POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Reliability Issues among Android Components: Analysis, Testing and Solutions

Relatori
prof. Antonio Lioy
prof. Ugo Buy

Vincenzo Chiaramida

Anno accademico 2017-2018

To my family, for always believing in me,

To my dearest and beloved friends, for supporting me even in the darkest moments,

Thank you.

Summary

Android is the most widespread mobile operating system in the world. Introduced by Google in 2008, it has been adopted by several companies to power their devices, including smartphones, tablets and wearable objects.

As an open source project, the Android source code is publicly available for download, allowing everyone to explore and study its working mechanisms. However, the enormous diffusion and the availability of this operating system have made Android subject to a very wide range of different attack vectors, that negatively affected the user experience under several aspects. The reliability is one of the aspects of the operating system that have been compromised most frequently, through the exploitation of various methodologies. This has lead Google to focus on the improving of specific components and modules at every new released version of Android. Nowadays, most of the attacks that once represented serious threats have been completely neutralised. Nonetheless, the perfectly protected operating system does not exist.
In our research, we focused on the last stable version of Android, nicknamed *Android Oreo*. We studied various components of Android applications and their interaction principles, and we tried to expose some potential design and implementation vulnerabilities that could affect the reliability and security of this operating system.

We first studied the elements of novelty that Android Oreo brought with respect to its previous version. This lead us to the uncover of a design flaw affecting the interactions among applications and specific components, called *foreground services*. Given the systematic presence of a pattern found in those applications subject to this flaw, we decided to build and implement a tool for the specific purpose of labelling input applications as vulnerable or safe. We then proceeded with the evaluation of the obtained classification results through some tests performed both on an emulated and a real device. The great majority of the tests confirmed the labels assigned by the tool, demonstrating how all the most popular Android applications are affected by this issue, including system applications. For this reason, we decided to report our findings to the Android Security Team, through the Google IssueTracker platform.

Next, our research focused on the feasibility of a potential attack that tries to exploit the foreground services flaw. This lead us to the discovery of two additional issues affecting two specific Java classes that are found in Android, named *Context* and *ClassLoader*. We demonstrated how the exploitation of some features of these classes can represent two different but equivalent ways to implement a real life attack against those vulnerable applications.

We provided some ideas and suggestions that could mitigate and fix the exposed

problem, and we realised how the next version of Android, whose code name is still unreleased at the time of writing, is actually moving toward the solution of the foreground service flaw. Finally, we briefly described another result of our researches in the field of class loaders, that could represent the starting point for future works.

Acknowledgements

I want to thank my committee: Prof. Buy, Prof. Gjomemo and Prof. Lioy for their time, attention and interest.
Beside my committee, I would like to thank my advisors, Prof. Buy and Prof. Lioy, whose dedication, enthusiasm and knowledge have always helped me during the making of this work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Goal of this thesis

Over the past years, mobile operating systems have been growing continuously and increasingly providing services that became essential to everyone's everyday life. We rely on our mobile devices so much that they now represent one of the main keepers of our personal data. For this reason, security and reliability issues affecting mobile operating systems must be treated with the same degree of awareness and seriousness as in any other OS. The aim of this work is to study the current state of confidentiality, integrity and availability of the Android OS, the most widespread mobile OS in the world, and to highlight some potential threats to these factors.
We then show how an attacker could exploit these issues and finally we propose solutions aimed at mitigating some of the flaws found.

1.2 Deliverables

Our research identified a new vulnerability, exploitable through a Denial of Service (DoS) attack, that is found in the whole set of tested applications. We built an automated tool for detecting the potentially vulnerable applications, that made the testing phase a lot more efficient, especially on a large scale. The results of these tests have been reported to the Android Security Team through the Google Issue-Tracker platform, and they are currently investigating the issue [7].
Moreover, this research highlighted some potential flaws concerning two special Java classes, *Context* and *ClassLoader*, for which some practical applications are presented.

1.3 Research environment

All our research has been performed directly on the source code of the last available stable Android release as of this writing, called Android Oreo. The main development tool is the official Android IDE, Android Studio, while all the tests have

been performed both on an emulated device (a Google Pixel 2) and a real device (a Google Pixel), both running Oreo with API levels 26 and 27.

1.4 Road map

In order to expose the results in the most accessible way, some basic Android background is described in Chapter 2, with the definition and explanation of all the concepts used. Chapter 3 is dedicated to the current state of the art in Android security and threat models. Chapter 4 and Chapter 5 are dedicated to the description of the main discovered vulnerability and our implemented tool. After that, we present how such a vulnerability could be exploited in a real use-case, introducing the *Context* and *ClassLoader* flaws. In Chapter 7, some proposed solutions are described in attempt to mitigate the effects of a malicious application trying to exploit this issue. In the final chapters an additional research about *ClassLoader* objects is introduced, that could represent the starting phase of some future works, together with some final considerations.

Chapter 2

Background

Android, introduced by Google in 2008, is the most widespread mobile operating system (OS) in the world, with about the 86% of the worldwide mobile market share [8]. The key of its diffusion is its openness to a wide variety of devices from a lot of different companies. Moreover, it is an open source project, thus representing a perfect choice for a research study like ours. It is based on a slightly modified Linux kernel, and its main official programming language is Java. However recently Google added the possibility to also use a new programming language called *Kotlin* [9].

2.1 Basic principles

2.1.1 The callback mechanism

Writing an Android application never means writing a standard program, that is a program with a main entry point of execution. Instead, the role of the developer is to write code that satisfies specific requirements of the OS, so that the main process, called *system_process* and started when the device is booted, can continuously interact with the application through what is known as the callback mechanism.

For example, the system_process has the responsibility of handling the lifecycle of every application, from its creation to its destruction. For this reason, it will call special methods in the context of the application at the right time. These methods are specific to every class that represents a component within an application, and their implementation is left to developers, that will adapt them according to their applications purpose. Indeed, in most of the cases, these special methods are declared as abstract, or they have a default implementation that developers must modify.

In this sense, an Android application is deeply coupled with the OS, because it satisfies specific design and implementation requirements that allow the system_process to communicate and interact with it. These sets of methods that the system_process will execute are referred to as *callback methods*.

2.1.2 API Levels

Every new version of Android, characterized by a unique code name, is released with a set of modifications that directly affect the source code. These changes can include:

- The introduction of new methods and variables, or the deletion of those considered obsolete;
- The modification of the prototype of existing methods;
- The deprecation of methods and variables that will be completely deleted in future versions, and that are left only for backward compatibility.

The overall collection of the methods declared in every Android version represents the platform Application Programming Interface, or API. It follows that new Android releases correspond to new API levels, that contain all the new features of the new version. Moreover, it can happen that the API is updated within the same version of Android. This is the reason why the name of the current version alone is usually not enough to identify which Android API level we are referring to. For example, Android Jelly Bean, corresponding to Android 4.0, contains three different API levels from 16 to 18. In the same way Android 7.0, nicknamed Nougat, corresponds to API levels 24 and 25, while Android 6.0 (Marshmallow) only includes API level 23 [10].

When creating an application, developers can decide the *minimum* API level that they want to target. This will provide them with the correct collection of methods available to implement their functions, as well as defining the set of devices compatible with their application, according to the version of the OS installed. The latest Android version, whose code name is Android Oreo, is linked to API levels 26 and 27.

2.1.3 The proxy pattern

The constant interactions among applications and system_process are at the core of the OS. For this reason, they must rely on a strong foundation able to provide stability, security and scalability properties. Android adopted the *proxy* design pattern for this purpose. Thanks to the proxy pattern:

- Clients (applications) communicate with a centralized server, the system_process, in a controlled way;
- The system_process can expose its functionalities that applications can access in a transparent way;
- The sensible resources hosted by the system_process are protected.

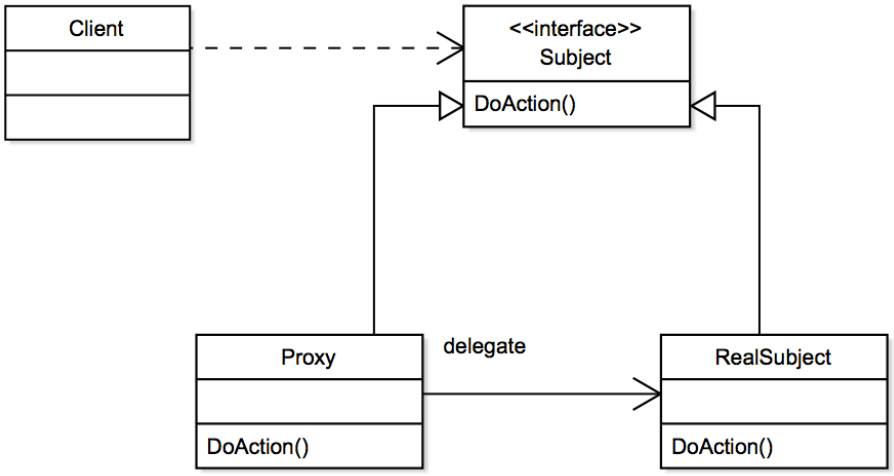

Figure 2.1. Proxy pattern class diagram. Adopted from [1].

This is achieved with the introduction of special objects, called *proxies*. The key point of this design pattern is that a proxy acts as a middleman in the communication between clients and server, providing controls and security, and clients are not aware of its presence because it implements and provides the same methods of the "hidden" object, in this case the system_process.

