POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Detecting and exploiting misexposed components of Android applications

Relatori
prof. Antonio Lioy
prof. Ugo Buy

Francesco Pinci

December 2018

To my parents, my sister, and my relatives, who have been my supporters throughout my entire journey, always believing in me, and providing me with continous encouragement.
This accomplishment would not have been possible without them.
Thank you.

Summary

Smartphones and tablets have become an essential element in our everyday lives. Everyone use these devices to send messages, make phone calls, make payments, manage appointments and surf the web. All these use cases imply that they have access to and collect user sensitive information at every moment. This has attracted the attention of attackers, who started targetting them. The attraction is demonstrated by the continuous increase in the sophistication and number of malware that has mobile devices as the target [1] [2].

The Android project is an open-source software which can be downloaded and studied by anyone. Its openness has allowed, during the years, an intensive inspection and testing by developers and researches. This led Google to constantly updating its product with new functionalities as well as with bug fixes. Various types of attacks have targetted the Android software but all of them have been mitigated with the introduction of new security mechanisms and extra prevention methods. Starting from September 2018, 16 major versions of the OS have been realized, reducing incredibly the attack surface exposed by the system.

The application ecosystem developed by the Android project is a key factor for the incredible popularity of the mobile devices manufactured and sold with the OS. The users can benefit from an immense official store as well as alternative stores, providing applications for every category and need. The essentiality of applications has increased the importance of their security in the OS platform. The development of strong security mechanisms is of primary importance, but it is not enough. Software is written by humans, which are not perfect and can make mistakes. This requires the creation of tools, essential for the analysis and testing of the security implemented in a system.

The Android architecture and applications structure require interaction between the various software running on a device. This is made available by applications components, modular objects which implements the different features provided by the app. This opening could create holes in the Android Security mechanisms. In particular, our research starts with the assumption that application's components can generate vulnerabilities when not developed correctly and with attention to their security.

The study of components interactions and system applications lead us to the discovery of possible interaction vulnerabilities, confirmed by the first major issue found in the PhoneApp system application. Due to the large size of the source code in applications, the need for a tool to automatize the process arose. At this point, we projected and developed the tool architecture, including a static analysis component and a dynamic analysis one for testing. The results obtained from the tool demonstrates that our assumptions were correct, leading us to discover the second type of vulnerabilities. Both the type of vulnerabilities have been exploited to present examples of possible malicious applications that could be developed by attackers.

Finally, the tool has been perfectioned and used to understand how the presented issues are widespread in the applications provided by the Android operating system or by third-party developers. The results are used to understand in which situations the components become more common and to define possible approaches to mitigate the problem.

Acknowledgements

I would first like to thank my advisor Prof. Ugo Buy and Prof. Rigel Gjomemo, for being always available and patient whenever i had questions or doubts and to help me solving the problems encountered in this journey.
I would also like to thank my advisor Prof. Antonio Lioy for hist attention, time and interest.

Contents

Summary IV

List of Figures X

List of Tables XI

1 Introduction 1

1.1 Context 1

1.2 Deliverables 1

1.3 Environment 2

1.4 Outline 3

2 Background 4

2.1 Android Platform architecture 4

2.1.1 Linux Kernel 4

2.1.2 Android Runtime 6

2.1.3 Java API framework 6

2.2 Android applications 7

2.2.1 Activities 8

2.2.2 Services 9

2.3 Intents 10

3 Android Security 12

3.1 Android security mechanisms 12

3.1.1 Application Sandbox 13

3.1.2 Permissions 13

3.2 Exposed application components 14

3.2.1 System and third-party applications 16

4 Vulnerabilities and Exploits 18
4.1 Unexpected intents . . . 18
4.1.1 Denial-of-Service . . . 18
4.1.2 Real-life attack: PhoneApp application vulnerability . . . 20
4.2 Permission re-delegation . . . 21
4.2.1 Real-life attack: Camera2 application vulnerability . . . 21

5 Vulnerabilities Detection 24
5.1 Detection of exposed components . . . 24
5.1.1 Algorithm . . . 24
5.1.2 Pseudo-code . . . 25
5.2 Identification of vulnerable components . . . 26
5.2.1 Android Debug Bridge . . . 26
5.2.2 Testing with adb . . . 27

6 Analysis Tool 29
6.1 Tool structure . . . 29
6.1.1 Apk Extractor . . . 30
6.1.2 Reverse engineering module . . . 31
6.1.3 Manifest analyzer . . . 31
6.1.4 Dynamic tester . . . 32
6.2 Analysis reports . . . 32
6.3 User Manual . . . 33

7 Results 35
7.1 Analysis setup . . . 35
7.1.1 Android operating system . . . 35
7.1.2 Third-party applications . . . 36
7.2 Analysis results . . . 37
7.2.1 Android OS applications results . . . 37
7.2.2 Third-party applications results . . . 37

8 Conclusion 40
8.1 Proposed solutions . . . 40
8.2 Fixes and achievements . . . 42
8.3 Future works . . . 42
8.3.1 Complete analysis of Android devices . . . 42
8.3.2 Auotomatic detection of permission re-delegation . . . 43
8.3.3 Creation of a complete tool . . . 43

Bibliography 44

List of Figures

1.1 Android platform versions distribution [3] 2

2.1 Android Platform architecture stack [6] 5
2.2 Android activity lifecycle [11] 9
2.3 Activity started with an intent [13] 11

4.1 System dialog dislpayed when an application crashes [17] 19
4.2 Permission re-delegation scheme [18] 22

6.1 Auotomated analysis and detection tool architecture 30

8.1 Android Studio dialog for service component creation 41

List of Tables

3.1 Declaration conditions for exposed components 16

7.1 List of devices used for testing 36
7.2 Results for components in Android 6.0.0 Marshmallow 37
7.3 Results for components in Android 7.0.0 Nougat 38
7.4 Results for components in Android 7.1.1 Nougat 38
7.5 Results for components in Android 8.0.0 Oreo 39
7.6 Results for components in Android 8.1.0 Oreo 39
7.7 Results for components in Android 9.0.0 Pie 39
7.8 Results for third-party applications from the Google Play store 39

Chapter 1

Introduction

1.1 Context

Smartphones became essential in our lives. Android is the world's most popular mobile OS with more than 2 billion devices using it every month. Due to the infinite functionalities that they can implement, a huge amount of user personal and private information are collected and stored. At the same time, thanks to sensors and hardware components, smartphones are capable of recording and collecting data of user everyday life. With such capabilities and distribution numbers, the Android OS security must be a priority.

Most of the functionalities of a device are managed by applications. They are developed and executed in the environment of the Android framework, which is a set of standards, protocols, and functionalities that developers can use to build apps.

This work aims at analyzing the security of Android, understanding in which cases its implementation can lead to unwanted errors generating security vulnerabilities and privacy issues. When these issues are found, we study how they could be exploited by malicious actors to craft different types of real-life attacks. Additionally, we evaluate how widespread the attacks could be according to the affected OS version distribution or third-party application popularity. Finally, the results are analyzed to propose specific solutions for each of the issues found.

1.2 Deliverables

The research work had to analyze the possible problems arising from a set of Android models widely used in the entire platform. The project source code and the number of third-party applications on the market are incredibly large, hence a manual analysis is not feasible. We developed a tool to automize the process of scanning the source code of Android applications to detect exposed components and test them to identify possible vulnerabilities. The analysis results demonstrated that the platform models are not ideal to help developers in writing secure applications. The problem does not only affect third-party applications but also applications released with the OS. We discovered two possible attack methods that malicious actors could use or even worst, already be using: a Denial of Service attack and a Permission re-delegation attack. Moreover, the numeric results show that the problem, in the newer version of Android and recent developed third-party applications, is not decreasing. Finally, we consider a set of possible actions that can be taken to reduce or eliminate the problems detected.

Our analysis is made on the set of APIs that, as shown in 1.1, represent the main distributions present on the market. The set of third-party applications analyzed have been downloaded directly from the official Play store which is the only official market guaranteed by Google. The tool has been developed in Bash and Python programming languages. The former was used because the Android platform contains a command line debug interface called Android Debug Bridge, optimal to programmatically interact with a device and automate the testing process. Python was chosen because of its efficiency in scanning the contents of a text file such as the AndroidManifest files.

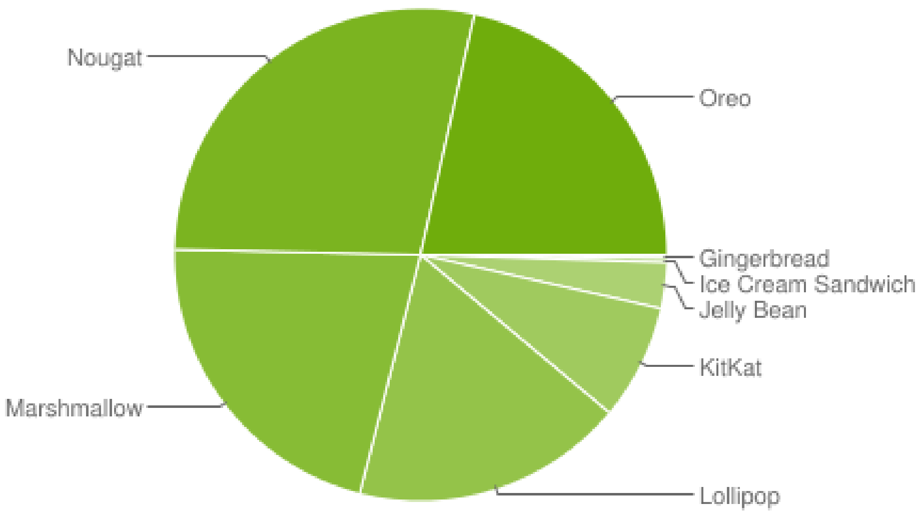


Figure 1.1: Android platform versions distribution [3]

1.3 Environment

Android is an open source software system and as such, most of the research was made directly accessing the source code. It is distributed through the Android Open Source Project located in a Git repository hosted by Google. The source code of each code-line can be downloaded and accessed, up to the latest release.

The work focuses on the most widespread OS versions: *Android Marshmallow*, *Android Nougat*, *Android Oreo* and the latest *Android Pie*. The development tool used is *Android Studio*, which is the official Integrated Development Environment for Android apps. Tests were run on the following devices: Google Pixel running Android 9.0 Pie with API 28, Google Pixel 2 running Android 8.1 Oreo with API 27 as real devices, and Google Pixel 2 running all previous listed versions with APIs 23, 24, 25, 26, 27, 28 as emulated devices.

1.4 Outline

For a better understanding of the concepts and elements involved in this research, the document starts with a background chapter. It includes the theoretical information necessary to have a clear idea of the Android framework and its mechanisms.

