involved element type on the execution path. For example, as shown in Figure 4, the recorded element type is "activity + intent-filter".

(2) *Source variable propagation.* TWINDROID maintains a source variable set and a passing variable set to trace the source variable propagation along with the execution path.

- Source variable set: retain source variables and their equivalent objects. This set keeps the original data all the time to avoid missing them (due to variable redefinition) during the tracing.
- Passing variable set: retain the variables that the manifest data (stored in source variables) passes into.

When the variable in the above two sets passes through a statement that can trigger variable propagation, TWINDROID will adjust the corresponding variable set's state. There are two kinds of qualified statements: invocation and assignment. They associate with two types of variables:

Table 1: Variable propagation types of invocation and assignment statements.

Statement	Variable Propagation Type	Code Example	Flow
Invocation	From the callee's instance to its parameter.	`a.toArray(b);`	a → b
	From the callee's one parameter to another.	`System.arraycopy(a, 0, b, 1, a.length);`	a → b
	From the callee's parameter to its instance.	`a.put(b,c);`	b, c → a
Assignment	From the variable at the right of the equal sign to the variable at the left.	`a = b + c;`	b, c → a
		`a = Math.max(b, c);`	b, c → a
	The same as the invocation statement.	`z = a.addAll(b);`	b → a

- Inflow variable: the variable flowing into an invocation or assignment statement. Its number can be one or more.
- Outflow variable: the variable flowing out an invocation or assignment statement. Its number only can be one.

Along with the execution path, the data propagates from an inflow variable to another outflow variable. Table 1 lists the basic variable propagation types. Combining the inflow position of the inflow variable with the particular propagation type, TWINDROID can determine its outflow variable and further adjust the corresponding variable set[5].

(3) *New base-unit construction.* When the traced variable is passed into a method as a parameter, TWINDROID will determine whether to construct a new base-unit based on the existence of the method return value. This new base-unit will be used for further inter-procedural variable tracing. The return value of a method reflects the propagation result of its parameters to some extent. Therefore, for the invoked method without a return value, TWINDROID will construct a new base-unit. For the one with a return value, TWINDROID will construct a new base-unit only for some special methods we are interested in. These special methods are determined according to the Android source code logic. The composition of a newly constructed base-unit is as follows.

- Base-method: the invoked method.
- Source variables: the variables passed into the invoked method as parameters.
- Associated element types: the element types recorded in step (1).

In addition, to avoid repetitious analysis, the base-units with the same base-method, source variables, and associated element types will only be constructed once.

(4) *Data structure saving.* As defined before, the passing variable set retains the variables that the manifest data (stored in source variables) passes into. Therefore, when the propagation of the variable in this set finishes, TWINDROID saves its data structure and the corresponding element type. If this variable is a class's field, TWINDROID will save this field's complete information, including its name, type (data structure), and source class name. For example, as shown in Block ④ of Figure 2, for the `permissions` field in the `ParsingPackageImpl` class, the saved information is "permissions + List<ParsedPermission> + ParsingPackageImpl".

[5]Appendix A.2 illustrates the specific adjustment rules.

Target Setting Filtering. When all base-units in Base-Set are analyzed, according to the saved data structures, TWINDROID filters the package settings supporting duplicate elements, like `[List<ParsedPermission> permissions]`. The data structure should satisfy the following three requirements.

- This structure represents the type of field belonging to the `ParsingPackageImpl` class. As mentioned in Section 2.2, this class stores the package settings.
- This structure can hold duplicate items, such as `List`, `Array`, `Stack`, `Queue`, and the `Value` part of `Map`.
- The stored object (manifest element) in this structure has multiple attributes, and one of them is related to an element identifier, e.g., `ParsedPermission`.

4.2 Identify Suspicious Processing Methods

After obtaining the target package settings, in the second stage, we try to identify the processing methods accessing these settings and causing attribute loss due to the ill-considered data structure conversion. To achieve this, similarly to Section 4.1, TWINDROID performs a data flow analysis against the processing procedures of the configuration data stored in target package settings. As illustrated in Figure 3 - Stage 2, the overall analysis procedure is similar to Stage 1, and the following steps are adjusted.

Base-Set Initialization. In this stage, the entry base-unit should represent the initial step of accessing the configuration data stored in the target package settings. These settings are the fields of the `ParsingPackageImpl` class. Field can be accessed by other classes directly or through invoking its `getter`, which is the method for reading a variable value [1]. Therefore, at first, TWINDROID obtains the `getters` corresponding to the target package settings. Then, TWINDROID iterates through all system-level modules of the Android OS to locate the entry base-units. The composition of each base-unit is as follows.

- Base-method: the method that accesses the target package settings directly or by invoking their getters.
- Source variables: the variables that store the configuration data from the target package settings.
- Associated element types: the element types corresponding to the accessed target package settings.

Source Variable Tracing. There are two adjustments for step (4) – *data structure saving*, as follows.

- Apart from saving information when variable propagation termination, since data removal also causes attribute loss, TwinDroid saves the related data structures. Suppose the variable in the passing variable set involves data removal, such as being passed in the `remove` method as a parameter (i.e., `b.remove(c)`). In that case, TwinDroid will save the data structure of the variable with information loss (i.e., b's data structure).
- When saving data structures, apart from the element types corresponding to them, TwinDroid also holds their corresponding processing method, which is the currently analyzed base-method or its farthest ancestor on the invocation chain.

Suspicious Method Identification. For the obtained data structures, TwinDroid eliminates the ones that can hold duplicate objects and do not involve data removal. To the remaining data structures, TwinDroid treats their associated processing methods as suspicious. Finally, TwinDroid outputs these methods' information, including methods' names, associated element types, and saved data structures.

For the suspicious processing methods identified by TwinDroid, we will take an in-depth analysis to check whether their affiliated data processing procedures can further lead to exploitable configuration inconsistency vulnerabilities.

5 Evaluation and Results

To evaluate the security implications of the Evil Twins flaw, we conducted a real-world evaluation. Here we summarize our evaluation setup and results.

5.1 Implementation and Experiment Setup

Implementation. We implemented a prototype of TwinDroid with around 6,100 lines of Java code. To facilitate the analysis of TwinDroid, we integrated Soot [22], a framework for analyzing and transforming Java code (including Android). Our implementation is based on the Jimple IR – Soot's primary IR (intermediate representation). To improve the analysis efficiency and accuracy of TwinDroid, we deployed a series of optimization strategies on it, such as path explosion avoiding and incorrect path pruning. More details are provided in Appendix A.3.