The presence of a proxy allows the protection of special objects that the system_process manages, called *system services*. These services control all the aspects of the device, from touch screen gestures (InputManagerService) to sensors (SystemSensorManager), from applications (ActivityManagerService) to external usb devices management (UsbService), and so on. Of course, external applications can use some of the methods that system services declare in order to exploit specific features of the device. For this reason, every system service exposes its own API, to which applications can access thanks to proxies object that implement it. In Android, proxies are usually retrieved through objects called *managers*, and they implement the API of the related service. A standard communication between an application willing to access a method belonging to the API of a system service, like the ActivityManagerService (AMS), is realised through the following steps:

1. The application obtains an *ActivityManager* instance through `Context.getSystemService(Context.ACTIVITY_SERVICE)`. The passed parameter is the constant defining the name of the AMS;
2. The application invokes one of the available method in the API of the AMS;
3. The manager forwards the request to the actual system service, that is in execution in the context of the system_process;
4. The AMS executes the method.

This protocol ensures that the system service will be accessed only through the corresponding manager, that can thus perform security checks and deny unauthorised requests. This layer of protection is not detected by an application and it

does not affect the usability of the framework. At the same time, it provides strong reliability and scalability properties since new clients just need to retrieve a new instance of the manager in order to access the functionalities of the desired service.

2.1.4 Signature and permissions

Signing an application

Every application is delivered to users in special archive files with the .apk extension. These files contain all the necessary resources of the application. In order to be publicly available for download, .apk files must be signed by developers. The signing process is achieved through the association of every application with a certificate that contains the public key matched by the private key owned by the rightful developer. If an .apk is correctly signed, its developer can be identified, and the OS can easily manage the process of applications update when new versions are available.

Permissions

One of the most crucial security mechanism in Android is represented by the permissions system. Every Android application runs in its own *sandbox*, meaning that it cannot normally access and make use of external features of the device (like sensors, network connection, etc.). However, thanks to permissions, an application can overcome this limit and increase its capabilities. For example, if an application wishes to access the system camera to take pictures, or if it needs to read the contacts from the SIM card, it must ask the user to have specific permissions granted.

It is clear that the process of granting a permission implies that the user trusts the application, leading to security implications. For this reason, not all the permissions are treated in the same way, and they are classified as:

- **Normal Permissions**: They are *automatically granted* by the system when the application is first installed. They are considered not to have a big impact on the user privacy and security [11], and so there exists no way a user can revoke one of these permissions from an application. Examples include: `VIBRATE` (allows an application to use the device vibrator), and, surprisingly, also `BLUETOOTH` (allows an application to access the device Bluetooth services) and `INTERNET` (allows an application to use the network);
- **Dangerous Permissions**: These kind of permissions must be *explicitly granted* by the user, because they are considered to have security and privacy implications. For this reason, starting from API level 23 [12], developers must follow specific implementation requirements in order to correctly prompt the user with a dialog window asking to allow or deny the grant of most dangerous permission their applications use. For lower API levels instead, the dialog is shown at application install time only. Examples include `CALENDAR` (allows an application to read and write calendar events), `CAMERA` (allows an application to start the system camera and take pictures);

- **Signature Permissions**: They provide a stronger level of security with respect to normal and dangerous level permissions. Indeed, these kind of permissions is *automatically granted* by the system, but only if the application requesting the permission has the same signature of the application that defined the permission. This means that if a signature level permission is defined by a system application (like the Settings app), only applications with its same signature can have that permission granted, thus restricting its use to system applications only;

2.2 Android applications

An Android application consists of four main kinds of components: activities, broadcast receivers, services and content providers. Each is implemented with an abstract class (called framework class) that developers must extend in order to include one or more instances of the desired component. This also means that every component has its own set of callbacks.

Moreover every Android application is published with a `AndroidManifest.xml` file, that contains the app overview and description. The manifest contains information about the application in general (package name, minimum targeted API, permissions) as well as about every activity, broadcast receiver, service and content provider that the application defines.

By default, when an app is launched, its operations are executed in its main thread, also known as *UI thread*. This thread is the only one able to add, delete and modify UI elements. For this reason, if a new thread is started by an application (either through `Thread` or `Runnable` classes), it must not perform operations on the UI. We shall refer to these created threads as *worker threads*, because their task is to perform heavy and time consuming operations without modifying the UI. This is a very important distinction that every Android developer should be aware of, because if long operations are performed directly on the UI thread for a period of time longer than 5 seconds, the result is the killing of the application with the *Application Not Responding* (ANR) error message dialog.

2.2.1 Activities and tasks

Activities represent single actions that users can perform on the device. They manage the user interface (UI) and handle the interactions through event listeners. Starting from API level 11, Android introduced a new feature that allows the split of the display into multiple sections, called *fragments*. This means that an activity can contain one or more fragments, and thus it can display more than one screen at time.
As Figure 2.2 shows, the lifecycle of an Activity is composed of 7 different callbacks, that correspond to 7 different activity states.

When users perform some operations, multiple activities are generally involved. Android defines the concept of *task* as the collection of subsequents activities that

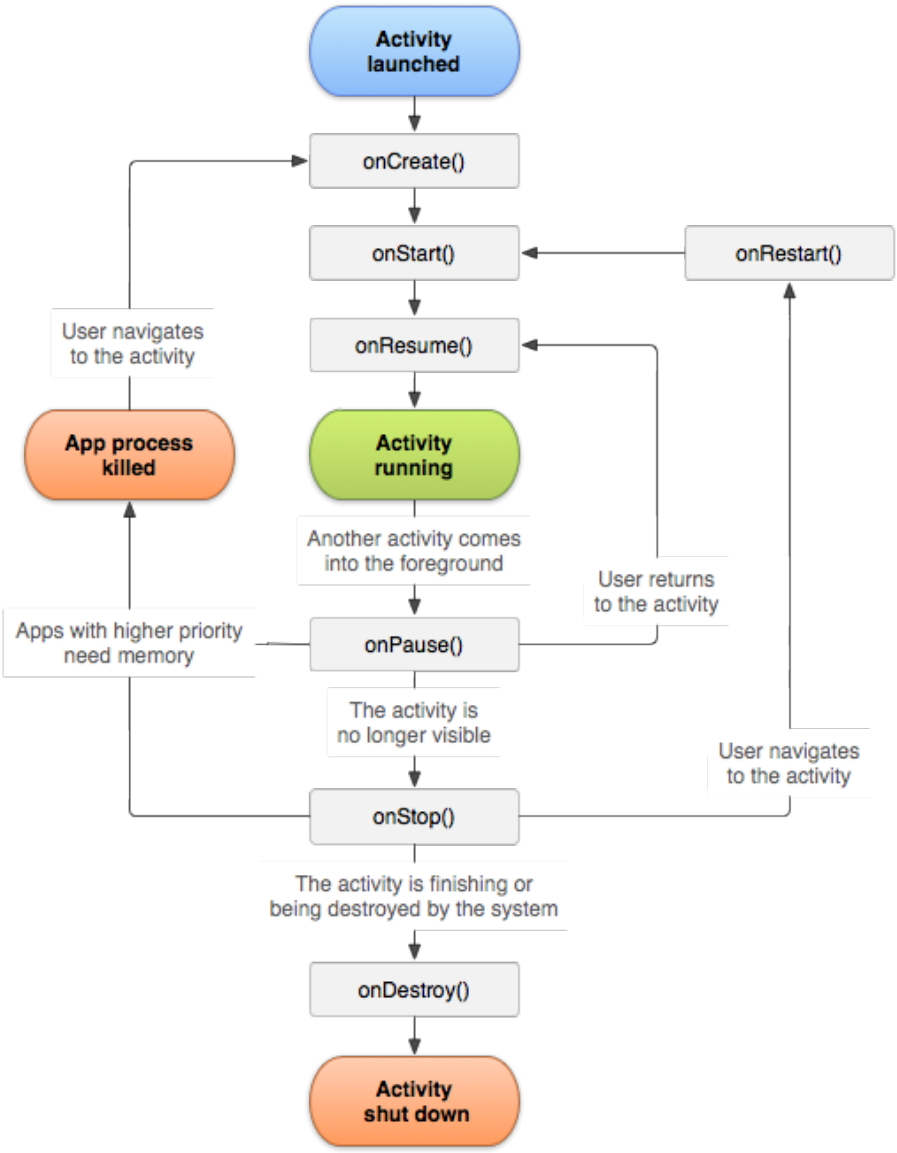

Figure 2.2. Activity lifecycle and callbacks. Adopted from [2].

users interact with. Within a task, activities are organized in a stack, denoted as *back stack*, with the most recently used activity at the top. Users can navigate through the stack by pressing the back button of the device, that pops the first activity from the top of the stack and resumes the one just below it.

2.2.2 Broadcast Receivers

A broadcast receiver is an application component aimed to listen and react to broadcast messages. These kind of messages are *Intent objects* that are sent to the whole system. Android widely relies on broadcast receivers to achieve InterProcess Communication (IPC), because they conveniently implement the *publish-subscribe* design pattern, providing low coupling between components and good scalability properties.

A receiver can *filter* the messages it wishes to receive, and can even require specific permissions for both sending and receiving broadcasts. The main callback is the method
`BroadcastReceiver.onReceive(Intent i, Context c)`, that is called by the system_process every time a broadcast message matching the receiver filter is sent.

2.2.3 Services

The next type of components are services. Unlike activities, services do not provide a UI, and are usually executed in the background for long-lasting operations. A very common example is a music player application, that keeps doing its job (playing music) even in the background.

Due to their capabilities, Services are widely exploited for several different purposes, and they represent the main target of the flaws uncovered by our work. We will analyse them in detail in the following sections.