The security aspects are then explained in Chapter 3, with a focus on the target of this research: application components. The possible risks raised by vulnerabilities in both System and Third-party applications are evaluated.

In Chapter 4, the vulnerabilities found are presented in detail along with proofs of concept to give an example of real-life attacks.

In the following two chapters, the methods used to detect the issues are explained and it is explained how they were integrated into the developed analysis tool. A solution approach is then proposed to help in reducing the attack surface and avoid the involuntary introduction of new vulnerabilities by developers.

Chapter 7 presents the final results obtained from the analysis of all the Android major release versions and a large set of third-party applications.

Chapter 2

Background

Android is a Google-developed mobile operating system for smartphones, tablets and touchscreen devices. By 2018, were sold over 383 million smartphones with 85.9 percent powered by Android [4]. It is an open source software distributed by the AOSP project through a Git repository. The OS has over two billion active users per month with an official application store featuring over 3.3 million applications.

AOSP kernel is derived from the Linux kernel's LTS branches. The use of a Linux kernel allows the project to rely on its security features which are deeply used and tested. On top of the Linux kernel, software including libraries and APIs is written in C, while application software runs on an application framework Java-based. Google has also recently included full support for the development of applications in the Kotlin programming language [5].

2.1 Android Platform architecture

Android Platform architecture is a software stack where each component depends from and communicate with the underlying layer. 2.1 shows the complete set of layers and components.

2.1.1 Linux Kernel

The kernel is the core of an operating system, a computer program loaded on start-up, in charge of managing and controlling every element of the system. The Android platform is founded on the *Linux kernel*, which is the most common choice for mobile devices. The role of the kernel includes managing the interaction between the hardware and the software. The popularity of Linux allows a reliable development and maintenance of hardware drivers by device manufacturers. Moreover, the advantage of using this kernel is that it contains several security features on which the OS can rely. It has been used for years in security-sensitive environments, receiving constantly research, attacks, and consequent fixes by thousands of developers. Currently is considered by many security professionals and institutions one of the most trusted secure kernel. Some of the main key security features that Android take advantage of are:

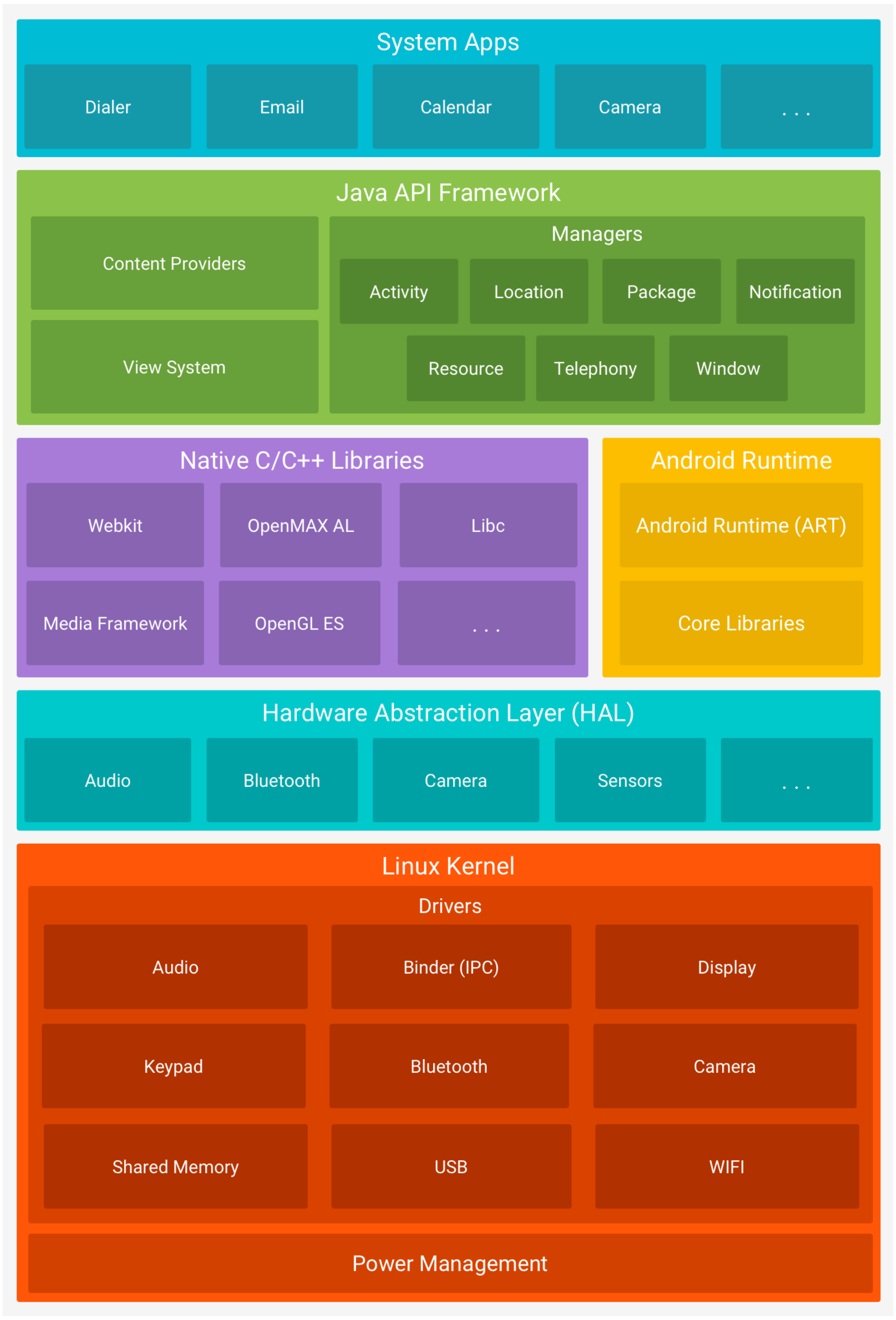


Figure 2.1: Android Platform architecture stack [6]

- Process sandbox
- Permission model
- A secure mechanism for IPC

- Modularity, with the possibility to remove components if they are potentially insecure [7]

2.1.2 Android Runtime

A runtime system is a set of software instructions implementing portions of an execution model. In particular, it executes actions which are not part of the running program but that are needed for its correct execution and interaction within the runtime environment. Every programming language defines a runtime system. The *Android runtime* (ART) translates the application's bytecode into instructions based on native code that can be executed by the device system. The specific instruction formats executed in the Android context are the Dalvik Executable format and the Dex bytecode. Developers write programs in the Java language and the bytecode is generated by build toolchains at compile time. From Android version 5.0 on, an instance of the ART is created in each app process. Prior to it, the runtime was the Dalvik but has been updated to introduce more efficient components such as Ahead-of-time compilation. A set of core runtime libraries are also included in Android. They provide the majority of Java programming language functionalities which are also used by the Java API framework.

2.1.3 Java API framework

Android exposes a set of features and functionalities accessible through Java-based Application Programming Interfaces. The APIs have the role of simplifying the use of modular system components implemented by the OS which are the essential building blocks for apps creation. Each API is developed and distributed in a package. Any new Android code-line released is characterized by a new API version containing additional functionalities. The latest Android 9 Pie version features the new API level 28. Updates to the framework API are created in order to be compatible with the earlier versions. While new functionalities or replacement are introduced, the previous parts are deprecated but kept in the framework so that applications developed on old APIs will keep working. In just a few cases components of the interfaces may be removed or modified for major reasons such as applications or system security. APIs new versions are distinguished with API Levels, increasing integer numbers starting from 1. The Level has an essential role in the apps development and users experience:

- It lets the platform indicate the maximum framework level supported
- It lets an application indicate the framework level required
- It lets the OS understand if an application is compatible with the Android device version [8]

The Java API framework access the hardware capabilities of the device through interfaces provided by the hardware abstraction layer. The HAL is a collection of library modules loaded whenever the OS receives a request or need to access an hardware component.

Android creates an entire set of software features that are made available through the API. Such features allow communicating with the system and help the interaction between the user and the apps. The following are a list of core components and services available:

- A system to build and manage an app's UI called View System
- A Notification Manager that allows apps to display notifications and custom alerts in the interface status bar
- A set of data managers named Content Providers. They help applications to share data and they manage the access to data of the system or of other apps
- An Activity Manager which control the status of applications and regulate their lifecycle. It also provides a navigation stack to remember the history of user interactions with each app.

2.2 Android applications

An Android app is a software application written using Java, Kotlin and C++ languages, running on the Android platform. Apps can be developed and tested with the official IDE Android Studio. The SDK tools compile source code, resource files, and any data into an Android package called APK with .apk extension, which is an archive file. It contains all the information necessary for an app to be installed on a device and to be executed by the operating system.

There are two types of applications: *system apps* and *third-party* apps. System applications are installed on the device with the OS itself but they can be part of the AOSP or related to the manufacturer that customized the ROM for the device it is selling. They can't be uninstalled by the user and have special permissions that grant risky privileges, giving them special controls over the system. The system apps purpose is to directly interact with the user, but also to provide functionalities that third-party apps can utilize. Examples of this apps are the *PhoneApp* (managing the network connection), or the *Camera2* app. Third-party applications can be developed by anyone and are installed by the user, who can get them from over two and a half million offered through the Google Play store. Besides, a relevant number of alternative sources are available with fewer guarantees on apps legitimacy.

The essential building blocks of Android apps are called App components. A component is a module through which the user, the system or another application can access your app. Usually, more than one component is declared in an app and it is possible that some depend on others [9]. Only four types of component exist in the Android framework:

- Activities

- Services
- Broadcast receivers
- Content providers

Components can be declared in a specific Android file called *AndroidManifest.xml* which is an XML file describing essential information about the application and its elements.

2.2.1 Activities

A mobile app is not always started from the same point. A user could open the email app from its icon and see the list of received emails, or open it from an external application to directly start composing an email. This paradigm is made easy thanks to Android activities. An application can declare more activities and each of them has a different interface representing different elements. An activity does not represent the entire application but just a module of it. Each one declares its own window where the developer can draw the optimal UI. Whenever a new activity is started, its window is displayed replacing the previous one and the user can start interacting with it. Usually, activities can depend on the application's data but they don't depend on other activities. They can start components of other apps and in the same way, they can be started by other components. An activity is declared in the manifest file using the <*activity*> tag as a child of the <*application*> element. For instance, to declare an activity defined by the *ExampleActivity* class, the XML code would be the following [10]:

```
<manifest ... >
  <application ... >
    <activity android:name=".ExampleActivity" />
    ...
  </application ... >
  ...
</manifest >
```

Activity declaration example

The system manages activities with an activity stack. Every time a new activity is started, it is placed on the top of the stack until the activity exits or another activity is started. The lifecycle of activities is defined with 7 states displayed in 2.2. The lifecycle is defined by specific methods that are called by the OS when necessary. The two main methods are onCreate and onDestroy. The former is called when the activity has to be created and is in charge of setting up all the necessary elements and objects, the latter is a final call necessary to release all resources before the activity gets destroyed.