Execution Environment. TwinDroid was deployed on a powerful server with Intel Xeon Gold 6226R CPU @ 2.90GHz and 256G RAM. The OS of the execution environment is Ubuntu 20.04.3 LTS.

5.2 Results and Findings

Result Overview. In evaluation, we extracted the DEX files under the `/system/framework` directory from the Android 11 (RQ3A.211001.001) & 12 (SP2A.220505.002) images of Pixel 3a [18] for analysis. The execution time was around 1.5h per image. TwinDroid discovered 12 target package settings supporting duplicate elements (Stage 1). These settings correspond to 17 element types. Further, it identified 47 suspicious processing methods from 329,846 system methods (Stage 2). Table 2 lists the evaluation results[6] that can cause security issues.

We manually checked the obtained suspicious processing methods and found that six of them were false positives. When constructing the package settings `usesPermissions` and `attributions`, ParsingPackageUtils checks them in real-time to avoid storing duplicate configuration data (with the same element identifier). Thus, these package settings cannot support duplicate elements in practice, but TwinDroid marked them as target package settings. Further, TwinDroid marked the methods processing these settings as suspicious incorrectly. After excluding the false positives, we finally obtained 41 suspicious processing methods corresponding to 8 target package settings and 10 element types.

Inconsistency Confirmation. As discussed in Section 4, due to the limitation of static analysis, we need to manually confirm whether the suspicious processing methods can cause risky system configuration inconsistency based on the obtained method information and the Android source code. During this process, for the suspicious processing methods associated with the same element types, we follow the four analysis steps: 1) figure out each method's functionality based on its name and context information (invocation chains); 2) based on the Android source code, locate the code block of data structure conversion procedures to confirm the lost attributes in each method; 3) find out correlative items (methods' functions or lost attributes) to determine whether they cause system configuration inconsistency; and 4) build PoC app to verify the discovered potential vulnerability.

Findings. Following the above procedure, we further analyzed the identified 41 suspicious processing methods in depth to confirm whether there exists the Evil Twins flaw. Note

[6]Due to the page limitation, the complete results are provided at https://github.com/little-leiry/TwinDroid/blob/main/Results.pdf.

Table 2: Discovered suspicious processing methods† with security issues.

No.	Suspicious Processing Method	Target Package Setting	Element Type	Vul#	Ver.‡
1	`addAllPermissions`	`List<ParsedPermission> permissions`	`permission`, `permission-tree`	**1, 2**	11
2	`addAllPermissionsInternal`	`List<ParsedPermission> permissions`	`permission`, `permission-tree`	**2**	12
3	`revokeRuntimePermissionsIfGroup-Changed`	`List<ParsedPermission> permissions`	`permission`, `permission-tree`	**2**	11
4	`revokeRuntimePermissionsIfGroup-ChangedInternal`	`List<ParsedPermission> permissions`	`permission`, `permission-tree`	**2**	12
5	`hasPermission`	`List<ParsedPermission> permissions`	`permission`, `permission-tree`	**3, 4**	11, 12
6	`addActivitiesLocked`	`List<ParsedActivity> activities`	`activity`, `activity-alias`	bug	11, 12
7	`queryIntentActivitiesInternalBody`	`List<ParsedActivity> activities`	`activity`, `activity-alias`	bug	12
8	`addReceiversLocked`	`List<ParsedActivity> receivers`	`receiver`	bug	11, 12
9	`addServicesLocked`	`List<ParsedService> services`	`service`	bug	11, 12

†: The complete evaluation results are detailed at `https://github.com/little-leiry/TwinDroid/blob/main/Results.pdf`.
‡: The version of Android OS.

that, as shown in Table 2, multiple elements may correspond to the same package settings. For example, in Item 1, both the `<permission-tree>` and `<permission>` elements correspond to the `[List<ParsedPermission> permissions]` setting. That is to say, if the names of these two elements are the same, they also will become the twin `ParsedPermission` instances (with the same name) in `permissions`. Therefore, in this case, the risk of the Evil Twins flaw also exists.

Eventually, we identified four exploitable inconsistency cases, say security vulnerabilities. Though their concrete exploiting approaches are different, their causes can be classified into the same problem – *the data structure conversion with attribute loss*. Also, we reported our findings to the Android security team, and all of them have been confirmed. The following section will discuss these vulnerabilities and their practical exploits.

6 Vulnerabilities and Exploits

This section discusses the discovered vulnerabilities related to the Evil Twins flaw and demonstrates their exploits. Note that, if not otherwise specified, this section's code logic analysis is based on Android 12. The PoC attack demos can be found at `https://sites.google.com/view/eviltwins`.

6.1 Break Permission Protection Levels

We re-discovered the motivation case described in Section 3.1 based on Item 1 of Table 2. Since its exploit and cause have been demonstrated in Section 3, here we only summarize this vulnerability under the framework of Evil Twins flaw.

Vul#1: *During processing the twin `<permission>` elements, due to the data structure conversion with protection level loss, the protection levels held by the system become inconsistent, further resulting in permission escalation.*

6.2 Break Permission-Group Mapping

In Table 2, Items 2 & 4 show the `addAllPermissionsInternal` and `revokeRuntimePermissionsIfGroupChangedInternal` methods assess the `[List<ParsedPermission> permissions]` setting. The former method is for registering the app-declared permissions to the system during app installation or updating. The system performs most permission-related management operations based on the registration information. The latter is for revoking runtime permissions if their groups are changed during the app updating, which is not allowed by Android [17]. Since runtime permission granting is group-based, if a granted runtime permission's group changes, the system will revoke its grant from apps to avoid automatically getting other permissions in the new group. However, we find that:

Vul#2: *During processing the twin `<permission>` elements, due to the data structure conversion with grouping loss, the permission-group mapping relationship becomes inconsistent, further resulting in permission escalation.*

Exploit. The adversary creates a malicious app `app-etf2` which declares twin `dangerous` custom permissions with the same name – `com.example.cp` – in its manifest file, as shown in Listing 2. They are assigned to the `STORAGE` and `PHONE` groups, respectively. Also, `app-etf2` requests this custom permission and the `CALL_PHONE` permission (`dangerous` system permission belonging to the `PHONE` group).

```
1  <permission
2    android:name="com.example.cp"
3    android:protectionLevel="dangerous"
4    android:permissionGroup="android.
         permission-group.STORAGE" />
5  <permission
6    android:name="com.example.cp"
7    android:protectionLevel="dangerous"
8    android:permissionGroup="android.
         permission-group.PHONE" />
9
10 <uses-permission
11   android:name="com.example.cp" />
12 <uses-permission
13   android:name="android.permission.
         CALL_PHONE" />
```

Listing 2: Manifest file (part) of `app-etf2`.