2.2.4 Content providers

The fourth kind of component of an Android application is represented by content providers. These components are a very useful abstraction of the data stored and used by an application. Indeed, they can be accessed and queried in a similar way to SQL databases, allowing applications to process and use data published by the application hosting the Provider.

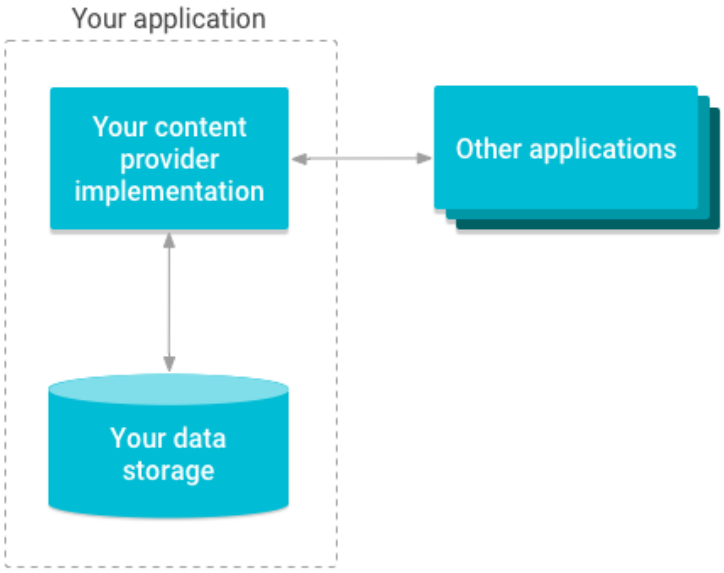

Figure 2.3. Diagram of content provider architecture. Adopted from [3].

Table 2.1. Application components description

Component Name	Role
Activity	A single action a user can perform. They can contain sub-displays, called fragments, and they are organized in tasks.
Broadcast Receiver	A listener to broadcasts sent to the whole system. They can filter the intents to be received.
Service	Used for long-term operations, with no UI. They can run in background.
Content Provider	A database-like object that applications can query to retrieve data.

2.2.5 Identify an application

The ability of identifying an application and a specific component within it is crucial in Android in order to achieve IPC.

UID and PID

When an application is launched, it is executed in its own process, identified by the Unix Process Identifier (PID). Moreover, every Android application runs under a specific user ID, or *UID*, denoting the application that owns the process in which it is being executed. Indeed, an application can specify in its manifest in which process(es) it wants to run: if more than one process is needed, different PIDs will be used, but the UID will stay the same.

For example, the Phone application, one of the system apps, runs under the pre-defined UID denoted by the constant `Process.PHONE_UID`, whose integer value is 1001. However, the PID that will be assigned to the `com.android.phone` process will vary at every new execution of the application.

Despite being precise and efficient, the use of UIDs and PIDs to identify a specific application is not so practical, and higher level approaches are preferred in most of the cases.

Package and context

Every application is denoted by a unique *package name*, a string that is usually derived from the company that published the application. Through the package name, objects of class Context can be retrieved. These objects hold a lot of relevant information about an application, including: package name, UID, location of the .apk file of the application on the device, UI thread data, etc. The context describes the environment in which an application runs, and it represents a convenient alternative to process and user identifiers in order to uniquely refer to a specific application.

2.3 Applications interaction

2.3.1 Intents

The first mechanism provided by Android to achieve IPC makes use of *intents*. Objects of this class represent the simplest way that an application has to communicate with another component, either within the same application or in external ones. In order to be correctly delivered, intents must be created so that they target the right component in the right application. To achieve this, developers can decide to use two different kinds of intents:

- **Explicit intents**: These kind of intents are created by providing the couple {package, class}. They are the most secure type of intents because they uniquely identify the target component thanks to the provided package and class. For this reason, no ambiguities can arise in the resolution of the receiver of the intent. They are generally used to target components within the same application, because of the availability of the class names;

- **Implicit intents**: Alternatively, intents can be created by providing specific values that will make the OS identify the target component through a mechanism known as intent resolution process. These values are labelled as *actions*, *categories*, *data*, *type* and *extras*. Every component of an application can declare in the manifest its own *intent filter*, that contains the values that an intent must hold in order to target that component. When an implicit intent is sent, the system_process will compare its fields to the ones of every registered intent filter. If an exact match is found, the OS will immediately deliver the intent to the matching component, otherwise the user will be prompted with a dialog to choose one among a list of applications able to handle the intent. Depending on the intent aim, its fields can be populated with pre-defined constant values (like the `Intent.ACTION_SEND` constant for intents targeting components capable of sharing some data), or with user-defined ones.

Regardless of the type of intent used, the *extras* field can always be exploited to deliver to the target any kind of data. This field is populated with a key-value mechanism, where the key is a simple string, and the value can be basically every kind of object. In this way the receiving component can extract data given the knowledge of the key. Despite being very easy to implement, the use of intents for data exchange between applications is not the most convenient way to achieve IPC, even if it is still widely used.

Intents are exploited in a wide variety of API methods, but the most commonly used are:

- `startActivity(Intent i)` and `startActivityForResult(Intent i, int id)`: Thanks to these methods, a new activity can be started. The latter method is used if the activity starting the new one is expecting a result back;

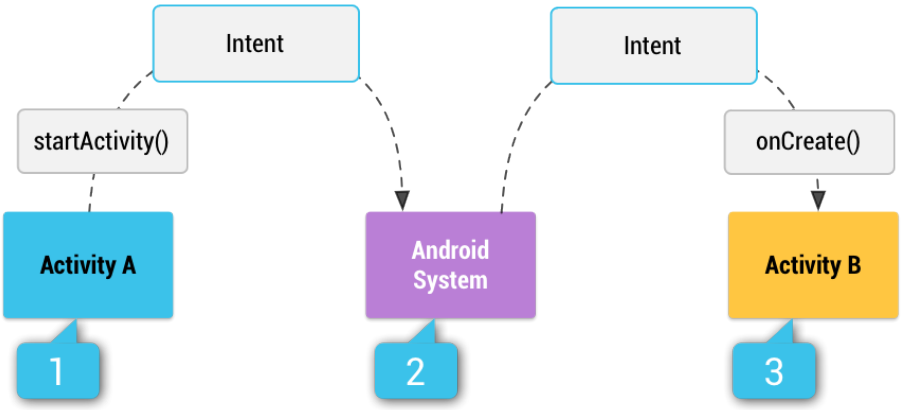

Figure 2.4. Intents can be conveniently used to start other activities. Adopted from [4].

- `sendBroadcast(Intent i)`: Through this method, the system will call the `onReceive()` callback of all the broadcast receivers whose intent filter matches the values of the sent intent;
- `startService(Intent i)` and `bindService(Intent i, ServiceConnection sc, int flags)`: These methods are used to start and establish a connection with the service component that the intent targets.

2.3.2 Handlers and messages

Objects of class handler are specific components with the only purpose of managing the exchange of *messages* between applications. An application can instantiate its own message handler by extending the class Handler and overriding the callback `handleMessage(Message msg)`, that is executed every time a message is sent through the methods `Handler.sendMessage(Message msg)` or `Handler.sendMessageDelayed(Message msg, long delayMillis)`. The messages are small objects with various fields that can be properly populated to pass various kind of information from applications to applications. Moreover, handlers organise messages in a queue system, ensuring that every sent message will be received and handled. The standard implementation of `handleMessage()` contains a `switch` statement that evaluates the `what` field of the received message, in order to execute the right method according to its value.

2.3.3 Binder

The final and most complex IPC mechanism used in Android is represented by the class *Binder* and the related interface *IBinder*. The use of binder objects allows an application to call methods of remote components as if they were declared locally. For this reason, binders are widely used to realise the proxy pattern in the communication with the system_process and the system services. Moreover, an external

application that defines a bound service can decide to expose its own custom API in order to allow clients to execute its methods. This is achieved either through the direct implementation of the methods `transact()` and the corresponding callback `onTransact()`, or through the exploitation of a specific language called AIDL (Android Interface Definition Language), that allows a more immediate implementation of the API that the service wants to expose.

2.4 Services

As Table 2.1 states, services are a kind of components without UI aimed to perform time consuming operations. Thanks to intents, services can be started from external applications, and IPC can be conveniently achieved through them. According to their purpose, there are different kind of implementation of a service.

2.4.1 Started and bound services

When an application is launched, the services that it declares are not automatically started. In order to put a service in the running state, an application can call the method
`Context.startService(Intent i)`. The parameter, an intent object, can be either implicit or explicit, but explicit intents are usually preferred to avoid starting a different service with respect to the requested one. If a service is started through this mechanism, it is a *started* service.

This is the most basic type of service, that does not allow external applications to access its functionalities and that keeps running in the background as long as the process (and thus the application) holding it is kept alive. Otherwise, a client has the capability to programmatically terminate the execution of a service in another application through the method `Context.stopService(Intent i)`, where the intent parameter targets the desired service. The call to `Context.startService(Intent i)` by an external application is matched by the callback methods `Service.onCreate()` and
`Service.onStartCommand(Intent i, int flags, int startId)`. Every call to `startService()` corresponds to a call to `onStartCommand()`, while `onCreate()` is called only after the first time `startService()` is executed.

The next type of services are *bound* services. If an external application binds to a service, it establishes a client-server relationship with it, with the server being the application where the service resides. In order to execute requests from a client, the server can expose an API through the special language called AIDL. This is achieved through the method `Context.bindService(Intent i, ServiceConnection sc, int flags)`. The intent parameter has the same meaning as in `startService()`, while the two new parameters are respectively the object where the API will be accessible and a combination of flags. One difference with respect to started services is that, starting from Android Lollipop (API level 21), the intent used to bind to a service must be *explicit*.
A bound service will be kept alive as long as there is at least one client bound to it.