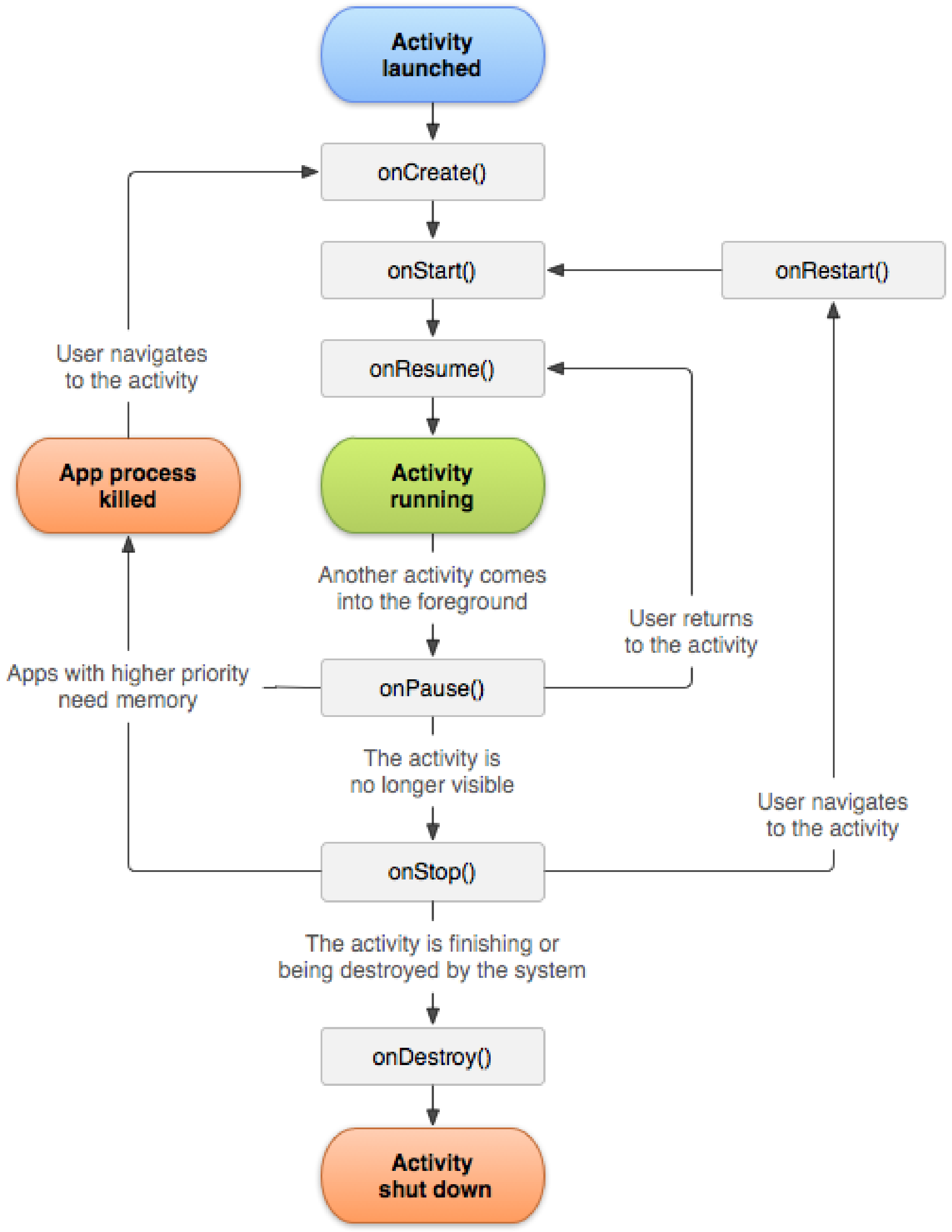

Activity launched
onCreate()
onStart()
onRestart()
onResume()
User navigates to the activity
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 2.2: Android activity lifecycle [11]

2.2.2 Services

A service, differently from an activity, is a general-purpose component without an interface. It is used for long-running operations that otherwise would interrupt the UI for too long to keep the device usable. For example, a service can be used to play music in the background while the user is interacting with another application or an activity of the same app. Services can be started and interact with other components as well as interact with components of another application. Two type of services can be instantiated: Started services and Bound services. Started services perform specific long-running operations and they remain active until their work is completed. Bound services are instead related to the applications that requested their service. This services usually provide an API for other processes. Different apps can bound to them simultaneously, and they remain active until all the apps unbound. Services lifecycle has a few additional methods compared to activities. They are started when a component calls startService or bindService and the OS then calls onStartCommand with the arguments passed by the client. When a client calls bindService, the service returns a special object called IBinder which can be used to call the methods of the service API. A service is declared in the manifest file using the <*service*> tag as a child of the <*application*> element. For instance, to declare a service defined by the ExampleService class, the XML code would be the following [12]:

```
<manifest ... >
   <application ... >
      <service android:name=".ExampleService" />
      ...
   </application ... >
   ...
</manifest >
```

Service declaration example

2.3 Intents

App components are supposed to communicate and interact. In most of the cases, the communication happens between components of different apps in different processes. Linux processes do not share memory space, hence they need a common protocol to communicate. The Android environment implements inter-process communication through special objects: intent messages. Intents can be used to interact with a component and ask for an action. In particular, they are used in two cases:

- **Starting an activity**
  An intent containing a description of the activity to start and any additional necessary data has to be created. Such intent is passed to the startActivity method and the system process is in charge of transferring the message to the destination activity, following the flow displayed in 2.3. If the component sending the intent wants to receive a result from the destination activity, the method startActivityForResult has to be called instead. Once the destination activity has performed the requested action, a new intent is sent back to the original activity containing the result.

- **Starting a service**
  Again, an intent containing the information about the destination service has to be created. In this case, the method to be called is startService to request a one-time action to be executed. Differently, if more than one action is needed,

the bindService method has to be called passing the intent. If the destination service is available, a Binder object is returned and the functions available in the service interface can be called.

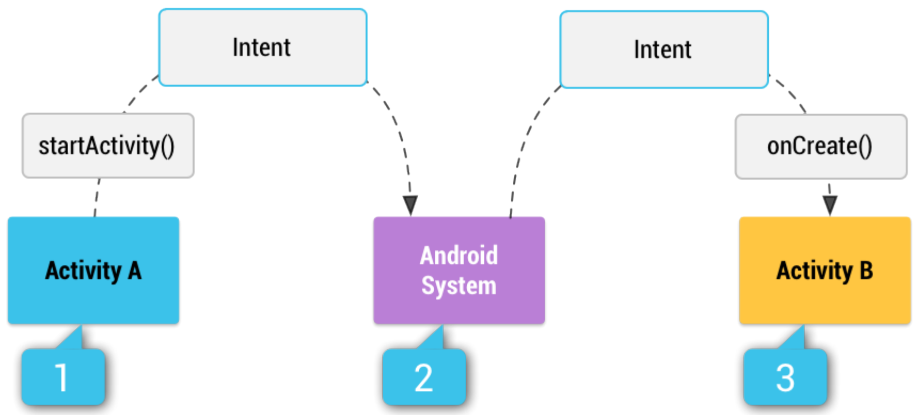


Figure 2.3: Activity started with an intent [13]

There are two different types of intent messages: implicit intents and explicit intents.

- **Implicit intents** can be created without declaring a specific component as a destination but a general name for the action needed is sufficient. If more then one app is capable of managing the requested action, the system lets the user choose which of the available app should handle it and forward the intent to the selected component.
- **Explicit intents** must contain specific information about the destination component: the concatenation of the package name and the class name or just the component package name [14].

Both types of intent can contain various type of standard information and the related data. Android defines a set of fields and the corresponding methods to populate such fields. The primary information in an intent are action, data and component name. An action is a constant string such as *ACTION_MAIN* or *ACTION_DIAL*, while data can be of various types depending on what kind of information the destination component has to operate on, such as an integer value or a contact number. The component name is what makes an implicit intent to become an explicit one.

An app has to inform the OS which actions can handle in order to receive an intent. This is done in the AndroidManifest file declaring intent filters. They are an Android feature that let the system know if the component wants to receive intents requesting a certain action. To use this feature, apps has to declare a <*intent-filter*> tag attribute in the component tag element. It must include an <*action*> field and possibly a <*category*> or a <*data*> property.

Chapter 3

Android Security

Android is a system software developed for portable devices. It is mainly used on smartphones, mobile computing devices that allow people to store, access and share any kind of data, in any place, at any moment during their daily routine. Users can download new software in the form of apps and it has to run alongside every other app or OS program. This variety of elements imply the need for different software protections against possible attacks introduced by each of them. During the years, Android has introduced several protection mechanisms and keeps adding new ones to defend devices against modern attacks and newly discovered vulnerabilities.

3.1 Android security mechanisms

Since every element of the Android software stack relies on the kernel, that's where the most important security mechanisms are. The Linux kernel is the base for a mobile computing environment and provides several key security features.

The kernel, the operating system libraries, the OS application runtime, the OS applications, and the application framework are stored in the System Partition, an area of the memory which has a read-only access. This means that any program running on the device will never be able to modify the system source code. By default, only a small group of core system applications and the kernel process run as root, a special user account with the highest level of capabilities on a Linux system. Even with such permissions, a process cannot edit data in the System Partition.

The Android security model is implemented in a two-layer structure:

- A kernel-level sandboxing and protection mechanism to isolate applications called *Application Sandbox* provided through Linux *discretionary access control* (DAC) and *Security Enhanced Linux* (SELinux).

- An application-level permission model implemented by the Android software. It controls access to system resources, such as the possibility to use the camera or communicate through the network, and to application components such as the ability to start services or activities implemented by other apps.

3.1.1 Application Sandbox

The *Application Sandbox* is one of the strongest security concepts in Android implementation. Each application that runs on the system is isolated in its own process so that it can't affect the operating system or other applications and at the same time it can't be accessed from malicious apps. The isolation is managed with DAC in two principal ways. First, it ensures that access to system resources can only happen indirectly by apps through system services. Such services address sharing concerns and manage access control. For certain situations, DAC directly authorizes or prevent apps to access system resources. Second, DAC isolates apps using the kernel features of user ID (UID) and group ID (GID), unique numeric codes associated with each app and assigned to data and processes. This mechanism prevents apps from directly accessing the files of other apps using the kernel interfaces.