Besides, the adversary prepares an updated version – `app-etf2-up`. The only difference is the first declaration of `com.example.cp` (Lines 1-4) is removed from the manifest.

▶ The victim user installs `app-etf2` on her phone. While the app is running, it presents a dialog to ask the user to grant it the permission belonging to the `STORAGE` group, and she allows it. Then, the user updates this app. During this procedure, the `com.example.cp` is put into the `PHONE` group. However, its granting state is not revoked. Further, *`app-etf2` **obtains the `CALL_PHONE` permission automatically without user consent***.

Cause Analysis. As illustrated in Figure 5, when installing `app-etf2`, ParsingPackageUtils (PPU) parses its declared permissions (Lines 1-4 & 5-8 of Listing 2), and further, PermissionManagerService (PMS) registers them in the system. Specifically, PMS obtains the permission data from `[List<ParsedPermission> permissions]` and puts the processed data into `[ArrayMap<String, Permission> mPermissions]` (the {`permission name` → `permission configurations`} mapping). For the twin `com.example.cp` permissions, *`mPermissions` retains the former's data* (Lines 1-4). The latter is neglected because PMS treats it as a duplicate declaration. When granting `com.example.cp`, the system gets its configuration data from `mPermissions`. Therefore, `com.example.cp` will be treated as a `dangerous` permission belonging to the `STORAGE` group.

Similarly, when installing the updated version – `app-etf2 -up`, PPU parses this new package. Then, PMS registers its declared permission in the system. At this moment, the group `com.example.cp` belonging to is updated to `PHONE`. Next, PMS judges whether this app update involves group changing of runtime permissions. To achieve this, PMS obtains the permission configurations from `permissions` of previously installed `app-etf2` and puts the processed data into `[ArrayMap<String, String> oldPermissionNameToGroupName]`, which reflects the {`permission name` → `group name`} mapping. For the declared twin `com.example.cp` permissions in `app-etf2`, *`oldPermissionNameToGroupName` retains the latter's data without duplication detection* (Lines 5-8). Therefore, PMS believes that the group `com.example.cp` belonging to does not change (always be `PHONE`) and will not revoke the existing permission granting.

Impact. This vulnerability has been acknowledged by Google with rating **High severity** (`Android-ID-213323615`), and a CVE ID has been assigned: `CVE-2022-20392`.

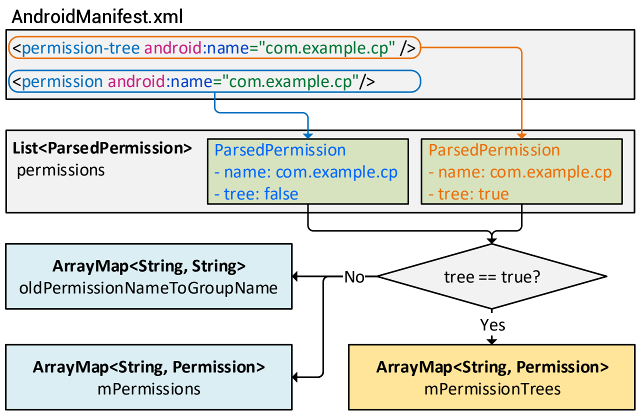

AndroidManifest.xml
<permission-tree android:name="com.example.cp" />
<permission android:name="com.example.cp"/>
List<ParsedPermission> permissions
ParsedPermission
- name: com.example.cp
- tree: false
ParsedPermission
- name: com.example.cp
- tree: true
tree == true?
No
Yes
ArrayMap<String, String> oldPermissionNameToGroupName
ArrayMap<String, Permission> mPermissions
ArrayMap<String, Permission> mPermissionTrees

Figure 5: Data structure conversion of `<permission-tree>` and `<permission>` elements.

6.3 Break Permission Registration Status

In Item 5 of Table 2, the `hasPermission` method accesses `[List<ParsedPermission> permissions]`. This method is for judging whether an app declares a permission (or permission tree[7]) with the given name. This judgment result is related to updating permission (or permission tree) registration data in the system and revoking granted permissions.

The Android OS maintains two `Map`-type data structures – `[ArrayMap<String, Permission> mPermissions]` and `[ArrayMap<String, Permission> mPermissionTrees]` – to store the permission and permission tree registration information, respectively. As illustrated in Figure 5, when parsing the manifest file of an installed app, for both `<permission>` and `<permission-tree>` elements, PPU creates their corresponding `ParsedPermission` instances and puts them

[7]An app can declare the base name for a tree of permissions through the `<permission-tree>` element. This app takes ownership of all names within the tree. Note that this element does not declare a permission itself, only a namespace where further permissions can be placed [14].

into this app's [List<ParsedPermission> permissions]. The difference is that, for <permission-tree>, the tree field of its ParsedPermission instance is true. Next, PMS further processes the data in [List<ParsedPermission> permissions] and puts them into [ArrayMap<String, Permission> mPermissions] (if tree == false) or [ArrayMap<String, Permission> mPermissionTrees] (if tree == true).

During app update, if the updated version no longer declares some permission (i.e., this permission's definition is removed from the system), PMS will revoke the grants of this permission from the corresponding apps and remove its Permission instance from mPermissions. To achieve this, PMS invokes hasPermission to judge whether the updated app still declares certain permission. However, we find that:

Vul#3: *Due to the tree attribute loss during the data structure conversion (method invocation), the <permission-tree> element is confused with the <permission> element, further resulting in the incorrectness of permission registration status in the system.*

Exploit. The adversary prepares a malicious app app-etf3 declaring a permission tree and a normal custom permission with the same name (com.example.cp), as shown in Listing 3. This app also requests the com.example.cp permission. Note that, declaring different element types with the same name is allowed by Android Studio.

```
1 <permission-tree
2   android:name="com.example.cp" />
3 <permission
4   android:name="com.example.cp"
5   android:protectionLevel="normal" />
6
7 <uses-permission
8   android:name="com.example.cp" />
```

Listing 3: Manifest file (part) of app-etf3.

The adversary also prepares an updated version of this app – app-etf3-up. It removes the permission declaration (Lines 3-5) from its manifest file.

▶ The victim user installs app-etf3, and the system grants it the normal-level com.example.cp permission automatically. Then, she updates app-etf3 with app-etf3-up. After updating, the original declaration of com.example.cp has been removed from the system. However, ***this permission is not revoked from** app-etf3-up**, and the system still holds its registration information***.