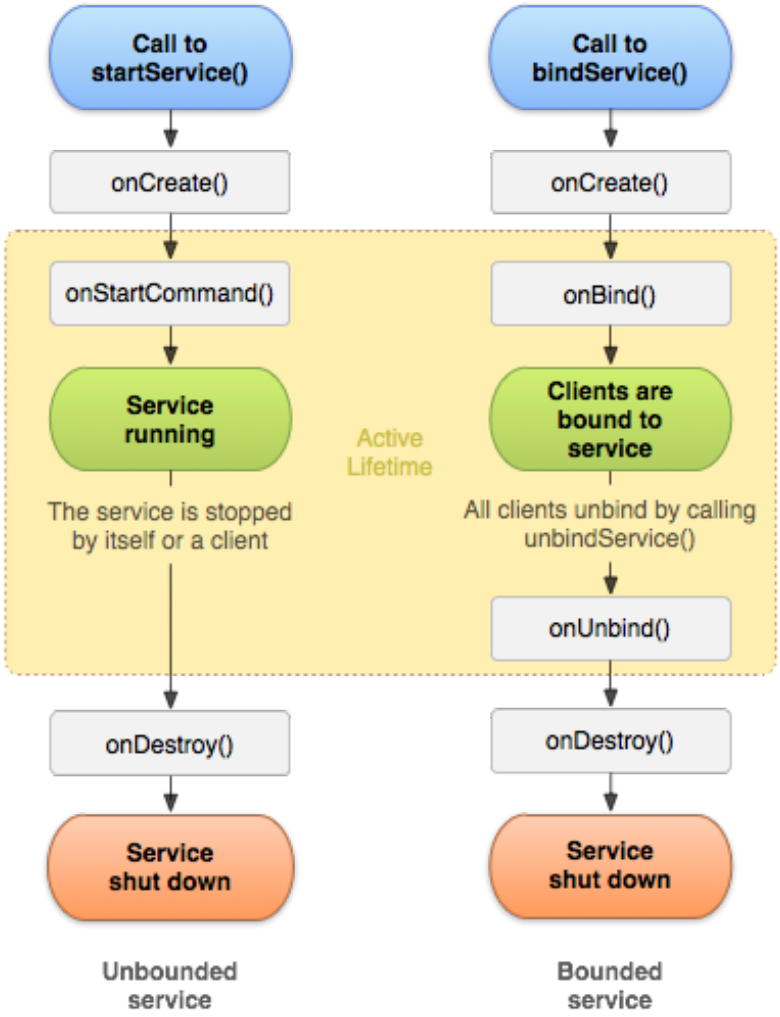

Figure 2.5. Started and bound services lifecycles and callbacks. Adopted from [5].

The call to `Context.bindService(Intent i, ServiceConnection sc, int k)` is matched by the callback method `Service.onBind(Intent i)`, executed every time a client calls `bindService()`.

2.4.2 IntentServices

One particular type of service is represented by the IntentService class. The previous considerations about the UI and worker threads are particularly important when services are involved. Indeed, services are designed to perform long lasting operations, but by default they are executed in the UI thread of the application hosting them. This means that in the majority of cases, services need to create a new worker thread as soon as they are started, to avoid blocking the UI thread and receiving the ANR error message.

The IntentService class solves this problem, by directly providing a service with a pre-defined worker thread, so that every operation will be safely executed without interactions with the UI. In order to use this type of service, developers must extend

the IntentService class and override the specific callback method `onHandleIntent(Intent i)`. This callback works just as `onStartCommand()`, being executed at every call to `startService()` or `startForegroundService()`. However, one major difference with standard services is that intent services do not provide any type of concurrency mechanisms, and every call to
`onHandleIntent(Intent i)` is thus executed sequentially.

2.4.3 Foreground and Background Services

A service can be declared to run in the foreground if it is executing some operations known to the user. For example, a service playing a song is a foreground service. This kind of services are required to display a *notification* in the the status bar of the device, to inform the user of the action they are performing. On the contrary, a service is running in the background if the user is not aware of its operations. For example, a service that is updating a resource that will be used by the application later on, possibly downloading it from the network, will be a background service. This classification becomes relevant when the system_process is in a low memory situation and it needs more CPU resources to perform operations. In this case, background services will be the first candidates to be killed by the system.

Started and bound conditions are not mutual exclusive: a service can be first started and then bound by a client. Vice versa, if a client first binds to a service, it will also start it if it was not already running. Moreover, a started Service can either be running in the foreground or in the background, with bound services immediately brought to the foreground as soon as a client executes one of their API call. Table 2.2 summarises these concepts.

Table 2.2. Service types

Service Type	Description
Started	Default service type. No API exposed. Runs as long as its process is kept alive.
Bound	Client-Server model for IPC. Runs as long as at least one client is bound.
IntentService	Standard service with a predefined worker thread. No concurrency.
(Started) Foreground	For actions noticeable by the user. Must provide a notification in the status bar.
(Started) Background	For actions not noticeable by the user. Can be killed by the system if it needs CPU resources.

2.4.4 Changes in Android Oreo

Android Oreo is the last stable version of Android, released in Semptember 2017 [13]. Like every new release, it brought some new features, and some of them are directly related to services [14]. The most interesting element of novelty is the method `Context.startForegroundService(Intent i)`. It works just as the previously mentioned `Context.startService(Intent i)` method, but it has the ability to demand that the started component will be a foreground service, independently from the state of the application hosting the service. This means that the user has now a way of telling the system that the service she wants to start is relevant to her, and it should not be killed if the system is low on resources.

Chapter 3

Evolution of security mechanisms in Android

Over the past years, the growth of the capabilities of the Android operating system has always been followed by a constant evolution of the threats to which mobile devices are exposed. For this reason, every new Android release is aimed at the complete or partial solution of the most serious security issues that previous versions suffered from. We now introduce some of the most famous and relevant attack vectors of the last years, and we will analyse which techniques have been adopted by the Android developing team in order to mitigate their effects.

3.1 Android threats

Malwares in Android are usually deployed and spread in form of malicious applications. Users have two main ways of downloading and installing a new application on their devices:

- From the official application store, called Google Play Store;
- Directly downloading the .apk file of the desired application from external sources like websites or even other applications.

Of course, the direct download of the .apk file from external sources is highly discouraged, as there is no verification on the authors and publishers of the application. However, several malicious apps have been found in the Google Play Store, disguising themselves as benevolent even for long periods of time [15].

Whatever the source is, a malicious application can execute a wide range of attacks exploiting specific features of Android in unconventional ways: phishing, man in the middle (MITM) attacks, privilege escalation attacks, denial of service (DoS), adware, ransomware, etc.

3.1.1 Intent hijacking

We already highlighted how intents represent a very convenient way to achieve IPC. However, implicit intents have been one of the most exploited Android feature since their introduction, through a kind of attack known as *intent hijacking*. The underneath working principle is relatively simple, and it is based on two main factors:

1. The possibility for a malicious application to declare the same intent filter of another application component at its own will;

2. The absence of controls on the received intents.

The result is that a component of a malicious application can be activated in place of another. Since intents can carry personal data, the security implications have often been very serious. Moreover, since intents are used to communicate with various application components, intent hijacking attacks can be divided into activities, broadcast receivers and services hijacking attacks.

Activity hijacking

When the intent filter of a benign activity is copied by a malicious application, the starting of that activity through the API method `startActivity(Intent i)` or `startActivityForResult(Intent i, int requestCode)` can result in the undesired invocation of the malicious activity. This is usually achieved by setting the attribute *android:priority* of the intent filter of the malicious activity to a high value, so that the malicious application will be opened instead of the benign one. Once an application can intercept implicit intents in this way, it usually clones the UI of the benign application, so that the user is not aware of the hijacking. Moreover, since activities started in this way can also extract data that the intents carry in their *extras* field, the malicious activity can manipulate such data in every possible way (like sending personal information to a malicious server to which the malicious application is silently connected). Through activity hijacking, attacks like phishing, MITM and DoS can easily be realised.

One immediate way that users have to detect the hijacking of an activity is to look at the list of recent open applications, because the malicious activity will still be part of a different application with respect to the one the user is expecting. However, an application can decide to exclude itself from the list of recent apps through the attribute *android:excludeFromRecents* set to `true` in the Manifest. This technique was widely adopted by several application performing this kind of attacks [16].

Apart from avoiding the use of implicit intents, it is responsibility of the developers to avoid using them to send sensible information and to perform the right controls on the received intents (in case of MITM attacks, where a malicious activity can impersonate another activity and send corrupted intents on its behalf).

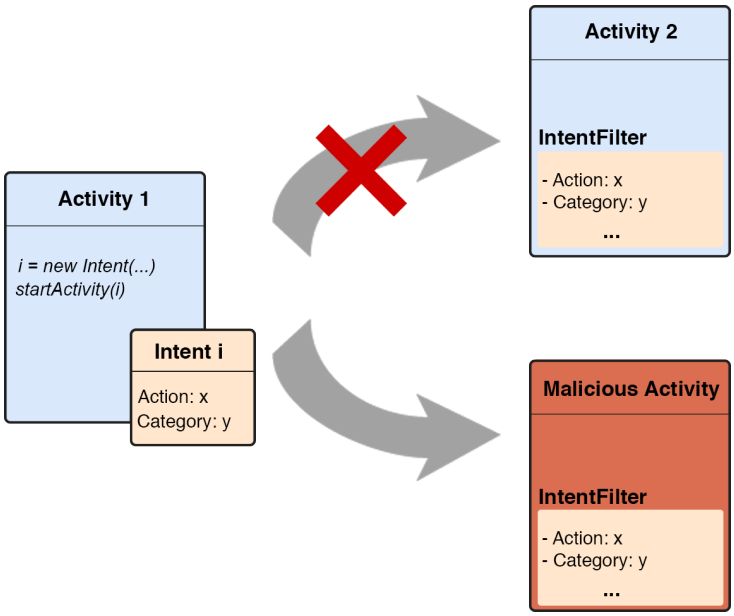

Figure 3.1. Activity hijacking attack.