The shortcomings of DAC have been resolved by Google introducing the SELinux module. Its main scope is to enforce mandatory access control (MAC), the protocol used to constrain the ability of an object to access or perform actions on another object on the same software environment. The module enforces MAC over every process, including the root processes. The operating principle of SELinux is of default denial, or, simply put, anything not explicitly allowed is denied [15].

Because this protection is implemented at the Kernel level, it is extended to operating system applications and native code libraries. The application runtime, the application framework, the operating system libraries, all applications and any other software running above the kernel its executed within the Application Sandbox. Every Android release introduces new protections to enforce the Application Sandbox, which now features a high number of security mechanisms.

The sandbox is not only necessary between process but also with app data. Store information in world accessible memory can lead to confused deputy vulnerabilities, information disclosure leaks and is the primary target for malware that targets sensitive data. From Android 9 on, apps are not allowed to use shared memory. Instead, a set of features are available to share data: if some data has to be shared with another application, developers can use a specific component called content provider or create a shared folder on the external storage. Content providers are mainly used to share a specific type of information, providing a client-server model with an interface which handles inter-process communication and a controlled access mechanism. External storage instead, is not guaranteed to be always available but can contain data which should be accessible to other apps and to the user.

3.1.2 Permissions

Android implements a security model based on permissions. A permission has the purpose of protecting the user privacy and to let him decide whether an application can use certain device features or access to specific sensitive user data stored on the device. Depending on the permission requested by the app, the system can directly grant it or display a request dialog to the user interacting with the device. A set of predefined permissions exists in the Android framework, but an application can declare its own permission to control who can access its features or components.

Permissions are divided into various protection levels. Whether a permission has to be requested or not depends on its level. The Android permission model defines three protection level available for third-party applications:

- **Normal permissions**

  Normal permissions are related to situations where the app needs to access resources or data that are available outside the sandbox of the application, with low risk for the operativity of other apps or user's sensitive information. Examples of normal permissions are the one to make the phone vibrate or the one to set an alarm. When a permission of this level is requested, the system directly grants it without the need of user interaction.

- **Signature permissions**

  Signature permissions are defined by an application (third-party or not) and are registered in the list of permissions at install time. Applications signed with the same certificate of the declaring app immediately get them granted. Some signature permissions are declared by system applications and can be requested at install time. Examples of signature permissions are the one needed to use the services available to assist users with disabilities, or the one needed to change the device settings.

- **Dangerous permissions**

  Dangerous permissions are related to situations where the app wants to read data that include the user's private information or operate on objects that could affect the system or other apps operations. Examples of dangerous permissions are the one necessary to use the camera of the device, or the one necessary to read the contacts from the device. In order to get a dangerous permission granted, the user has to grant it at runtime from a specific dialog displayed by the system.

To request a permission, an app has to declare it in the AndroidManifest.xml file using the <*uses-permission*> tag, specifying the permission name through the android:name attribute of the tag. If dangerous permissions are requested, the app has to prompt a dialog at runtime in order to get the permission granted by the user.

Permissions can also be used by application's components to enforce who can request and use their functionalities. To do so, a component has to explicitly contain the android:permission attribute when it is declared in the manifest file of the application. Whenever an external app tries to start a component declaring a permission, the system checks if the requesting app has the necessary custom permission granted.

3.2 Exposed application components

All the describe security mechanisms are intended to keep applications as separate as possible within the system, but Android application components are often supposed to interact and communicate with other applications. This imposes the necessity of mechanisms to manage apps interaction and communication. In order to let other applications start activities or services of an app, Android defines a set of attributes related to whether the component should be accessible to other apps or not. When an application component is defined in such a way that other applications can start it, it is considered an exposed component and can be started using an intent.

Three attributes are influent to limit the exposure:

- **android:enabled**

  This attribute can set if the component can be instantiated or not by the system. The value ”true” will let the system create and start the component in the application process while the ”false” value will make the activity or service invisible to other applications and ignored by the system. In the latter case, none of the other attributes are meaningful and the component won't be exposed in any case. If this attribute is not declared, the default value is true.

- **android:permission**

  This attribute declares the name of the custom permission that a client app must have obtained in order to launch the component. If the app doesn't have the necessary permission, the intent is not forwarded by the system to the receiving application and no component will be started. If android:enabled is set to ”true” but the permission attribute is defined, the component is not exposed unless. If the permission requested is of the normal level, then it can be considered exposed since any application could start it without user interaction.

- **android:exported**

  This attribute is the most complex and confusing regarding whether an activity or service is exposed or not. It set exactly if the element can be launched or not by components of different applications, setting it to ”true ” if yes and ”false” if not. When it's set to ”false”, which is the default value, it can not be started by apps even if it defines intent filters. When the value is set to ”true”, the component is exported and can be started using an explicit intent with the class name. Additionally, if intent filters are defined for the element, it can be started with implicit intents. When filters are defined, the value is set to true by default and the component become exposed.

3.1 summarize the conditions necessary to make a component exposed according to the attributes previously described. The ”-” symbol means that the value in the Exposed column will be the same for any possible value of the element in the column. The intent type can be Implicit or Explicit or both, depending on the elements.

When the declaration of an element in the manifest file equals one of the rows with the ”V” mark in the exposed column of 3.1, the application component is exposed and potentially open a breach in the Android security mechanisms of process isolation. The purpose of intents is not only to start components, but they are also intended to transmit messages or data. This means that through them, applications

Table 3.1: Declaration conditions for exposed components

Attribute			Tag	Exposed	Intent type
enabled	permission	exported	<intent-filter>		
false	-	-	-	X	-
true / -	dangerous	-	-	X	-
true / -	signature	-	-	X	-
true / -	X / normal	- / false	-	X	-
true / -	X / normal	-	V	V	I
true / -	X / normal	true	V	V	I / E
true / -	X / normal	true	-	V	E
true / -	X / normal	true	V	V	I / E

let data written by other processes enter into their sandbox. If developers do not correctly validate the input or check their status before a component is started by an intent, a variety of possible vulnerabilities are created.

3.2.1 System and third-party applications

System and third-party applications must follow the same coding rules and structures to be developed. Independently from their role and importance, all the applications have the same set of features to protect themselves and secure their data. This implies that discovering a vulnerability generated by exposed components can have different impacts depending on the type of application.

Third-party applications have limited power on the system, but they can obtain dangerous permissions granted by the user if he trusts them and knows why they need to use a specific feature of the device. When a dangerous permission is granted to a third-party app, the user is not aware that such application could potentially expose components to other apps and indirectly generate vulnerabilities in the object protected by the permission.

In the case of system applications, the risk is even more serious. First of all, system apps are installed on every device sold, which means that a vulnerability generated by an exposed component would affect all the devices on the market. Secondly, an app installed to be part of the system has various dangerous permissions granted by default. They manage the core features of the OS and they need to have special powers to do so. Many Android features are managed by system-app processes, which means that attacking a vulnerable exposed component directly affects the entire service created in the process.

Chapter 4

Vulnerabilities and Exploits

Whenever a secure program is modified, its definition of secure cannot be guaranteed anymore. Android was designed with the goal of creating an open ecosystem enriched by applications, where critical functionalities can be extended or replaced by third-party applications. To meet this goal, the system needed to have mechanisms to go through the limits imposed by the security model. Exposed components create a hole in the sandboxing security mechanism.

Under this assumptions, we decided to analyze whether Android developers are aware of the risk and write secure applications. Furthermore, we investigated if exposed components can create vulnerabilities leading to security and privacy attacks.

4.1 Unexpected intents

When a component is exposed, an application can send an intent to start it. When an application A sends an intent with destination an application B, the system receives the intent and check if an exposed component expecting that intent exists. If yes, the OS starts the component of application B according to the process displayed in 2.2. For both activities and services, the system calls the onCreate() method defined by their class source code. This method is executed in the existing application process, or a new one is created.

If the exposed component is not intentionally exposed, or it is supposed to be called only under certain circumstances, its context may not be correctly initialized. An accessed variable may not be initialized, or an object instance may not exist. These situations can happen, for instance, if the exposed component is supposed to be started by an application after the execution of other components [16]. Such intents can be defined as unexpected intents, due to their characteristic of being sent/received when they are not expected to.

4.1.1 Denial-of-Service

When unexpected intents are sent, the receiving program will raise unexpected exceptions potentially causing the crash of the entire process. When a process crash, the entire app stops functioning and the program exits. If an Android app crashes, the system displays a dialog informing the user that the application has stopped with an option to open the app again as shown in 4.1.

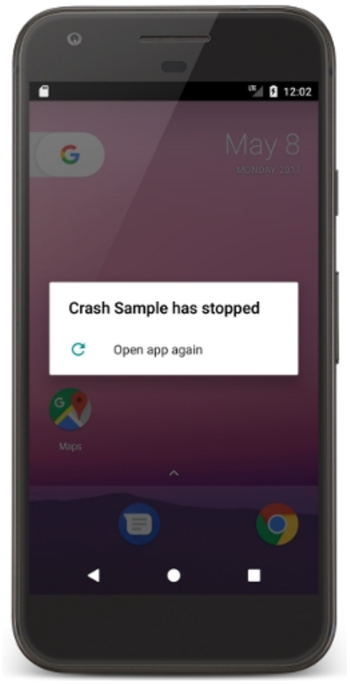


Figure 4.1: System dialog dislpayed when an application crashes [17]

An application doesn't have to be in the foreground or displayed on the screen to crash. Any component, such as a service running in the background to play a song, can fail and crash an app. In such cases, the crash is often confusing because the user was not interacting with the app named in the system dialog.

The situation in which an actor makes a device or a resource unavailable to a user is a type of attack called Denial-of-Service (DoS). Since there is no limit imposed by the Android system on how many times an application can send an intent, a malicious application could repeatedly start a vulnerable app with an unexpected intent making the app unavailable to the user for an indefinite amount of time. This attack not only prevents the usage of the app but slows down the system forcing the OS to use a lot of its resources in the process of restarting the app and displaying the crash dialog every time. Even worst, the user doesn't have a method to understand which app is doing the attack, unless the malicious app keeps the DoS while the user uninstalls every third-party app one at a time until the problem stops.

Supposing that the application provides a service continuously needed by the system or the user and not only used when the user interacts with it, attacking an exposed component of the app will crash the entire process preventing the service to keep working. In this case, the attack will not only prevent the use of an app but also an entire feature of the device. The DoS attack can be also crafted in order to crash the process only under certain circumstances, such as every time that the user requires to use such feature.