The residue of permission registration data will also lead to potential app installation failure. The Android OS does not allow an app to declare a permission with the same name as an existing permission registered in the system, unless this app and the app defining this permission are signed with the same certificate [6]. Thus, if another app re-declares the com.example.cp permission, but with a different certificate, its installation will not be allowed.

Cause Analysis. As mentioned before, the ParsedPermission members of [List<ParsedPermission> permissions] can be constructed through two types of elements – <permission> and <permission-tree>. Their corresponding ParsedPermission instances can be distinguished through the value of ParsedPermission.tree. However, the hasPermission method only judges whether there exists a ParsedPermission instance with the given name in permissions, then returns true / false. That is, during the judgment, *the tree label is not touched, and its carried necessary data is lost*. Also, the boolean-type return value cannot indicate if the true / false corresponds to a permission or a permission tree.

During app updating, since the permissions of app-etf3-up still holds a ParsedPermission instance with the name com.example.cp (from the permission tree), PMS misjudges that it still declares the com.example.cp permission. Further, PMS will not revoke the granted com.example.cp from app-etf3-up and not remove the corresponding Permission instance from mPermissions.

Impact. This vulnerability has been acknowledged by Google with rating **Low severity** (Android-ID-227340775).

6.4 Break Permission Granting Status

During the cause analysis of **Vul#3**, we discovered another vulnerability by chance. It also exists in the processing procedures against the <permission> and <permission-tree> elements. Though its cause cannot be boiled down to the defective data structure conversions directly, its exploits also could be achieved through the twin elements.

Recall the case of permission declaration removal described in Section 6.3. Similarly, if the original definition of a permission tree is removed during the app update, PMS will remove its corresponding Permission instance from [ArrayMap<String, Permission> mPermissionTrees].

Besides, the app uninstallation may also involve the removal of permission (or permission tree) declarations. It will further trigger the adjustment of permission (or permission tree) registration data and granting status. Specifically, during the app uninstallation, if this uninstalled app declares some permissions or permission trees, PMS will remove their corresponding Permission instances from mPermissions or mPerimissionTrees and revoke the corresponding permission grants. However, we find that:

Vul#4: *Due to the inconsistency of data structure member adding and removing during processing twin elements, the <permission-tree> element is confused with the <permission> element, further resulting in the incorrectness of permission granting status in the system.*

Exploit. The adversary constructs two malicious apps – `app-etf4-d` and `app-etf4-r` (their signing certificates can be the same or not). The former app defines a permission tree and a `normal` permission with the same name – `com.example.cp`, as shown in Listing 4. The latter app requests the `com.example.cp` and `CALL_PHONE` permissions.

```
<permission-tree
  android:name="com.example.cp" />
<permission
  android:name="com.example.cp"
  android:protectionLevel="normal" />
```

Listing 4: Manifest file (part) of `app-etf4-d`.

The adversary also prepares an updated version of `app-etf4-d`, named `app-etf4-d-up`. This updated version re-declares a `dangerous` `com.example.cp` permission and puts it into the `PHONE` group, as shown in Listing 5.

```
<permission
  android:name="com.example.cp"
  android:protectionLevel="dangerous"
  android:permissionGroup="android.
      permission-group.PHONE"/>
```

Listing 5: Manifest file (part) of `app-etf4-d-up`.

▶ The victim user installs both `app-etf4-d` and `app-etf4-r` on her phone. At this moment, `app-etf4-r` is granted the `normal` `com.example.cp` permission automatically. Then, `app-etf4-d` crashes on purpose frequently and induces the user to uninstall the current version and re-install an updated one. After `app-etf4-d-up` is installed, ***`app-etf4-r` obtains the `CALL_PHONE` permission silently without user consent***.

Cause Analysis. When installing `app-etf4-d`, PMS creates two `Permission` instances for the declared permission and permission tree. Then, these two instances are added to [`ArrayMap<String, Permission> mPermissions`] and [`ArrayMap<String, Permission> mPermissionTrees`], respectively. Since the declared permission and permission tree have the same name, both `mPermissions` and `mPermissionTrees` have a `Permission` instance corresponding to the name `com.example.cp`.

During uninstalling `app-etf4-d`, PMS updates the permission and permission tree registration data. It first should adjust `mPermissionTrees`. However, during this procedure, *PMS incorrectly removes the `Permission` instance (corresponding to `com.example.cp`) from `mPermissions`*. Next, PMS adjusts `mPermissions`. At this time, there no longer exists a `Permission` instance corresponding to `com.example.cp` in `mPermissions`. Therefore, PMS will mistakenly assume that the `com.example.cp` permission is not registered in the system. Then PMS will not revoke it from `app-etf4-r` even if its original definition and permission registration have been removed from the system.

After re-installing the updated version of `app-etf4-d`, say `app-etf4-d-up`, the `com.example.cp` permission is re-added to the system as a `dangerous` permission belonging to the `PHONE` group. Therefore, `app-etf4-r` gets the `CALL_PHONE` permission in the same group automatically.

Discussion. After analyzing the Android source code, we found that only the app uninstallation can trigger PMS incorrectly removing the `Permission` instance from `mPermissions`. Therefore, the exploit of this vulnerability cannot be completed through the app update.

Impact. This vulnerability has been acknowledged by Google with rating **Moderate severity** (`Android-ID-225880325`). They commented that: *"We applied a -1 modifier due to this vulnerability requiring non-trivial and unlikely user actions."* A CVE ID has been assigned: `CVE-2023-20971`.

7 Discussion

Here, we discuss other Evil Twins flaw-related issues that do not directly affect the Android OS security. Besides, we propose some possible improvements to the Android OS and discuss the limitations of our work.

7.1 Other Relevant Bugs

As summarized in Section 5.2, TwinDroid finally identified defective data structure conversions related to 10 element types. However, all our reported vulnerabilities are permission-related. The reason is that the permission mechanism is closely tied to system security. Permission-related vulnerabilities often have direct exploit targets, such as achieving privilege escalation. For other element types, part of them cannot result in exploitable system configuration inconsistencies because the operation sites of their suspicious methods are different. For example, both `preparePackageLI` and `addAllPermissionGroupsInternal` methods correspond to the `<permission-group>` element and are marked as suspicious. The former method is for blocking the installation of the app declaring an existing permission group, and the latter is for permission group registration after app installation.