Broadcast receiver hijacking

The hijacking of broadcast receivers can be even easier to realise. Indeed, a malicious application only needs to register its broadcast receiver, either statically in the Manifest or dynamically through the API method `registerReceiver(BroadcastReceiver receiver, IntentFilter filter)`, so that its intent filter matches the type of broadcast intents that will be sent. In this way, every time a broadcast intent is sent, the `onReceive(Context c, Intent i)` callback of the malicious broadcast receiver will be executed, performing every kind of operation that can harm the device.

One way developers have to protect against the hijacking of broadcast receivers is to restrict the use of broadcast messages only to components within their application, by defining the broadcast receiver as *local*. Otherwise, they can decide to use a feature of Android called *ordered* broadcasts. This kind of messages is sent with a number that represents the maximum number of broadcast receivers that will process the intent. The order in which a receivers will process this kind of broadcasts is determined by the *android:priority* attribute of the intent filter of each registered receiver.

Services hijacking

The hijacking of services is probably the most dangerous type of intent hijacking attacks. The reason is that, except for those services that modify the UI or the user experience in general (like playing a song), users cannot know which service they have started through an implicit intent. This means that if a malicious service is started instead of a benign one, it can perform its operations indefinitely in the background. Additionally, if the malicious application is granted specific permissions

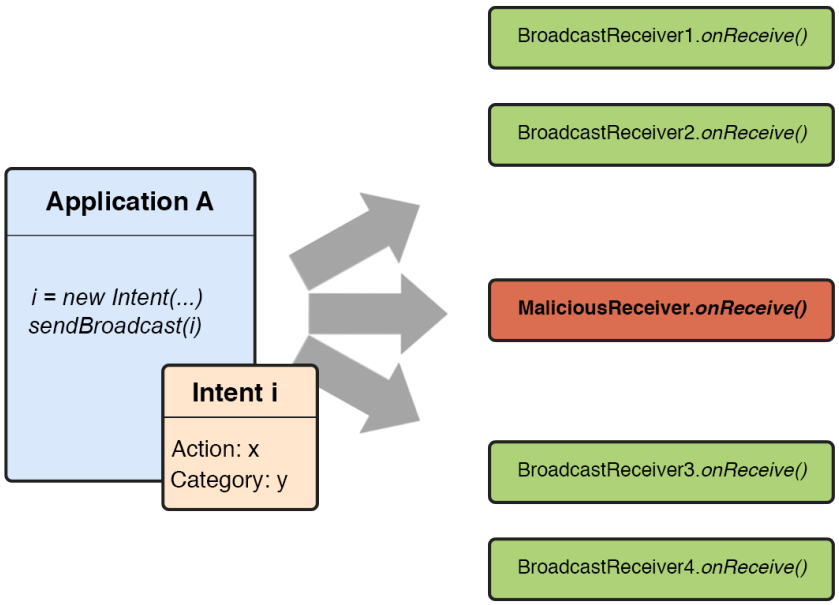

Figure 3.2. Broadcast receiver hijacking attack.

(like the normal level `INTERNET` permission), the service is left free to execute its task in an uncontrolled way, and it can become difficult to detect and stop.

3.1.2 User interface attacks

This kind of attack was usually exploited as the second step of an activity hijacking attack. As we already mentioned, once a malicious activity is loaded instead of a rightful one, it usually clones the UI of the benign application in order not to be detected by the user. However, this process implies the ability to know which application and which activities are being hijacked, in order to adapt the UI design accordingly. Since this ability has been restricted to only system applications starting from Android Lollipop (API level 21), UI hijacking attacks are usually developed to target just one application. In other terms, a malicious application tries to copy every aspect of a legitimate app, starting from the icon and the name (that cannot be exactly the same), hoping to be downloaded instead of the original one. This kind of attack can harm the most naive and forgetful users, and is usually triggered through social engineering means, that lead to the installation of undesired applications.

Cloak and Dagger

While UI hijacking is a relatively old and simple category of attacks, nowadays being replaced by more sophisticated ones, a totally different mention has to be

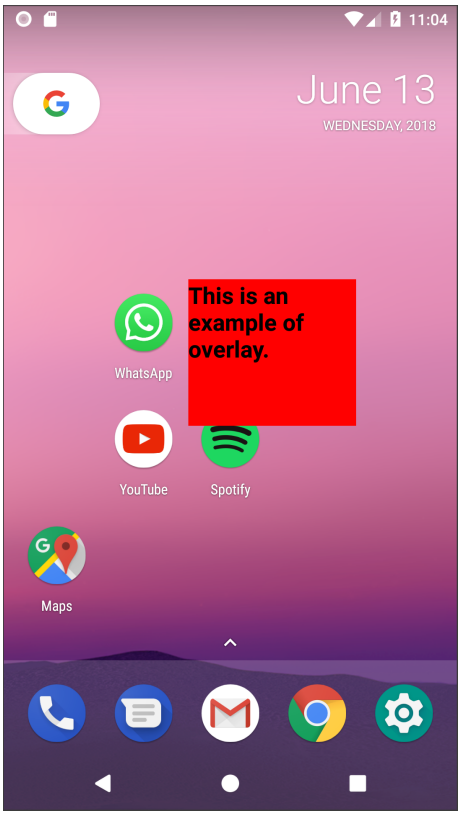

Figure 3.3. Example of overlay UI element. The TextView stays on top of everything else on the screen even when the app is closed.

done for a set of attacks called *Cloak and Dagger*, deployed in May 2017 [17]. This new attack vector is based on a vulnerability concerning applications downloaded from the Play Store and requesting the special permission `SYSTEM_ALERT_WINDOW`. If an application is granted this `SIGNATURE` level permission, it has the ability of drawing permanent UI elements on top of every other applications, that are displayed even if the application defining them is closed. The surprising element is that this permission, being requested by a lot of famous applications, is *automatically granted* if the application is downloaded from the Google Play Store.

Cloak and Dagger researchers found that the overlay elements could be designed to occupy the full screen of the device, and they could even choose to intercept or ignore touch events. In this way, users could be tricked into clicking specific portions of the screen covered by an overlay, while they were actually performing the chain of touches needed, for example, to grant all the permissions to the malicious application. Despite being based on a relatively easy mechanism, this kind of attack gave a whole new relevance to UI focused attacks, because of the huge variety of dangerous behaviours that could be achieved by convincing the users to perform just few clicks.

3.1.3 Callback mechanism failures

Another category of attacks involves the system_process and its interactions with third party applications. We highlighted how system services, managed by the system_process, expose their own APIs so that external applications can access their methods. However, this design must be carefully implemented, because it makes the system_process work with inputs arbitrarily chosen by an external application.

Specifically, in 2016 researchers found that some of the methods belonging to several system services, like the *ActivityManagerService*, the *LocationManagerService*, and the *PackageManagerService*, were poorly implemented, because they accepted as parameters objects on which the system_process would call specific methods [18]. This mechanism has the same working principle of the callback mechanism, except that the callbacks method are executed on objects instantiated by external applications.

This means that by properly designing the parameters, an attacker could decide to execute arbitrary code in the context of the system_process. Moreover, some of these vulnerable methods involved `synchronized` statements and blocks, so that the system_process could be put in a frozen state by denying the access to specific resources to other processes. Another point of attack made use of arbitrary code that explicitly raised exceptions that could not be handled by the `try-catch` blocks of the system_process.

All these failures of the main Android process could lead to the soft reboot of the device, resulting in DoS attacks that made devices impossible to use.

3.1.4 Bluetooth based attacks

The Android OS, being built on a Linux kernel, was not excluded from a list of several systems found to be vulnerable to a wide variety of attacks involving the Bluetooth protocol stack. In 2017, the *BlueBorne* attack vector was deployed by *Armis Labs*, that demonstrated how the Bluetooth protocol implementation was full of vulnerabilities like stack and heap overflows and integer underflows [19].

These vulnerabilities were present basically at every level of the protocol stack, resulting in major security issues. The main reason is the complex structure of the protocol, that is composed of a lot of different basic blocks in every layer, resulting in a very high fragmentation level. The researches demonstrated how an attacker could gain control of a Bluetooth connected device just by being physically close to it for a small amount of time (about 10 seconds). Moreover, the compromised device could also represent the starting node for the diffusion of malwares of any kind.

Given the amount of devices subject to these vulnerabilities, that included smartphones, laptops, IoT devices (like vocal assistants, smart houses etc.) and many more, this Bluetooth based attack drew the attention of many famous companies, like Google, Apple, Microsoft, Amazon.