4.1.2 Real-life attack: PhoneApp application vulnerability

A relevant example of an exposed vulnerable component was found in the PhoneApp application. The PhoneApp is an Android system application which is in charge of managing the device mobile telecommunications network. The app runs the com.android.phone process, containing all the objects created to keep the device connected through the mobile operator network. Among the components exposed by the application, the EmergencyCallbackModeExitDialog is vulnerable to the DoS attack. The activity declares two intent-filters which can be used to start it and generate the attack: *com.android.phone.action.ACTION_SHOW_ECM_EXIT_DIALOG* and *com.android.internal.action.ACTION_SHOW_NOTICE_ECM_BLOCK_OTHERS*. In the following section of code extracted from the manifest file of the app, can be seen that no attribute is set to prevent the activity from being exposed.

```
....
<activity android:name="EmergencyCallbackModeExitDialog"
   android:excludeFromRecents="true"
   android:label="@string/ecm_exit_dialog"
   android:launchMode="singleTop"
   android:theme="@android:style/Theme.Translucent.NoTitleBar">
   <intent-filter>
      <action android:name="com.android.phone.action.
 ACTION_SHOW_ECM_EXIT_DIALOG" />
      <action android:name="com.android.internal.intent.action.
 ACTION_SHOW_NOTICE_ECM_BLOCK_OTHERS" />
      <category android:name="android.intent.category.DEFAULT" />
   </intent-filter>
</activity>
....
```

PhoneApp AndroidManifest.xml

The PhoneApp component expects to receive such intent only if an object indicating that the phone is in the emergency callback mode (ecm) has been instantiated. The *onCreate* method checks if the device is in the ECM status calling the method isInEcm on an object of the class Phone. Before doing so, it doesn't check whether the object has been instantiated or not probably because the activity is not supposed to be called by other apps in an unexpected state. A subsequent method call on the object raises a *NullPointerException* and crashes the com.android.phone process. The original vulnerable code can be seen in the following block.

```
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);

   mPhone = PhoneGlobals.getInstance().getPhoneInEcm();
   // Check if phone is in Emergency Callback Mode. If not, exit.
   final boolean isInEcm = mPhone.isInEcm();
   ....
```

PhoneApp EmergencyCallbackModeExitDialog.java

The process is automatically restarted by the system but since it manages the phone signal, when the dialog notifying the crash of the application is closed, the device doesn't have any wireless phone network. This condition lasts for a few seconds until the com.android.phone process is restarted and the service restored. If the device is receiving or sending a phone call, crashing the PhoneApp application will force close it. Subsequent starts of the activity will cause repeated crashes of the process, sometimes resulting in a system soft reboot. The danger of the discovered vulnerabilities is that exploiting them to craft an attack is really simple. For example, to generate a DoS attack from the described vulnerabilities is enough to write the following lines of code inside a loop or a conditional statement.

```
Intent i = new Intent();
i.setAction("com.android.phone.action.ACTION_SHOW_ECM_EXIT_DIALOG");
startActivity(i);
```

Code to exploit the PhoneApp application vulnerability

The example provided is not the only vulnerable component exposed by the PhoneApp application but it contains other vulnerable activities and services.

4.2 Permission re-delegation

As previously stated, granting dangerous permissions gives application's components special power on the device. The Android permission model is intended to prevent applications from performing actions against the user will. As defined by [18], permission re-delegation occurs when an application to which was granted a dangerous permission performs a privileged action for another application without the consent. This kind of vulnerability can be considered as a confused deputy attack or, similarly, a privilege escalation attack. In this scenario, the attacked application is the deputy and receives the authority from the user by getting the permission granted. The deputy declares a vulnerable component exposing its functionalities to external applications. The application performing the attack doesn't have the permission that the deputy has. The requester sends an intent to the exposed component, causing the app with the permission to perform an action. The action will be executed because the deputy has the necessary permission and the malicious app succeeds in obtaining a privileged action to be executed.

Permission re-delegation attacks can be more dangerous than the previously described DoS attacks because they can directly impact the user privacy. Moreover, this kind of vulnerabilities can be exploited to generate more sophisticated attacks where more than a vulnerable component is used.

4.2.1 Real-life attack: Camera2 application vulnerability

A clear example of this type of vulnerability is given by the Android Camera2 system application. It is part of the *android.hardware.camera2* package and its function is to provide an interface to individual hardware camera components connected to

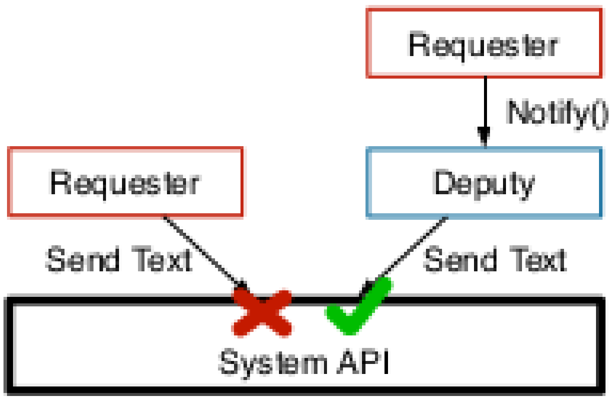


Figure 4.2: Permission re-delegation scheme [18]

the device. The application exposes a component for the *VideoCamera* activity as can be seen from its declaration in the app Android Manifest file. The activity tag contains an intent-filter and no attribute *android:permission* or *android:exported* is used to enforce its exposure.

```
....
    <activity−alias
            android:name="com.android.camera.VideoCamera"
            android:label="@string/video_camera_label"
            android:targetActivity="com.android.camera.CaptureActivity">
            <intent−filter>
                <action android:name="android.media.action.VIDEO_CAMERA" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent−filter>
            ....
    </activity−alias>
....
```

Camera2 AndroidManifest.xml

This application has the dangerous level permission *android.permission.CAMERA* necessary to use the camera. In particular, its onCreate method makes a specific call to open the camera and to start recording a video. Any third-party application can start the activity sending to the system an intent with the action set to *android.media.action.VIDEO_CAMERA* and the device immediately starts recording. In this situation, the camera app becomes the deputy in a permission re-delegation scenario. The following lines of code are enough to create an attack recording a video without the necessary permission.

```
Intent i = new Intent();
i.setAction("android.media.action.VIDEO_CAMERA");
startActivity(i);
```

Code to exploit the Camera2 application vulnerability

This vulnerability can be exploited by a malicious actor to write an app with the ability to spy the user. Every Android device has a proximity sensor capable of detecting when the device is close to an object such as the face, during a phone call, or in a pocket. To read data from a device sensor, no permission is needed since the information received do not give sensitive data. The malicious app can read the data from the sensor and detect when the device is close to an object, which indicates that the user cannot see the display. In this situation, the application sends an intent to start the vulnerable component and the camera app starts recording a video. As soon as the proximity sensor detects that the object is no more close to the device, an intent to display the home screen can be sent to the system. When this happens, the camera app stops recording and the video is saved in the device memory. The malicious app immediately saves the video to its private memory space and delete it from the gallery. In this way, the spy app has recorded audio and video through the device camera both without the user being aware of it and without the camera permission. For instance, if the user makes a phone call without headphones, the entire call audio is recorded alongside the video of what's around the person.

Chapter 5

Vulnerabilities Detection

The AOSP project contains the source code of the entire Android OS implementation. Some of the system applications are included, but others are added when manufacturers customize the code for their devices. Every Android release contains on average more than 100 applications. According to [19], on the Google Play store are available more than 2.6 millions of applications. With such numbers, a manual analysis to detect the vulnerabilities presented is not feasible. A tool to automate the process of detection and testing has been developed in order to understand how widespread the problem is and to find a possible approach to solve it.

5.1 Detection of exposed components

The first step in the analysis of an application is to detect the exposed components among those declared. The Android app coding rules impose that all the components have to be declared in the Manifest file. This implies a standard way of defining components, implemented in section 2.2.1, which allows the use of static analysis on the XML source code.

5.1.1 Algorithm

The analysis starts with the search for the <*application*> tag, which indicates that everything declared from there up to the corresponding <*application*> closing tag is an element of the application. The set of attributes of the <*application*> is parsed, checking if the *android:enabled* attribute is not set to false, otherwise the entire application is not exposed. Next, the components declaration tag of activities or services is searched. As soon as it is found, it set of attributes is parsed. If one of the attributes limiting the component exposure is found, the analysis skips the element until another <*activity*> or <*service*> is detected. In particular, the patterns searched are:

- *android:enabled="false"*
- *android:exported="false"*

- *android:permission*

Note that, when the permission attribute is detected, the component could still be exposed if the declared permission is of the normal level. The list of app components requesting a permission are logged in a separate file to check the permission level at a later time. If none of the listed elements is found, the component is potentially exposed. The research continues detecting if any <*intent-filter*> is declared. At this point two situations can occur:

1. **No intent filters declared**
   If the exported attribute has been set to true explicitly, then the component is exposed and can be started sending an explicit intent.

2. **Intent filters declared**
   The component is considered exposed and can be started sending an implicit intent after extracting the intent action from the filter tag attribute android:name.

5.1.2 Pseudo-code

The following pseudo-code represents the implementation of the algorithm used by the developed tool to find exposed activities.

```
file manifest = applicationApk/AndroidManifest.xml
# read the manifest file line by line
while manifestline = manifest.readLine():

    # [step 1] scan <application> tag
    if '<application' in manifestLine:
        # read line from file until the application tag is closed
        while manifestLine != "/>":
        # no exposed components if the application is disabled
        if 'android:enabled="false"' in manifestLine:
            exit

    # [step 2] scan <activity> tag
    if '<activity' in manifestLine:
        # read all <activity> tag attributes
        while manifestLine != "/>":
            if 'android:enabled="false"' in manifestLine:
                # skip to next component
                break
            if 'android:exported="false"' in manifestLine:
                # skip to next component
                break
            if 'android:permission="...."' in manifestLine:
                requirePermission = True

        # [step 3] scan for <intent−filter> tags
        # read line from file until the activity component tag is closed
        while manifestLine != "</activity>":
            if '<intent−filter' in manifestLine:
                hasIntent = True
                intentFilterList.add(filter)

        # [step 4] write activity and related data
        if requirePermission:
            logPermission.write(activity)
            exit
        if hasIntent:
            # if it has an intent−filter, can be started with an implicit intent
            logImplicit.write(activity)
        else:
            logExplicit.write(activity)
```

Pseudo-code to detect exposed components of an application

5.2 Identification of vulnerable components

The second part of the vulnerability detection process requires detecting which of the exposed components are actually vulnerable. Since the possible reasons for an applications crash are almost infinite, the static analysis is not enough. The ideal method to perform the study by executing a program is the dynamic analysis. For this purpose, an Android command-line tool is used in our process.