The rest element types have limited effects. TwinDroid discovered a case that only affects app security via Items 6 & 7 in Table 2. In brief, while processing the twin `<activity>` elements defined in apps, due to the data structure conversion with protection-related attribute loss, the {`app component` → `protection status`} mapping becomes inconsistent, further resulting in bypassing app's component protection.

In previous work, Aafer et al. [25] first reported a Samsung app's bug, which declares duplicate receivers with different protections. Since the second declaration of the component name replaces the first one, the first receiver's permission protection is bypassed, making the app vulnerable. They attributed this problem to the misconfiguration introduced by vendor customization. In fact, the corresponding manifest processing procedures are also vulnerable. Back to our case, the activity declarations are converted from `List<ParsedActivity>` into `ArrayMap<ComponentName, ParsedActivity>` and `ArrayMap<String, F[]>` with protection loss, resulting in inconsistent component protection status. Exploiting the Evil Twins flaw, the component protection can be bypassed whether it is put on the first or second component. Appendix A.4 provides a detailed analysis. We reported this issue to the Android security team. After assessing, they did not recognize it as a vulnerability and labeled it **Won't Fix (Intended Behavior)** – `Android-ID-237404762`. They commented that: *"If a component is intended to be protected, it is the developer's responsibility to ensure that there are no declarations to the contrary."*

Although our discovered vulnerabilities are permission-related, the Evil Twins flaw demonstrates the security risk in manifest processing procedures. Its potential impacts will be widespread if other security-related processing methods or element types are introduced with Android evolution.

7.2 Mitigation

Google's Fixes. Google has released the patches of **Vul#1 & #2**. For **Vul#1**, Google adjusted the update logic mentioned in Section 3.2 to eliminate the inconsistent permission protection levels stored in `bp.perm` and `bp.protectionLevel` [3]. For **Vul#2**, Google blocked the installation of apps declaring duplicate permissions with different protection levels or groups [4]. Google's fixes only introduced small code changes to block specific steps of our reported attack flows, not covering **Vul#3 & #4** (exploiting permission and permission tree). Android 12 and later versions still face the security threats of the Evil Twins flaw.

Lessons to Learn. The Evil Twins flaw-related issue can cause severe implications. For Google and OEMs, to eliminate this flaw thoroughly and prevent its recurrence, they need to follow a general principle: *avoid potential information loss during processing configuration data*. It can be achieved as follows:

(1) *Android OS Design.* In the design or customization of the Android OS, the system should perform a duplication check when installing an app to ensure the uniqueness of configuration data. Besides, the same type of data should not be stored in different locations to avoid inconsistency, and different types of data should not be stored in the same location to avoid information loss.

(2) *App Development and Distribution.* Android Studio should deploy a more comprehensive duplication check covering different element types. Google Play and third-party app markets also should check app manifest files to forbid the distribution of apps declaring twin elements.

For app developers, they should avoid defining multiple elements with the same name, even for different element types. Such elements may be an attack vector for exploiting the Evil Twins flaw.

7.3 Limitations

Suspicious Methods. To detect the Evil Twins flaw, we designed TwinDroid to analyze the Android system code statically. Our analysis results are coarse-grained. Due to the complexity of the Android OS and the limitation of static analysis [26], we filtered out a set of suspicious methods as vulnerability clues rather than directly locating the procedures that can cause risky data inconsistency. The identified vulnerabilities still need manual efforts. On another aspect, only 41 suspicious methods were left from around 330,000 system methods. Combing with the method-related information output by TwinDroid, the manual efforts are acceptable.

Results Accuracy. Since the static analysis does not execute code, our evaluation results cannot guarantee complete accuracy. As mentioned in Section 5.2, due to the diversity of code implementations, TwinDroid cannot determine whether the system performs the duplication checking in the data structures supporting duplicate objects. It may cause false positives, but they can be easily excluded from the final results.

8 Related Work

Here we review the related work on Android configuration and custom permission security.

App Misconfiguration. Aafer et al. [25] detected security configuration changes introduced by Android customization via comparing various ROMs and presented an insecure duplicate components declaration case in a Samsung app (as discussed in Section 7.1). They attributed this case to developers' non-safe practices, but it gives us a clue to consider the manifest's security impact on the Android OS level. Jha et al. [31] reported various types of mistakes committed by developers in writing Android manifest files. They also discussed the duplicate components declaration issue. Han et al. [28] proposed a logic-based approach to discover misconfiguration vulnerabilities in Android manifest files and demonstrate that misconfiguration in app development is common. Scoccia et al. [36] conducted an empirical study of 574 GitHub repositories of open-source Android apps and found that permission-related issues are common in Android manifest files. Most recently, Yang et al. [38] proposed ManiScope to detect manifest misconfiguration by constructing manifest XML Schema. ManiScope identified 33.20% misconfigured Android apps on Google Play and 35.64% misconfigured preinstalled apps from 4,580 Samsung firmware images.

Unlike the above studies, our work focuses on the manifest processing procedures of the Android OS, not developers' misconfigurations. Also, we discovered a design flaw – the Evil Twins flaw, lying in the Android OS and built the corresponding detection tool – TwinDroid, targeting the code of the Android OS.

System Misconfiguration. On the Android OS level, Zhou et al. [40] designed ADDICTED, a tool that automatically detects certain types of flaws in custom driver protection. Aafer et al. [24] investigated hanging attribute references (Hares) caused by Android customization. They implemented Harehunter and found 21,557 possible Hare flaws. Other related analysis works also include bootloader vulnerabilities [29], Android service security [30, 39], residual API security [27], and large-scale firmware measurement [35]. Again, our work does not focus on the misconfiguration issues of vendor customization but on the design flaw of the Android manifest processing procedures.

Custom Permission Security. Our discovered vulnerabilities demonstrated that the manifest file can affect the Android OS security beyond the app, especially the custom permission mechanism security. Tuncay et al. [37] manually discovered two custom permission-related vulnerabilities caused by the inadequate isolation between the system and custom permissions. Their main contribution is proposing a new modular design for the permission model. Li et al. [33] systematically studied the security implications of the custom permission mechanism and developed CuPerFuzzer to detect its design flaws. Unlike these works, our study does not focus on the design of the custom permission mechanism but on the security implications of manifest processing procedures, especially the data structure conversion. Also, considering the huge state space of manifest element combinations, we selected static analysis for code coverage instead of dynamic analysis, like fuzzing. Though our discovered vulnerabilities are permission-related, the Evil Twins flaw can also affect other element configurations (see Section 7.1).