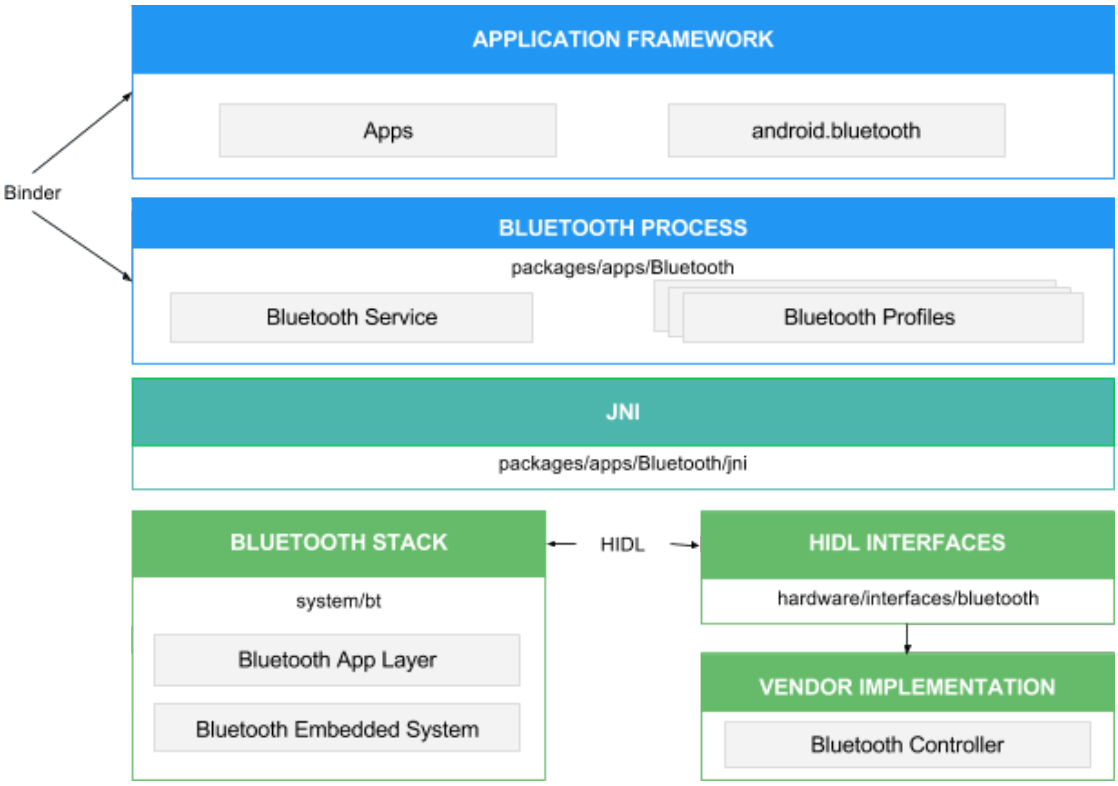

Figure 3.4. Bluetooth protocol stack in Android. Adopted from [6].

3.2 Android security mechanisms

The described attacks represent a high-level overview of the main categories of vulnerabilities that Android has been subject to. Some of them are still exploitable, especially in devices running older versions of the OS, while others have been completely fixed. For this reason, the first way users have to defend against these threats is to keep their devices constantly updated to the latest Android release.

However, Google has adopted a wide set of techniques aimed not only to mitigate the specific issues highlighted by some of these attacks, but also to solve the underneath flaws that affect its operating system.

3.2.1 Google Bouncer

With the explosion of the number of malicious applications found on the Play Store, one of the first security mechanisms adopted by Google was the *Google Bouncer*, a new layer of security "*[...] which provides automated scanning of Android Market for potentially malicious software without disrupting the user experience [...]*" [20]. Bouncer was introduced in February 2012, with the aim of drastically reduce the number of malicious applications that silently resided on the Play Store. It uses a combination of static analysis techniques, emulated software execution and data collection techniques in order to identify potentially malicious software and exclude

it from the store.

Theoretically, the use of a centralized control system in the context of the Play Store is a good approach. However, Google Bouncer was *fingerprinted* quite rapidly from researchers [21], so that its working details were made public. Among them, we now know that the Bouncer:

- Runs the applications it tests in an emulated device based on the QEMU emulator;
- Analyses applications for 5 minutes, and if it does not detect any malicious behaviour in this amount of time, the application is labelled as benign.

Of course, the disclosure of such details about the main security mechanism of the Play Store is very dangerous, because attackers can develop their software in such a way that every security control can be bypassed, and still be able to publish their application on the store.

3.2.2 Implicit intents deprecation

Given the amount of hijacking techniques that could be realised through the exploitation of implicit intents, the use of this kind of objects is being discouraged more and more in every new Android release. For example, starting from Android Lollipop (API level 21), the execution of the method `Context.bindService(Intent i, ServiceConnection sc, int flags)` with an implicit intent as parameter is forbidden [22].

Moreover, developers should pay attention to the attributes *android:exported* and *android:permission* that can be set in the Manifest of their applications, in order to restrict in the most precise way the access to activities, broadcast receivers and services from external applications.

3.2.3 API methods restrictions

Another line of protection adopted by Android is represented by the reduction and limitations of the results returned by specific API methods. For example, we noticed how intent hijacking attacks strongly relied on the ability of dynamically understanding which processes were in execution on the attacked devices, in order to adapt their malicious behaviours accordingly. For these reasons, several methods belonging to system services APIs, ActivityManagerService in particular, have been radically changed in order to return results that do not have security implications anymore. Examples include:

- `getRecentTasks (int maxNum, int flags)`: this method used to return the list of recent tasks that were running in the device. Starting from Android Lollipop (API level 21), this method was deprecated and it now returns only the recent tasks relative to the calling application, and not to the whole system [23];

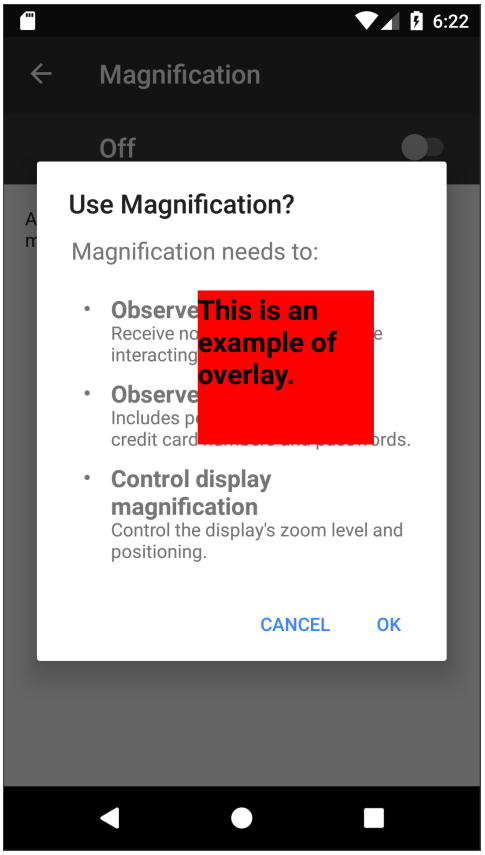

Figure 3.5. In applications targeting API level lower than 26, overlays were allowed to be displayed even on top of dialogs windows.

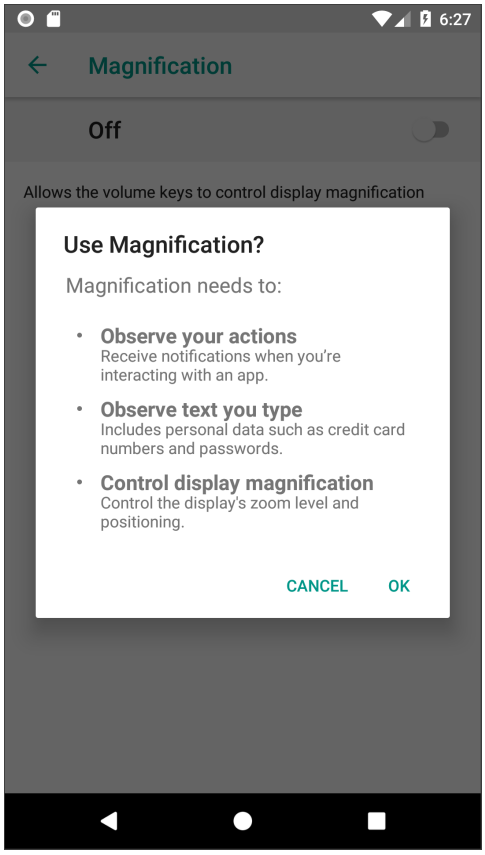

Figure 3.6. For applications targeting API level 26 and higher, the behaviour has been fixed. Every dialog window now disables any overlay element.

- `getRunningAppProcess()`: this method used to return the list of all the running processes in a device, directly reading it from the `proc/` folder. However, due to the security consequences it implied, it now requires the `SIGNATURE` level permission `INTERACT_ACROSS_USERS_FULL`, that is only granted to system applications [24];

- `getRunningServices(int maxNum)`: this method used to return the list of all the services in execution, whose size is limited by the `maxNum` parameter. Starting from Android Oreo (API level 26), this method was deprecated, and it will only return the list of running services declared by the calling application [25].

3.2.4 Overlays limitations

The specific vulnerabilities exposed by Cloak and Dagger attacks made Android drastically change the way in which overlays can be used by third party applications.

First of all, prior to Android Oreo, developers had the possibility to create their overlays elements with the flags `TYPE_SYSTEM_OVERLAY` or `TYPE_SYSTEM_ALERT`. These flags allowed the UI widget to be displayed on top of every other element, included status bar, popups and keyboard. These flags are now deprecated, and the only flag available to developers is called `TYPE_APPLICATION_OVERLAY` [26], that allows the system to manage overlay elements in a more controlled way. For example, the status bar and the dialog windows prompted to the user when he/she has to make a choice (like granting a permission) are now protected against overlays, sharply reducing the feasibility of some attacks introduced by Cloak and Dagger.

Moreover, the system_process can decide to resize or move the overlay element, if it covers relevant portions of the screen.