5.2.1 Android Debug Bridge

The Android Debug Bridge (adb) is a command-line tool that can be used to communicate with an Android device. It consists of a client-server program with three components:

- A **daemon** (adbd), which runs as a background process on the device. It is in charge of running commands received from the server on the device.
- A **server**, running on the machine which manages the transmission of requests and data between the client and adbd.
- A **client**, the command-line terminal on the machine where commands are submitted to the server.

With adb, commands can be used to make calls to the *activity manager* (am), an Android class managing components, which can start an activity and a service using intents. In order to start a component, the following commands have to be used in the command line:

```
    adb shell am start −a "implicit.intent.action"
    adb shell am start −n "explicit.intent.action"

    adb shell am startService −a "implicit.intent.action"
```

```
adb shell am startService −n "explicit.intent.action"

adb shell am startforegroundservice −a "implicit.intent.action"
adb shell am startforegroundservice −n "explicit.intent.action"
```

Adb commands to start components using the activity manager

The "-a" option has to be followed by an intent action, while the "-n" option has to be followed by a specific component name in a fixed format: the application package name as a prefix, followed by the class name. For instance, "com.example.app/.ExampleActivity", where the forward slash separates the two elements.

A second adb feature used in the dynamic analysis part is the Logcat tool. Android has a special log where every process writes its status or useful routine messages. When an application crashes, the system writes on the log a set of information that can be used by developers to understand why the crash has happened. The first functionality necessary is the command to clean the crash channel of the log, while the second is the command to read the content of the same channel, and the syntax is the following:

```
adb logcat −b crash −c
adb logcat −b crash −d
```

Adb commands to manage the Logcat crash channel

where the "-c" option stands for "clear", and the "-d" option stands for "dump" the log.

Finally, a last useful feature of adb is the possibility to simulate input touch on the screen of the device. The command only requires the screen coordinates expressed in the number of pixels of distance from the bottom left corner of the touch area and it directly submits an input. The format for the command is the following:

```
# X and Y must be integer numbers
adb shell input tap X Y
```

Adb command to simulate a user touch on the screen area

5.2.2 Testing with adb

All this tools and features can be combined together to automatically analyze the previously selected exposed components, testing if they are vulnerable. To do so, the first step is to clear the device screen and display the home screen to start from a neutral situation. A user input has to be simulated over the home button, giving a couple of seconds to the device to display the home screen. The next step is to clear the crash log, in order to be sure that what it contains when we will read it is only related to an eventual crash happened in the last few seconds. Then, the actual testing command has to be submitted, sending an intent to start an exposed component. Again, a few seconds should be given to the system to start the component and let it he following code is the portion of the script implementing the check.

```
# Click home button
adb shell input tap 540 1855
sleep 2

# Clear Logcat crash channel
adb logcat −b crash −c

# Start the component
adb "$action"
sleep 4

# Dump Logcat crash channel
adb logcat −b crash −d > logDump
if [ logDump != "" ];
then
    echo "The␣exposed␣component␣is␣vulnerable"
fi
```

Script code to automatically test and detect if an exposed component is vulnerable

Chapter 6

Analysis Tool

Software testing is a process conducted on computer programs to provide information about quality and reliability. It involves the execution of system and software components to evaluate one or more properties. The manual testing is infeasible in most of the software due to the number of lines of code and randomness of program states and inputs. Test automation requires the use of computer programs which are separate from the tested software to execute analysis and collect information on the results.

In our research, we developed an automated tool which can be used by AOSP developers and app developers to analyze their application code against the flaws and vulnerabilities arising from exposed components. The tool provides a command line interface with different analysis options and has been designed with a modular design for benefits such as augmentation and exclusion.

6.1 Tool structure

An efficient software analysis tool has to be simple and easy to use with few requirements on data preparation. The developed tool requires only the path to a folder containing Android applications files and a connected device (real or emulated), to perform all the necessary actions to prepare the data for the analysis. For instance, the input folder could be the AOSP source code of an Android version with a connected device running the same release, and the tool will produce a complete report on all the applications of the corresponding Android version.

6.1 represents the tool architecture containing all modules involved in the analysis process as well as input and output data.

The architecture consists of four modules developed to interact and share data. Specifically, the output files generated by a module, are in a format that the next module can interpret. The input directory can contain any type of file, but the analysis starts with the Android application file format called apk. During the tool execution, different file formats are generated. At the end of the analysis, the results are collected in text files that can be used by the tester for further analysis. In the following sections, a detailed description of each module is provided.

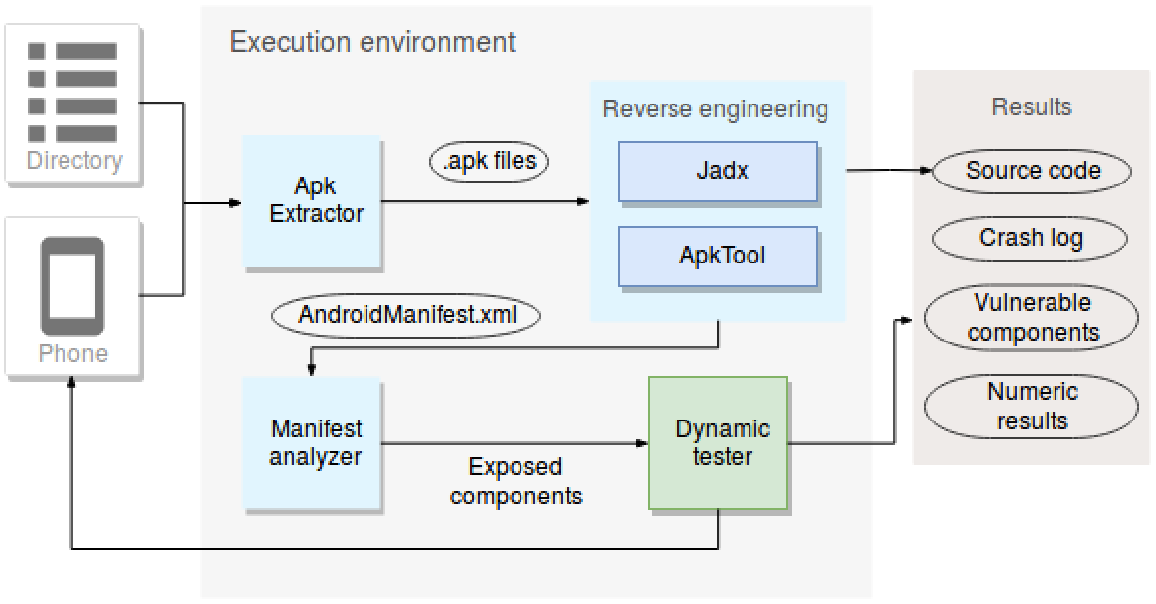


Figure 6.1: Auotomated analysis and detection tool architecture

6.1.1 Apk Extractor

The first step of the analysis is the extraction and preparations of the files used for the test. This module requires to search within a directory tree, find file paths and move files. An efficient way to do the listed actions is to use a bash script. Three different sources for applications can be provided:

- **Apk files**

  The apk file format is the standard format used to build an Android application. When the user downloads an app from the Google Play store, an apk file is retrieved. More in general, all third-party apps are provided as apk files, whether they come from the official store or from another source. These files are searched within the provided directory by their extension ".apk". The list of files is created recursively searching within the root directory using the bash command "find".

- **AOSP source code**

  The source code provided by the AOSP is open-source and it is not compiled. This means that the tool can directly access the applications files without any additional manipulation or decompilation. In this case, to detect an application folder, the Android manifest location is used.

- **Android device** (physical or emulated)

  Not all system applications come from the AOSP. Each manufacturer installs its own developed system applications to add features and to let the user handle specific Android components. Such applications are extracted by the tool from the connected device. The adb tool is used firstly to create the list of applications which comes with the device, and secondly to extract the apk files.

The following code shows the main commands used by the tool in the Apk Extractor module.

```
# Generate list of apk files in the source directory tree
find /path/to/directory −name "∗.apk" > outputFile

# Generate a list of apk files present in the connected device
# Similar to the previous command but it's executed in a shell process on the device
  created using adb
adb shell su < find system/ −name "∗.apk" > outputFile
```

Commands used by the Apk Extractor module

When an AOSP directory tree is provided as a source for the analysis, the tool automatically extracts the additional apks from the connected device and merge them together. In this way, the numbers presented in the results are comprehensive of the entire set of applications provided by the OS for the Android release.

6.1.2 Reverse engineering module

Android PacKage is the package file format for the distribution of a mobile app for the Android operating system. In the apk package are collected the set of files obtained from the compilation of a program for Android. In the compilation process, the source code is first processed by the Java compiler obtaining the byte-code format, corresponding to files with the 'class' extension. Next, the DEX compiler transforms the content of the .class files in dex byte-code, a byte-code format for the Android Runtime. To decompile the byte-code to a human-readable format, two tools can be used: Apktool and Jadx.

- **ApkTool** is a tool for reverse engineering binary Android apps. It decodes Java resources to the Smali format, an intermediate format between the byte-code and the Java source code. The tool also decompiles XML files and return them to the original format. With ApkTool, the AndroidManifest file of an application can be retrieved and analyzed to perform the search of exposed components.
- **Jadx** instead, is a tool for decompiling apk or dex byte-code files. After the decompilation process, the files are restored to a really close version of the original Java source code. The code retrieved is stored in the results folder and can be used to understand and fix the vulnerable methods.

6.1.3 Manifest analyzer

This module implements the exposed components detection. The Reverse engineering module has decompiled all the apk files in the directory tree preparing the files for the Manifest analyzer. A file containing the list of AndroidManifest.xml paths is created with the bash find command as previously described. Next, each manifest is given as input to the code implementing the algorithm explained in section 5.1.1. The process also collects statistic data about different information:

- Number of manifest files analyzed
- Number of exposed components detected
- Number of components startable by implicit intents
- Number of components startable by explicit intents
- Number of components requiring permissions

The module generates a set of output files with information about the exposed components and special files necessary as sources for the dynamic analysis performed by the last module. The most important file contains details about each exposed component. In particular, the manifest path, the package name, the app name, and the component name are collected.

6.1.4 Dynamic tester

The last module implements the dynamic analysis. This module needs to read data from two files written by the Manifest analyzer in a specific format. One file contains the actions declared by the exposed components that can be started by an implicit intent. The actions are expressed as a string and must be used with the exact value coded in the manifest file. The second file contains the component name as specified in 5.2.1. These files are used as input for the program implementing the algorithm explained in 5.2.2. During the analysis, the information collected from the Logcat when a vulnerable component is found are saved in a crash log. Thanks to this log, when a developer runs the analysis on its application, he can not only detect the existence of the problem but also easily understand what should be fixed.