9 Conclusion

In this work, we systematically studied the Android manifest processing procedures and discovered a new category of vulnerabilities called the Evil Twins flaw. Exploiting this flaw, a malicious app can perform harmful actions, even permission escalation. To detect the Evil Twins flaw, we designed an automated static analysis tool – TwinDroid, and discovered four vulnerabilities in the real-world evaluation. Our findings have been confirmed by Google and assigned `CVE-2021-39695`, `CVE-2022-20392`, and `CVE-2023-20971`. We also proposed mitigation measures against our discovered issues.

Acknowledgements

We thank the anonymous reviewers for their insightful comments. We also thank Haojin Zhu for the helpful discussions. This work was partially supported by Taishan Young Scholar Program of Shandong Province, China (Grant No. tsqn202211001). Kehuan Zhang was partially supported by Hong Kong S.A.R. Research Grants Council (RGC) General Research Fund No.14209720.

References

[1] Adding Setter and Getter Methods. https://docs.oracle.com/javaee/6/tutorial/doc/gjbbp.html.

[2] App Manifest Overview. https://developer.android.com/guide/topics/manifest/manifest-intro.

[3] Bug: 209607944. https://android.googlesource.com/platform/frameworks/base/+/b5efdf729385cc54f225496d3ba20f1cb5b68250.

[4] Bug: 213323615. https://android.googlesource.com/platform/frameworks/base/+/548edbb850227e076735615f83f8e23352b0b82d.

[5] Define a custom app permission. https://developer.android.com/guide/topics/permissions/defining.

[6] Define and enforce permissions: Naming convention. https://developer.android.com/guide/topics/permissions/defining#naming.

[7] Honor Magic UI Security Update, March 2022. https://www.hihonor.com/global/support/bulletin/2022/3/.

[8] Honor Magic UI Security Update, September 2022. https://www.hihonor.com/global/support/bulletin/2022/9/.

[9] HUAWEI EMUI/Magic UI security updates April 2022. https://consumer.huawei.com/sa-en/support/bulletin/2022/4/.

[10] HUAWEI EMUI/Magic UI security updates January 2023. https://consumer.huawei.com/en/support/bulletin/2023/1/.

[11] Intents and Intent Filters. https://developer.android.com/guide/components/intents-filters.

[12] Introduction to activities. https://developer.android.com/guide/components/activities/intro-activities.

[13] openXmlResourceParser. https://developer.android.com/reference/android/content/res/AssetManager#openXmlResourceParser(int,%20java.lang.String).

[14] <permission-tree>. https://developer.android.com/guide/topics/manifest/permission-tree-element.

[15] Permissions on Android. https://developer.android.com/guide/topics/permissions/overview.

[16] R.styleable. https://developer.android.com/reference/android/R.styleable.

[17] Runtime Permissions: Defining custom permission. https://source.android.com/docs/core/permissions/runtime_perms#defining-custom-perms.

[18] "sargo" for Pixel 3a. https://developers.google.com/android/images#sargo.

[19] Security Updates. https://security.samsungmobile.com/securityUpdate.smsb.

[20] Security Updates (in Chinese). https://r1.realme.net/general/20221214/1671005743438.pdf.

[21] Security Updates (in Korean). https://www.lge.co.kr/support/mobile-sw-update-SW20220928222002?keyword=&page=1&orderType=%EC%B5%9C%EC%8B%A0%EC%88%9C&cate=&category=.

[22] Soot. https://soot-oss.github.io/soot/.

[23] The Android Permission Model: Accessing Protected APIs. https://source.android.com/docs/security/overview/app-security#the-android-permission-model-accessing-protected-apis.

[24] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiao-yong Zhou, Wenliang Du, and Michael Grace. Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS), Denver, CO, USA, October 12-16, 2015*, 2015.

[25] Yousra Aafer, Xiao Zhang, and Wenliang Du. Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis. In *Proceedings of the 25th USENIX Security Symposium (USENIX-Sec), Austin, TX, USA, August 10-12, 2016*, 2016.

[26] Brian Chess and Gary McGraw. Static Analysis for Security. *IEEE Security & Privacy*, 2(6):76–79, 2004.

[27] Zeinab El-Rewini and Yousra Aafer. Dissecting Residual APIs in Custom Android ROMs. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS), Virtual Event, Republic of Korea, November 15 - 19, 2021*, 2021.

[28] Zhihui Han, Liang Cheng, Yang Zhang, Shuke Zeng, Yi Deng, and Xiaoshan Sun. Systematic Analysis and Detection of Misconfiguration Vulnerabilities in Android Smartphones. In *Proceedings of the 13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Beijing, China, September 24-26, 2014*, 2014.

[29] Roee Hay. fastboot oem vuln: Android Bootloader Vulnerabilities in Vendor Customizations. In William Enck and Collin Mulliner, editors, *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT), Vancouver, BC, Canada, August 14-15, 2017*, 2017.

[30] Antonio Ken Iannillo, Roberto Natella, Domenico Cotroneo, and Cristina Nita-Rotaru. Chizpurfle: A Gray-Box Android Fuzzer for Vendor Service Customizations. In *Proceedings of the 28th IEEE International Symposium on Software Reliability Engineering (ISSRE), Toulouse, France, October 23-26, 2017*, 2017.

[31] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. Developer Mistakes in Writing Android Manifests: An Empirical Study of Configuration Errors. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR), Buenos Aires, Argentina, May 20-28, 2017*, 2017.

[32] Li Li, Tegawendé F. Bissyandé, and Jacques Klein. Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark. *IEEE Transactions on Software Engineering*, 47(4):676–693, 2021.

[33] Rui Li, Wenrui Diao, Zhou Li, Jianqi Du, and Shanqing Guo. Android Custom Permissions Demystified: From Privilege Escalation to Design Shortcomings. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland), San Francisco, CA, USA, 24-27 May 2021*, 2021.

[34] Radek Pelánek. Fighting state space explosion: Review and evaluation. In *Proceedings of the 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS), Italy, September 15-16, 2008*, 2008.

[35] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland), San Francisco, CA, USA, 24-27 May 2021*, 2021.

[36] Gian Luca Scoccia, Anthony Peruma, Virginia Pujols, Ivano Malavolta, and Daniel E. Krutz. Permission Issues in Open-Source Android Apps: An Exploratory Study. In *Proceedings of the 19th International Working Conference on Source Code Analysis and Manipulation (SCAM), Cleveland, OH, USA, September 30 - October 1, 2019*, 2019.