However, despite their potential, Google did not fully recognise the threats originating from these kind of attacks, so that the `SYSTEM_ALERT_WINDOW` permission is still automatically granted to applications requesting it if they are downloaded from the Play Store. This means that overlays could still be used for other malicious behaviours. For example, our research found that it is still possible to draw overlays on top of the *Camera* application, and a user could be tricked into taking pictures both from the primary and the secondary cameras in undesired contexts.

Figure 3.7. Overlays can still be displayed on top of many applications, like the Camera application of the emulated device. The overlay is designed to be fullscreen with a transparent background in order to show what it hides behind it.

Chapter 4

Vulnerability Description

4.1 A new design choice

The new method introduced in Android Oreo implies a change of responsibilities for both the actors involved in the scenario: the application requesting the start of a service (application A) and the application hosting the service itself (application B). First of all, application A has to understand if the service it is requesting must be started through `startService()` or `startForegroundService()`. However, application A does not have any mechanism to draw such a conclusion.
On the other hand, as the official SDK documentation states, the started service in application B must call the second new method introduced in Oreo, called `startForeground(int i, Notification n)`, where the parameters represent an identifier for the notification that must be displayed by every foreground service, and the notification object itself. This new method must be called in the `onStartCommand()` callback and should be used by those applications for which the killing of a service would be "disruptive to the user" [27], compromising the whole purpose of the application itself. Figure 4.1 and Figure 4.2 show a legal use case of these new methods and the corresponding callbacks.

From the above considerations, we can identify four different use cases according to what both the actors (application A and application B) decide to execute. Table 4.1 explains these combinations.

Table 4.1. Foreground services - Possible use cases

App A	App B	Behaviour
`startService()`	`onStartCommand()`	Normal: background service
	`onStartCommand()` + `startForeground()`	Normal: foreground service
`startForegroundService()`	`onStartCommand()`	**Undefined behaviour**
	`onStartCommand()` + `startForeground()`	Normal: foreground service

```
public class MainActivity extends Activity {

    ...
    Context c;
    Intent i;
    ...

    @Override
    protected void onCreate(Bundle savedInstanceState){

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ...
        try {
            c = createPackageContext("app_B_package_name",
                CONTEXT_IGNORE_SECURITY);
        } catch(PackageManager.NameNotFoundException e){
            e.printStackTrace();
        }

        i = new Intent(c, ForegroundService.class);
        startForegroundService(i);
        ...

    }

    ...

}
```

Figure 4.1. Application A code showing the correct way how to start a foreground service.

We notice that if application A decides to start the service in application B as a foreground service, but application B is not prepared for this, and thus it does not call `startForeground()` in its `onStartCommand()` callback, the behaviour is left undefined. By performing some tests both on the emulated and the real device, the observed behaviour is actually well defined, but it is definitely not the intended one: application B crashes programmatically, with the undesirable "*Application Not Responding*" (ANR) dialog error message. The dialog appears after a timeout set to 5 seconds. The error message printed on the system_process log reads as follows: "*Context.startForegroundService did not then call Service.startForeground*".
This means that application A can make application B crash at its own will, and application B does not have any error recovering or backup mechanism to prevent that.

4.2 Technical details

Starting a service, either in the background or in the foreground, always involves the system_process. Indeed, `Context.startService(Intent i)` and

```
package app_B_package_name;
...
public class ForegroundService extends Service {

    ...
  private int id = 1;
  private Notification n;
  private Notification.Builder mBuilder;
  ...

  @Override
  public int onStartCommand(Intent i, int flags, int startId){

    ...
    n = mBuilder.build();
    startForeground(id, n);
    ...

    return START_NOT_STICKY;

  }

  ...

}
```

Figure 4.2. Application B code showing the correct way how to bring a service in the foreground.

`Context.startForegroundService(Intent i)` are only wrapper methods that invoke the function
`startService(..., Intent service, ..., boolean requireForeground, ...)`, that is part of the API exposed by the ActivityManagerService (AMS), one of the most important system services hosted by the system_process.

We can notice that one of the parameters of the API method is a boolean variable, whose value depends on how an application requested the start of the service: It is `true` if the service is being started through `startForegroundService()`, `false` otherwise. Once the client application posts a request to the system_process, the AMS has the responsibility to perform the following actions:

1. **Perform security checks**. This task is performed thanks to the class *Binder*, that allows the AMS to retrieve the UID and PID of the application requesting the service. Since the system_process holds a record of all the processes being executed by a given user and since it is in charge of checking what permissions are granted to every user, it can block any unauthorised request and prevent security issues.

2. **Identify the hosting application**. The ability to know what process is currently in execution on a device is specific to the system_process, starting from Android Lollipop (API level 21). This means that the system_process

can identify what application is targeted by the intent used to start the service, and it can start it if it is not already running.

3. **Start the service**. Once the application that defines the service to be started is identified, the service lifecycle can start. To execute the callbacks, the AMS uses the helper class *ActiveServices*. If it is the first time that the service is started, the first callback to be executed is `onCreate()`, otherwise it can immediately proceed to call `onStartCommand()`. The procedure is the same in both situations. It involves both the class ActiveServices and the AMS, that asynchronously schedule the creation and/or starting of the service by sending an object of class *Message* to the *Handler* of the hosting application, and starting the 5 seconds timeout. The sent messages can contain the constant values `CREATE_SERVICE` or `SERVICE_ARGS`. The handler defined in main execution thread of the target application is implemented to handle these messages with the proper methods: `handleCreateService()` and `handleServiceArgs()`. These methods contain the call to respectively `onCreate()` and `onStartCommand()`. Once these methods are executed, the application communicates back to the AMS the performed action. Finally, if the boolean `requireForeground` was `true` and `startForeground()` is executed in this flow, the timeout is immediately cancelled and the application can run safely. Otherwise, the timeout expires and the ANR dialog appears.

Chapter 5

Automatic Detection and Evaluation Tool

5.1 Aim of the tool

Once the vulnerable pattern has been recognised, it is important to understand if a given application is subject to its effects or not. For this reason, the first aim of the implemented tool is to classify an application as vulnerable or safe according to the presence of such a pattern. Moreover, we built the tool with the purpose of evaluating the obtained results, to compare expected and actual behaviour of every tested application.

5.2 Reasons to use the tool

The first advantage of the tool is scalability. Indeed, since the initial input is the .apk file of the application of interest, it is relatively easy to collect a large number of inputs and to analyse them in a fast and efficient way. In order to hide their internal mechanisms, most of the companies publishes their applications only after a process known as *code obfuscation*. Some common ways to achieve this result are:

- Renaming of methods and variables with meaningless strings (usually one or two-letters words);
- Adding redundant code not relevant to the logic of the application;
- Adding random comments.

The use of code obfuscation techniques implies that the reverse engineering phase performed by the tool does not lead to the original source code. Instead, the obfuscated source is retrieved. However, obfuscation techniques do not affect API calls (that are thus recognizable), while every custom defined method, variable and class name is modified. This means that manual inspection of the code can become a really hard and time consuming task, especially when multiple classes are involved as in the services hierarchies. In this context the use of a testing tool like ours can be very convenient.

```
public class AddGroupParticipantsSelector extends aau {

    private final Set<String> y = new HashSet();
    private final sg z = sg.a();

    public final void a(e eVar, fo foVar) {
        super.a(eVar, foVar);
        if (this.y.contains(foVar.s) || this.v.a(foVar.s)) {
            int i;
            eVar.a.setClickable(true);
            TextEmojiLabel textEmojiLabel = eVar.d;
            if (this.y.contains(foVar.s)) {
                i = AnonymousClass1.cP;
            } else {
                i = AnonymousClass1.Ez;
            }
            textEmojiLabel.setText(i);
            eVar.b.setEnabled(false);
            eVar.d.setTypeface(null, 2);
            eVar.c.a(b.c(this, f.bJ));
            return;
        }
        eVar.c.a(b.c(this, f.bM));
    }

    public final void a(fo foVar) {
        if (!this.y.contains(foVar.s)) {
            super.a(foVar);
        }
    }

}
```

Figure 5.1. Example of obfuscated code extracted from application Telegram.

5.3 Logical description

Given the nature of the vulnerability described in Chapter 4, our tool can be conveniently labelled as a static analysis tool, completed by a final evaluation phase. Indeed, its task is to look for a chain of executed methods directly in the extracted source code, through a *source-sink* mechanism, without the actual execution of the code.

A source-sink mechanism is the analysis of the execution flow that starts at specific locations in the code (sources) where the user has the ability to control some inputs. The sinks are the end points of the flow, where a potentially corrupted input can lead to unexpected behaviours. In this specific case, the source method is one between the
`Service.onStartCommand(Intent i, int flags, int startId)` and
`IntentService.onHandleIntent(Intent i)` callbacks, while the sink method is
`Service.startForeground(int id, Notification n)`.