6.2 Analysis reports

Each module executed during the analysis generates output log file which can be used for different reasons. The program creates a folder in the directory where the tool is run with the name of the analyzed directory with an additional underscore followed by 'a' or 's' depending if the user asked to analyze activities or services. Within this directory, at the end of the analysis the following files can be found:

- **crash_actions.txt**

  It contains the list of actions necessary to start only the vulnerable exposed components. The list is divided into actions for implicit intents and components name for explicit intents.

- **crash_report_e.txt**

  It contains the lines written by the OS on the Logcat crash channel when each of the vulnerable components causes the crash of an application process. The reports contain the list of function calls executed up to the crashing method, including name and line numbers of the java source file. This file includes only logs of vulnerable activities crashing with explicit intent.

- **crash_report_i.txt**

  It contains the same information as the previous file but related to the vulnerable components crashing with implicit intent.

- **exposed_components_e.txt exposed_components_i.txt and exposed_components_perm.txt**

  They contain detailed information about the exposed components, such as manifest path, package name, app name, and class name. The third file contains the exposed components to be checked for the level of permission.

- **results.txt**

  The numeric result collected during the analysis.

- **test_components_e and test_components_i**

  These files have a special format and are only used by the Dynamic tester module.

6.3 User Manual

The following list contains the list of requirements to run the tool:

- Machine Hardware: no specific requirements
- Operating system: Unix-like OS / macOS
- Python 3.6
- Android Debug Bridge tool (included in the Android SDK Platform-Tools package or downloadable as a standalone program)
- Logcat tool (included in the Android SDK Platform-Tools package)
- ApkTool tool (open-source software available on GitHub)
- Jadx tool (open-source software available on GitHub)
- Android device (optional for the static analysis)

Some parameters of the tool must be set before running the program. In the extract_APKs.sh file, the 'adb' variable has to be set to the path containing the ADB tool sources. In the 'activities.py' and 'services.py' files, the folder where the result files are created can be set. The default location is within the folder in which the tool is run. No other parameter has to be set to run the program.

To run the tool, a set of options is available as presented in the following block.

```
# List of options for the tool usage

# Analyse all components of the applications contained in the
# directory provided
/tool/directory: $ tool /directory/containing/apks

# Analyse activities of the applications contained in the
# directory provided
~/tool/directory: $ tool −a /directory/containing/apks

# Analyse services of the applications contained in the
# directory provided
~/tool/directory: $ tool −s /directory/containing/apks

# Analyse components of the applications contained in the AOSP
# version provided and in the connected device
~/tool/directory: $ tool /path/to/AOSP

# Analyse activities of the applications contained in the AOSP
# version provided and in the connected device
~/tool/directory: $ tool −a /path/to/AOSP

# Analyse services of the applications contained in the AOSP
# version provided and in the connected device
~/tool/directory: $ tool −s /path/to/AOSP
```

Tool commands

Chapter 7

Results

The final goal of the research is not only to understand which problems can arise with exposed components but also how relevant and widespread the vulnerabilities are. To answer these questions, the tool has been used to analyze a diverse set of applications. With the results obtained, several conclusions can be drawn as well as possible approaches to reduce the risks. Below, the results are presented and interpreted.

7.1 Analysis setup

7.1.1 Android operating system

The first set of analysis has been carried out on different Android versions. As shown in 1.1, the OS releases have a various distribution on the market. The code-line verisions analyzed have been selected in order to support the majority of the devices. The oldest release *Marshmallow*, with a distribution of 21.3%, is not supported by Google anymore but still represents almost a quarter of active devices. *Nougat* and *Oreo* together are installed on 50% of the Android devices and are still updated. Finally, the latest release of Android called *Pie* has also been tested since it will be supported for at least 3 years from August 2018. 1.1 shows data collected on October 26 2018 and do not include the latest version which has been announced on August 6 and its distribution was still under 0.1%.

The source code has been downloaded from the official Git repository hosted by Google. The tests have been performed on different devices depending on the release, some on physical devices and other on emulated devices. In both cases, the devices were the Google official devices called Pixel. Among all the devices on the market, they run the Android version closest to the AOSP original source-code. This implies that most of the vulnerabilities detected are present on all the Android devices running the same version. Table 7.1 summarizes the devices used for each analyzed version.

Table 7.1: List of devices used for testing

Codename	**Android version**	**API**	**Device name**	**Device type**
Marshmallow	6.0.0	23	Google Pixel 2	emulated
Nougat	7.0.0	24	Google Pixel 2	emulated
Nougat	7.1.1	25	Google Pixel 2	emulated
Oreo	8.0.0	26	Google Pixel 2	emulated
Oreo	8.1.0	27	Google Pixel 2	physical
Pie	9.0.0	28	Google Pixel 1	physical

7.1.2 Third-party applications

Third-party applications can be downloaded as apk files. Various websites and repositories implement services to publish or download apps. The Android official applications market is the Play Store. Google reviews every app before authorizing its publication on the market. Android includes also a built-in malware protection called *Google Play Protect*. These software are not perfect and do not consider all possible vulnerabilities. To understand if the exposed components generate vulnerabilities in applications different from the system apps, we used the tool to analyze a large set of applications published by independent developers.

The application packages analyzed have been downloaded from the Google Play store. To have relevant results, the number of apps cannot be too small and a manual download would be infeasible. To automate the download process an unofficial API for the store has been used. The project is open-source and available on *GitHub*, with the name of *Google Play python API* [23]. It contains an unofficial interface to communicate with the Google servers emulating the packages sent by the official store application installed with Android. The Google store organizes the applications in categories and numbers of download. The selection criteria were to test only free applications from all categories, with more than a hundred thousand downloads. Third-party applications are usually available for the latest Android release but are compatible with previous versions. The analysis has been executed on a physical *Google Pixel 2* running Android 8.1. A total of 1496 applications have been downloaded and tested.

7.2 Analysis results

7.2.1 Android OS applications results

The results presented group together the applications from the AOSP source code and the device itself. The apps defined in the AOSP code, are not always installed on the actual device. A set of them is only for testing by Google developer and others are deprecated versions not used anymore but kept for compatibility. This is clear considering the number of manifest files analyzed for each version. It is increasing of hundreds at every release. The results are relative to the only set of apps that are actually installed on a device. The number of exposed activities among the Android releases does not variate too much, while the number of exposed services increases regularly. The exposed services are more protected by permissions compared to the activities. Considering that the number of exposed services is approximately half of the number of exposed activities, the services protected by permissions are three times more than the activities. The vulnerable components increased considerably after Android *Nougat*. This could show that some of the changes in the architecture generated a high number of issues in the applications framework. Starting from *Oreo*, Android releases have an average of 12.8% of vulnerable components among the exposed ones. It is a relevant result considering that it refers to system applications, implementing features of the OS itself.

Google releases major updates of the same Android version when a set of important updates are necessary. It is usually for security reasons and to update important bugs that arose after the first release to the public. Comparing the differences in results between two major updates of the same release, for instance between 7.3 with 7.4 and 7.5 with 7.6, show that the number of vulnerable components does not really reduce. This can be for two reasons. Firstly, this kind of vulnerability is not detected simply by a user interacting with the device. Secondly, if some of them get fixed, new features introduced could create different issues keeping the numbers stable.

Table 7.2: Results for components in Android 6.0.0 Marshmallow

Component	**Manifest files**	**Exposed components**	**Implicit intents**	**Explicit intents**	**Requiring permissions**	**Vulnerable**
Activities	1469	541	413	95	33	24
Services	1469	242	142	26	74	10

7.2.2 Third-party applications results

The results on third-party applications refer to apps with at least a hundred thousand downloads. Apps with such numbers are used from many people and in most of the cases are probably developed by companies which are specialized in Android

Table 7.3: Results for components in Android 7.0.0 Nougat

Component	**Manifest files**	**Exposed components**	**Implicit intents**	**Explicit intents**	**Requiring permissions**	**Vulnerable**
Activities	1681	688	506	142	40	27
Services	1681	310	155	32	123	12

Table 7.4: Results for components in Android 7.1.1 Nougat

Component	**Manifest files**	**Exposed components**	**Implicit intents**	**Explicit intents**	**Requiring permissions**	**Vulnerable**
Activities	1759	664	480	141	43	45
Services	1759	336	154	30	152	24

development. For this reason, they should be less vulnerable since the number of components in such applications is way smaller than the one in OS applications. Additionally, third-party apps require less interaction between their components and those of others since they are not usually intended to generate features for other applications.

The analysis of the entire set of downloaded apk files detected a total of 116 applications vulnerable because of exposed activities and 132 because of exposed services. The 6.85% of applications expose at least a vulnerable activity and the 7.79% a vulnerable service. The number of exposed components, on average, is of 4 per application. As predicted the number of vulnerable third-party apps is lower compared to the system apps, but the percentage is still pretty high. According to the results, one every X downloaded apps with the same parameters of the ones tested could be vulnerable to a DoS attack.

Table 7.5: Results for components in Android 8.0.0 Oreo

Component	**Manifest files**	**Exposed components**	**Implicit intents**	**Explicit intents**	**Requiring permissions**	**Vulnerable**
Activities	2044	658	470	141	47	85
Services	2044	396	163	53	180	60

Table 7.6: Results for components in Android 8.1.0 Oreo

Component	**Manifest files**	**Exposed components**	**Implicit intents**	**Explicit intents**	**Requiring permissions**	**Vulnerable**
Activities	2158	690	496	144	50	74
Services	2158	397	164	55	178	70

Table 7.7: Results for components in Android 9.0.0 Pie

Component	**Manifest files**	**Exposed components**	**Implicit intents**	**Explicit intents**	**Requiring permissions**	**Vulnerable**
Activities	2728	668	454	161	53	79
Services	2728	434	198	51	185	55

Table 7.8: Results for third-party applications from the Google Play store

Component	**Manifest files**	**Exposed components**	**Implicit intents**	**Explicit intents**	**Requiring permissions**	**Vulnerable**
Activities	1694	4051	1983	1758	310	220
Services	1694	3210	827	444	1939	261

Chapter 8

Conclusion

The research started with the assumption that Android components definition can possibly lead to unwanted situations generating vulnerabilities. To understand it the problem was real and popular, we reviewed important Android definitions and concepts to understand where the issues could come from. After understanding the faults of the possible problems, we created a tool to automate the process of analysis and detection. Finally, the results have been studied to answer the initial suppositions and to give a general overview of the current situation of the problem.