[37] Güliz Seray Tuncay, Soteris Demetriou, Karan Ganju, and Carl A. Gunter. Resolving the Predicament of Android Custom Permissions. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 18-21, 2018*, 2018.

[38] Yuqing Yang, Mohamed Elsabagh, Chaoshun Zuo, Ryan Johnson, Angelos Stavrou, and Zhiqiang Lin. Detecting and Measuring Misconfigured Manifest in Android Apps. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS), Los Angeles, CA, USA, November 7-11, 2022*, 2022.

[39] Lei Zhang, Zhemin Yang, Yuyu He, Zhenyu Zhang, Zhiyun Qian, Geng Hong, Yuan Zhang, and Min Yang. Invetter: Locating Insecure Input Validations in Android Services. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS), Toronto, ON, Canada, October 15-19, 2018*, 2018.

[40] Xiao-yong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland), Berkeley, CA, USA, May 18-21, 2014*, 2014.

A Appendix

A.1 Changeable Permission

During runtime permission granting, the system will check whether each permission in a granted group is *changeable*. Changeable permission requires its corresponding `bp.protectionLevel` is `signature|development` or `dangerous`. The system will throw an exception if the granted group contains unchangeable permissions. Hence, we set the protection level of the second declared `com.example.cp` to `signature|development` to avoid app crashing.

A.2 Adjustment Rules against Variable Sets

The adjustment against variable sets contains three operations: *add*, *remove*, and *update*. Table 3 illustrates the specific adjustment rules.

- Add variable: add an outflow variable to the corresponding variable set.
- Remove variable: remove a variable from the corresponding variable set.
- Update variable: remove an inflow variable from the corresponding variable set and add the outflow variable.

A.3 Optimization Strategies

Path Explosion Avoiding. During generating execution paths from the base-method's control flow graph (CFG) (see Section 4), the CFG containing lots of branchs may generate massive execution paths. To avoid path explosion [34], TwinDroid uses the following optimized steps to generate paths for such CFG. Note that an execution path is composed of various logically connected code blocks, and every block is assigned an ID in the depth order.

(1) *Collect noteworthy blocks.* TwinDroid first collects all *noteworthy blocks* of the input base-method. Such a block contains an *noteworthy statement* that is relevant to source variables or an element type. There are three kinds of noteworthy statements: assignment, invocation, and conditional, as illustrated in Table 4.

(2) *Find the least common ancestor (LCA).* Next, TwinDroid finds out the LCA of collected noteworthy blocks from the CFG. Specifically, it obtains each noteworthy block's ancestor set (including the noteworthy block itself) and calculates the intersection of all ancestor sets. The block in the intersection with the largest ID is the LCA.

(3) *Generate execution paths.* TwinDroid performs the depth-first search against CFG from LCA to generate execution paths. Note that, generating execution paths from LCA can guarantee that the obtained execution paths cover all propagation paths of source variables in the base-unit. At the same time, it excludes unnecessary paths.

Incorrect Path Pruning. The execution paths obtained by TwinDroid (based on Soot) may be incorrect, which cannot be executed in practice. For example, for a `switch` statement with n branches, Soot may convert it into two relevant `switch` statements described by Jimple with n branches each. Therefore, after transforming the Android source code from compiled bytecode to Jimple, the scale of generated execution paths will be increased from n to n^2. However, in these n^2 paths, $n^2 - n$ paths are incorrect. Therefore, during tracing source variables along with an execution path, when facing a conditional statement (i.e., `if` or `switch`), TwinDroid judges whether the next statement is the matched target statement based on the result of the condition checking and terminate tracing if it does not match.

Abstract Method Implementation. During the new base-unit construction, the invoked method may be abstract without a method body. For this situation, TwinDroid seeks its corresponding method implementation. Specifically, TwinDroid first obtains the declaring class of this abstract method. (1) If the declaring class is an abstract class, TwinDroid iterates its subclasses to locate the concrete method of which declaration is the same as this abstract method. (2) If the declaring class is an interface, TwinDroid locates the corresponding method implementation in its class implementation. If there are multiple implemented classes, TwinDroid determines the specific one based on the definition of the instance invoking this abstract method.

Unnecessary Tracing Elimination. To avoid redundant tracing of the same base-unit (having the same base-method, source variables, and associated element types), TwinDroid keeps the information of each analyzed base-unit, including

Table 3: Adjustment rules against the variable set (the initial variable set is [a]).

Operation	Situation	Code Example	Set Adjustment	Adjustment Object
Add the outflow variable.	Get a copy of the inflow variable.	`b = a;`	[a] → [a,b]	source variable set passing variable set
	Get part of the inflow variable.	`b = a.getName();`	[a] → [a,b]	passing variable set
	Get a judgment result about the inflow variable.	`z = a.isEmpty();`	[a] → [a,z]	passing variable set
Remove a variable.	Redefine a variable. This variable is no longer relevant to the data we want to trace.	`a = null;`	[a] → []	source variable set passing variable set
	Update the recorded element type. The currently traced variable is no longer associated with the new element type.	–	[a] → []	passing variable set
Update the inflow variable to the outflow variable.	Others. The propagation from the inflow variable to the outflow variable represents a complete flow of the traced data.	`s = a.toString();`	[a] → [s]	passing variable set

Table 4: Noteworthy statements.

Statement	Required Conditions
Assignment	The definition of an entry source variable.
	Source variables are at the right of the equal sign.
Invocation	The callee uses source variables as parameters.
	The instance of the callee is the source variable.
Conditional	Comparison with an element type.

the newly generated base-units from it and the tracing results of its source variables. If an unanalyzed base-unit is the same as an analyzed base-unit, TwinDroid can get the data flow analysis results of this unanalyzed base-unit directly based on the kept information corresponding to this analyzed base-unit. Besides, TwinDroid skips unimportant statements during tracing, such as those related to information logging.