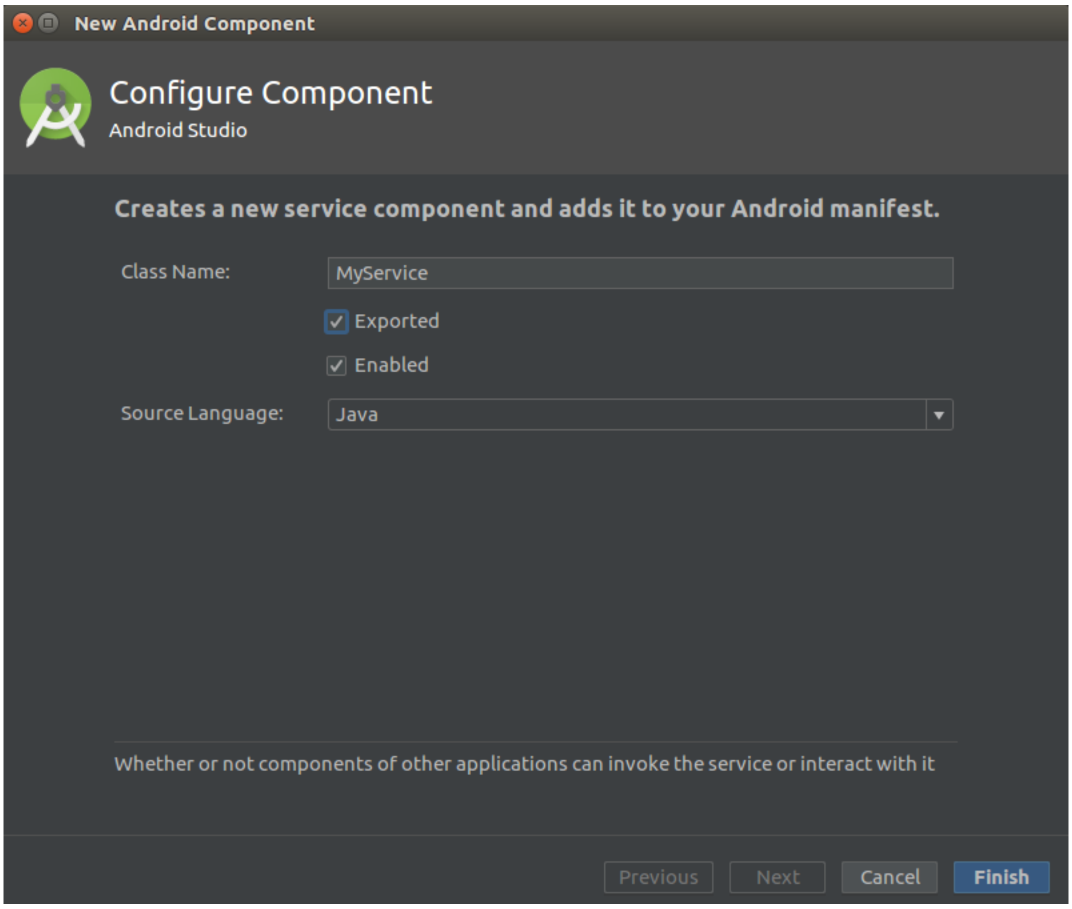
The results obtained with the tool demonstrate that the problem actually exists and that vulnerabilities can be generated if components are not defined with attention. Specifically, have been presented two possible vulnerabilities arising from a bad exposure of an application component. Was also demonstrated that the vulnerabilities could be potentially exploited to generate real-life attacks against the user device and privacy.

The tool should be used by Android developers as well as by independent developers to detect if their software is vulnerable and to easily understand how to fix the issues.

8.1 Proposed solutions

The Android OS will always need let applications expose some of their functionalities to other applications. The system architecture itself is developed to share and control every feature of the system using applications components. The solution can not be to remove the possibility of exposing components to other processes. Instead, what we think two actions should be done:

- Google should make developers more aware of how components must be declared to develop safer applications. The first step for a component creation in the official developer IDE Android Studio shows a dialog with specific checks for the component exposure. Such checks have the default values to expose the component. The developer should manually select the option instead of having to deselect it to un-expose the component. Figure 8.1 shows the dialog displayed in Android Studio when a new service class is created.


New Android Component

Configure Component
Android Studio

Creates a new service component and adds it to your Android manifest.

Class Name: MyService

✓ Exported

✓ Enabled

Source Language: Java

Whether or not components of other applications can invoke the service or interact with it

Figure 8.1: Android Studio dialog for service component creation

Additional information should be also provided on the online documentation made available by Google on the Android developer website. It is the main source for a developer to learn about the system and it doesn't clearly state how dangerous and exposed component could be. Only a few lines are used to describe that if the android:exported attribute is not declared but an intent-filter is, the component is automatically exposed. This is a pattern found in many vulnerable applications. Developers should not use an intent-filter if they want to share the component only with their app but they should use explicit intents which are a safer option.

- Android Studio should include a software component to help developers run security checks on their applications before publishing them. The feature should implement the checks and tests developed in our tool. The analysis of a single application requires a low running time for completion and can give an exhaustive set of information to help developers quickly fix possible vulnerabilities. The tool could be optimized to use the emulator integrated into Android Studio and do both the static and the dynamic analysis.

8.2 Fixes and achievements

The PhoneApp vulnerability has been one of the first discovered with a high impact on the system. For this reason, it has been reported to the Google security team through the appropriate service. The Android security team has assigned a moderate severity level and fixed the problem on all their devices with the first security update release in the month following the report.

The vulnerability has been officially recognized and has received the *CVE-2018-9447*. *Common Vulnerabilities and Exposures* (CVE) is a system providing a list of entries for publicly known cybersecurity vulnerabilities. Google released the security patch an August 2018 and disclosed the vulnerability on the relative bulletin. In particular, the issue has been reported as of type Dos with a moderate severity, and the updated AOSP versions are 6.0, 6.0.1, 8.0, 8.1. Manufacturers have been notified of the issue before the public disclosure and can choose to incorporate the fix as part of their device updates. The report has also been rewarded by the Google Security rewards committee.

At the time of writing, the permission re-delegation vulnerability found in the CameraApp application have been reported. The Google security team has filed an internal report to investigate and fix the issue. It has been marked with the moderate severity which means that a fix will probably be included in a future security update.

Finally, the tool will be used to collect information about the other vulnerabilities and a report will be submitted for each of them.

8.3 Future works

Our analysis demonstrates that exposed components can generate weaknesses in the Android security system. We believe that they are a starting point for the generation of various vulnerabilities which are not limited to the ones we presented. This problem could be further investigated since this and previous researches do not cover all the possible attack points.

8.3.1 Complete analysis of Android devices

The results obtained by this research up to this point are limited to the Google official devices. Each manufacturer develops a specific Android customization for their devices, adding features and applications. The tool will be used to analyze all vendors Android releases to obtain an overview of the problem considering the entire market. Since this vulnerabilities are not decreasing with the new releases of the Android OS, the analysis will be repeated at each new version of the system to provide constant monitoring and reports, essential for the user security.

8.3.2 Auotomatic detection of permission re-delegation

Permission re-delegation vulnerabilities, at this point of the research, are limited to manual detection. We believe that this aspect of the analysis can be studied and upgraded to use methods including both static and dynamic analysis to automate the process. In particular, the use of existing software for Android analysis could be integrated to improve this limitation and possibly demonstrate the existence of more vulnerabilities generated by exposed components.

8.3.3 Creation of a complete tool

Similar works ([20], [21]) have been presented in the past to detect related problems in inter-application communications, components interaction, or similar problems but with different approaches. The Android developers community could obtain great advantages from the availability of a free open-source tool including a set of features to analyze their applications before the release. The ideal tool would give the possibility to analyze an application against all the known issues which can be generated by applications components. It could also implement the different testing approaches presented by researchers, to give the highest possible coverage. To boost this work, the tool developed in our research will be made available as a public open-source project on a GitHub repository.

Bibliography

[1] 2014 Mobile Threat Report, https://blog.lookout.com/mobile-threat-report-2014

[2] Y. Zhou, X. Jiang, "Dissecting Android Malware: Characterization and Evolution", 2012 IEEE Symposium on Security and Privacy, DOI 10.1109/SP.2012.16

[3] Distribution dashboard, https://developer.android.com/about/dashboards/

[4] Mobile operating system, https://en.wikipedia.org/wiki/Mobile_operating_system

[5] Android (operating system), https://en.wikipedia.org/wiki/Android_(operating_system)

[6] Android stack, https://developer.android.com/guide/platform/images/android-stack_2x.png

[7] System and kernel security, https://source.android.com/security/overview/kernel-security.html

[8] Android API Levels, http://www.dre.vanderbilt.edu/~schmidt/android/android-4.0/out/target/common/docs/doc-comment-check/guide/appendix/api-levels.html

[9] Platform Architecture, https://developer.android.com/guide/platform/

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

[11] Android activity lifecicle, https://developer.android.com/images/activity_lifecycle.png

[12] Introduction to Services, https://developer.android.com/guide/components/services

[13] Introduction to Services, https://developer.android.com/images/components/intent-filters@2x.png

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

[15] Stephen Smalley, Robert Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android", DOI 10.1.1.391.2279

[16] Chiaramida Vincenzo, Pinci Francesco, Buy Ugo, Gjomemo Rigel, "AppSeer: Discovering Flawed Interactions Among Android Components", Proceedings of the 1st International Workshop on Advances in Mobile App Analysis, Montpellier (France), pp. 29-34, DOI 10.1145/3243218.3243225

[17] Crashes, https://developer.android.com/topic/performance/vitals/crash

[18] Felt Adrienne Porter, Wang Helen J., Moshchuk Alexander, Hanna Steven, Chin Erika, “Permission Re-delegation: Attacks and Defenses”, Proceedings of the 20th USENIX Conference on Security, San Francisco (California)

[19] Google Play Store: number of available apps 2009-2018, `https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`

[20] Daniele Gallingani, Rigel Gjomemo, “Static Detection and Automatic Exploitation of Intent Message Vulnerabilities in Android Applications”

[21] Chin Erika, Felt Adrienne Porter, Greenwood Kate, Wagner David, “Analyzing Inter-application Communication in Android”, Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, Bethesda, Maryland, USA, pp. 459-466, DOI 10.1145/1999995.2000018

[22] Hay Roee, Tripp Omer, Pistoia Marco, “Dynamic Detection of Inter-application Communication Vulnerabilities in Android”, Proceedings of the 2015 International Symposium on Software Testing and Analysis, Baltimore, MD, USA, DOI 10.1145/2771783.2771800

[23] Google Play Unofficial Python API, `https://github.com/NoMore201/googleplay-api`