A.4 Component Protection Bypassing

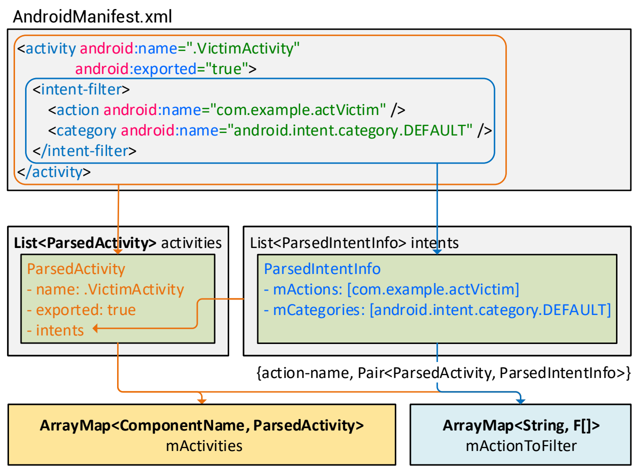
Background. Activity is a crucial component of an app, and it is the entry point for interacting with the user [12]. To use activities, an app must declare them in its manifest file through `<activity>` elements (as shown in Listing 6). The app can restrict the activity's exposure to other apps through `android:exported` and `android:permission` attributes (Lines 3, 12, and 13). An activity can be activated by an asynchronous message called Intent [11]. There are two types of Intent: explicit and implicit. The explicit Intent specifies a specific activity to respond to this Intent. In comparison, the implicit Intent declares a general action to perform. The Android OS will then check which registered activities can handle that action. Defining an Intent filter with an `action` for an activity (Lines 4-7 and 14-17) specifies the type of implicit Intent to which this activity can respond.

Evil Twins Flaw-related Bug. In Table 2, Items 6 & 7 show the `addActivitiesLocked` and `queryIntentActivities-InternalBody` methods access the package setting – [`List<ParsedActivity> activities`]. The former method registers an app's declared activities in the system. The latter is for querying matched target activities while launching the activity through an implicit Intent. When launching an activity, the system will enforce corresponding restrictions based on its configurations. For example, if the activity is protected by the `CALL_PHONE` permission (i.e., Line 13 in Listing 6), access to it will fail without obtaining the `CALL_PHONE` permission. However, we find that:

Bug: *During processing the twin `<activity>` elements, due to the data structure conversion with component protection-related attribute loss, the {`app component` → `protection status`} mapping relationship becomes inconsistent, further resulting in the app component protection failure.*

Exploit. Suppose that there is an app `app-victim` (package name: `com.example.victim`) on the user's phone. This app declares two activities with the same name (`VictimActivity`) and the same Intent filter (action: `com.example.actVictim`), as shown in Listing 6. The first activity is completely exposed, and the second is protected by the `CALL_PHONE` permission.

```
1 <activity
2   android:name=".VictimActivity"
3   android:exported="true">
4   <intent-filter>
5     <action android:name="com.example.
          actVictim" />
6     <category android:name="android.intent
          .category.DEFAULT" />
7   </intent-filter>
```


AndroidManifest.xml
List<ParsedActivity> activities
List<ParsedIntentInfo> intents
ParsedActivity
- name: .VictimActivity
- exported: true
- intents
ParsedIntentInfo
- mActions: [com.example.actVictim]
- mCategories: [android.intent.category.DEFAULT]
{action-name, Pair<ParsedActivity, ParsedIntentInfo>}
ArrayMap<ComponentName, ParsedActivity>
mActivities
ArrayMap<String, F[]>
mActionToFilter

```
<activity android:name=".VictimActivity"
        android:exported="true">
  <intent-filter>
    <action android:name="com.example.actVictim" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</activity>
```

Figure 6: Data structure conversion of `<activity>` elements.

```
</activity>

<activity
  android:name=".VictimActivity"
  android:exported="true"
  android:permission="android.permission.
      CALL_PHONE">
  <intent-filter>
    <action android:name="com.example.
        actVictim" />
    <category android:name="android.intent
        .category.DEFAULT" />
  </intent-filter>
</activity>
```

Listing 6: Manifest file (part) of `app-victim`.

The adversary prepares a malicious app – `app-bug`, which tries to access `VictimActivity` using Intents, as shown in Listing 7.

```
// through explicit Intent
Intent intent = new Intent();
ComponentName cp = new ComponentName("com.
    example.victim", "com.example.victim.
    VictimActivity");
intent.setComponent(cp);
startActivity(intent);

// through implicit Intent
Intent intent = new Intent();
intent.setAction("com.example.actVictim");
startActivity(intent);
```

Listing 7: Launch `VictimActivity` through Intents.

▶ The victim user installs `app-bug`. When this app tries to launch `VictimActivity` of `app-victim` using the explicit Intent, the access is forbidden. The system throws a security exception indicating permission denial. However, ***through the implicit Intent,*** `app-bug` ***launches the*** `VictimActivity` ***successfully***.

Cause Analysis. As illustrated in Figure 6, during the installation of an app, for each declared activity in its manifest, ParsingPackageUtils (PPU) creates its corresponding `ParsedActivity` instance to store this activity's configuration data. During this procedure, for each declared Intent filter in this activity, PPU creates its corresponding `ParsedIntentInfo` instance and adds this instance to the [`List<ParsedIntentInfo> intents`] field of this `ParsedActivity` instance.

After that, ComponentResolver (CR) registers each declared activity into the system. For each `ParsedActivity` member of `activities`, CR adds it to [`ArrayMap<ComponentName, ParsedActivity> mActivities`], which stores all registered activities' information. The `ComponentName` object keeps this activity's fully qualified class name (composed of the source package name and the activity name). Besides, for each `ParsedIntentInfo` instance in this `ParsedActivity` instance's `intents`, CR creates its corresponding {`action name` → `Pair <ParsedActivity, ParsedIntentInfo>`} mapping and adds this item to `mActionToFilter`, an `ArrayMap<String, F[]>` storing the information of all registered activities' declaring Intent filters.

For the twin activities declared in `app-victim`, since their fully qualified class names are the same, `mActivities` retains the configuration data of the latter one. While since both activities define the Intent filters, `mActionToFilter` retains both the configuration data.

When launching the activity using explicit Intent, the system queries the target activity in `mActivities`. Therefore, *when* `app-bug` *tries to launch* `VictimActivity` *explicitly, the matched activity is the latter* `VictimActivity`. Since this `VictimActivity` is protected by the `CALL_PHONE` permission, `app-bug` cannot access it without `CALL_PHONE`.

When launching the activity using implicit Intent, the system queries the target activity in `mActionToFilter` according to the `action name` specified in this Intent. If there exist multiple matched activities with the same fully qualified class name, the system will select the first one. Therefore, *when* `app-bug` *tries to launch* `VictimActivity` *implicitly, the matched activity is the former* `VictimActivity`, and the launching is successful.

Besides, according to the above analysis, if the permission protection is put on the first `VictimActivity`, it also can be bypassed through explicit Intent.

Discussion. The other two component types – receivers and services – also support being activated by explicit and implicit Intents [11]. After analyzing, we find that this bug also exists in accessing them (based on Items 8 & 9 in Table 2).