The initial input of the tool is a publicly available .apk file. These kind of files

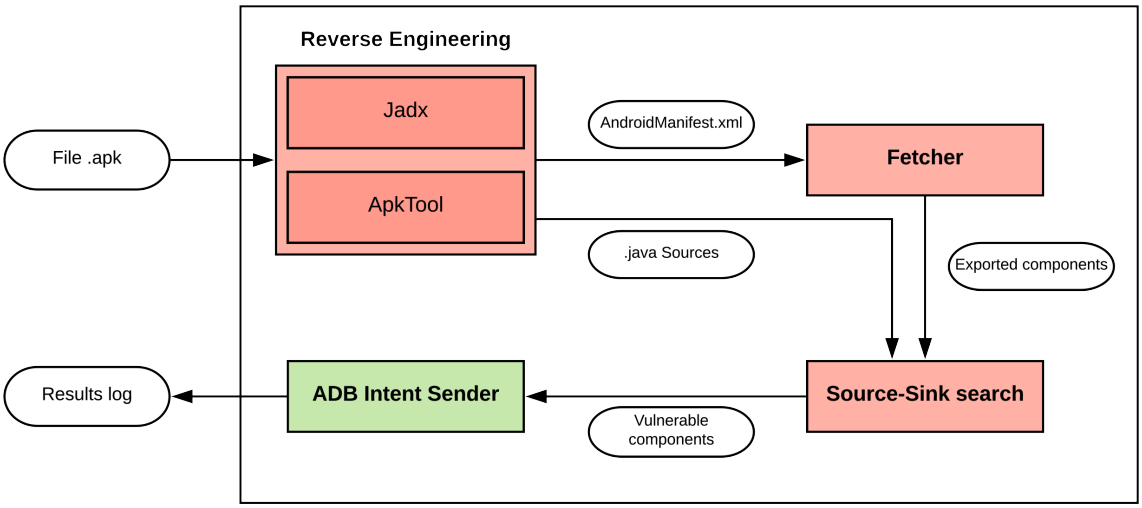

Figure 5.2. Tool architecture overview.

are archives containing all the compiled classes (in Android machine instructions, called *Dex bytecode*), resources and libraries of an application, and they represent how applications are delivered and installed on devices. The final output is instead a log file reporting the obtained results.

In order to achieve the desired results, the tool consists of four different components, each with a specific task. Figure 5.2 shows the internal architecture of the tool, highlighting inputs and outputs of each component.

5.4 Limitations

The main limitation of the tool we implemented is the absence of coverage for possible *callback paths*.

The execution of the sink method `startForeground()` can only be found in one of the classes that belong to the hierarchy of the service under investigation, because this function is specific to the *Service* framework class, and only subclasses have access to it. However, the call to `startForeground()` can be found in classes external to the service hierarchy if callback paths are used. This kind of execution flow is achieved if one class of the service hierarchy passes its own reference, the Java `this` object, to a method belonging to a class not in the service framework. In this case, the passed parameter is a valid object of a subclass of *Service*, and it supports the execution of `startForeground()`. The implementation of the exploration of this kind of paths represents one future expansion of the tool.

5.5 Developer Manual

We shall now analyse the working mechanism of every module of the tool, describing the used algorithms and data structures. Differently from many standard static

analysis tools, we also decided to include a module that allows the immediate evaluation of the obtained results as the final block of our tool. For this reason, the first three modules are aimed to the actual analysis of the source code and the detection of candidate vulnerable services, while the last block directly tests the results on a connected device (either real or emulated).

5.5.1 Reversing the apk

The first task of the tool must be a reverse engineering action able to reconstruct the source code from the bytecode found in the given .apk file in the most reliable way. Various tools are already available for this purpose, but we decided to adopt two of them:

1. **ApkTool** [28]: A famous tool for reverse engineering apk files and extracting their contents, especially the `AndroidManifest.xml` file;
2. **Jadx** [29]: A tool able to convert apk/jar files directly to Java source files very close to the original versions.

Specifically, we mainly used ApkTool to extract the `AndroidManifest.xml` file, while we relied on Jadx to retrieve the (obfuscated) source code of most of the analysed applications.

In order to retrieve a significant amount of .apk files we leveraged one famous tool available online, known as APKMirror [30]. The main advantage is that APKMirror allows the download of apk files that are compatible with several different architectures. As a matter of fact, Android runs mostly on devices supporting various *ARM* instructions sets, and so most of the online available apks are built only for such architectures. However our research environment, and thus the Android emulated device, runs on a *x86* architecture, meaning that we needed a specific version of apk files compatible with our setup. Thanks to APKMirror, we retrieved the entire set of apks of the tested applications.

5.5.2 Manifest fetcher

This module scans the `AndroidManifest.xml` file previously extracted in order to obtain a list of potentially vulnerable components of the input application. In our scenario, the interesting components are those services that are declared as *exported*. Indeed, among the several properties that can be specified in the manifest, there is the possibility to restrict the use of a component only to the application defining it. This is achieved through the tag `android:exported` set to `false`. If components, and services in particular, are declared as not exported, no other application but the declaring one is allowed to execute `Context.startService(Intent i)` or `Context.startForegroundService(Intent i)`, thus restricting the use of the service to just one application.

The default value for the `android:exported` attribute is `false`, but it is automatically set to `true` if a service declares an intent filter even if it does not specify

the `exported` value. This is a common mistake found in many applications; services intended to be private are exported instead.

Another way an application has to limit the use of one of its components from external applications is through the Android permissions mechanism. By using the attribute `android:permission`, the defining application asks all the clients that try to start one of its components to have that particular permission granted. Since the granting of a permission involves the user approval, this is another way to protect an application component from being accessed from external parties.

```
<service android:label="@string/service_label_main"
         android:name="com.spotify.mobile.android.service.SpotifyService"
         android:exported="false">
    <intent-filter>
        <action android:name="com.spotify.mobile.service.action.COSMOS_PROXY"/>
    </intent-filter>
</service>
```

Figure 5.3. Service declaration in AndroidManifest.xml file of application Spotify.

For all these reasons, this module of the tool looks for those services meeting either conditions:

1. attribute `android:exported` set to `true`, or;
2. attribute `android:exported` not defined, but an intent filter is declared.

Among these exported services, the tool also considers the ones that require permissions to be started. This decision arises from two different reasons:

1. There are a lot of Android applications requesting a higher number of permissions with respect to those actually needed for the application purpose [31];
2. The users are not normally aware of the effects that each permission implies. As a result, they tend to grant whatever permission the application asks them.

This means that even if user approval is necessary to start/bind services requiring permissions, this approval is very likely to be given, and such services should thus be included in the list of potentially vulnerable components.

The manifest fetcher we implemented exploits a *SAX* (Simple API for XML) algorithm. This approach was preferred with respect to the classical *DOM* (Document Object Model) mechanism for both efficiency and efficacy reasons. Indeed, the DOM approach builds a tree-like representation of the input XML file, with a node for every attribute or tag found. This means that it provides a global overview of the whole document, as well as the capabilities to efficiently traversing the tree. For these reasons, a DOM approach is usually adopted when dependencies between different sections of the XML input file are present (like the modification of attributes

according to previously found tags), or when operations like sorting, rearranging or searching are involved.

In our scenario, the overhead represented by the creation of the document tree of the AndroidManifest.xml file was not needed, because we were able to fetch all the desired services by sequentially scanning the manifest file. Moreover, the SAX algorithm is *event-driven*, and it provides useful callbacks that are invoked as reaction to specific elements found in the input file. For example, the detection of the opening of XML tags is matched by the invocation of the method `startElement(..., String qName, Attributes attributes)`. When this method is executed, the `qName` parameter holds the name of the current XML tag, for example `<service>` or `<intent-filter>`. The `attributes` parameter instead contains all the specified attributes of the given tag, like the values of the above mentioned `android:exported` and `android:permission` fields.

5.5.3 Source-sink search

Once a list of candidate services has been populated, the search for the vulnerable pattern must be executed in each of them. The first step executed by this component of the tool is to build the *inheritance hierarchy* of the service under investigation. This is needed because the call to `Service.startForeground(int id, Notification n)` can happen in every class belonging to the hierarchy. Moreover, the name of the service retrieved from the manifest fetcher is the name of the *lowest* class in its hierarchy, and this means that there can be other classes in between that must be included in the analysis. We found that the studied applications use 4 or 5 levels of inheritance on average, meaning that most of the built hierarchies were composed of 4 or 5 classes.

Formally, we can define the hierarchy H of a service Σ as an object with the following properties:

- H is a set of classes: $H = \{\Gamma_1, \Gamma_2, ..., \Gamma_n\}$;
- $\Gamma_1 \in \{\text{“}Service\text{”}, \text{“}IntentService\text{”}\}$;
- $\Gamma_n = \Sigma$;
- $\forall i \in \{1, ..., n-1\}$: Γ_{i+1} `extends` Γ_i;
- $H(k) = \Gamma_k$, meaning that we can extract a specific class from the hierarchy given its index k, with $1 \leq k \leq n$.

The first class in the hierarchy is one between *Service* and *IntentService*. The latter is a specific variation of the general framework Service class, used for a well-defined purpose: executing actions sequentially in a system-created worker thread, avoiding any concurrency mechanism. Their general working principles do not differ from a standard service, so they have to be included in order to consider every kind of services.

```
Read currentClass line by line {

    store packageName;
    collect all import statements;

    if(class declaration is found)
        store superClass;
        break;

}

/*If currentClass actually extends a superClass*/
if(superClass is not null){

    /*If the top class is Service or IntentService, start building H and return the positive
        outcome*/
    if(superClass equals Service or IntentService)
        H.add(superClass);
        return 1;

    /*Look for superClass in the package directory first*/
    if(isInDirectory(superclass, packageName))
        currentClass = superClass;
        if(recur() == 1)
            H.add(superclass);
            return 1;

    /*Look for superClass in paths denoted by import statements previously collected*/
    for(import i : importsList)
        if(isInDirectory(superClass, i))
            currentClass = superClass;
            if(recur() == 1)
                H.add(superClass);
                return 1;

}

/*Negative outcome: either H does not terminate with Service or IntentService, or a
    superClass file was not found.*/
return 0;
```

Figure 5.4. Pseudocode describing the approach used for building the hierarchy of a service, starting from the lowest class.

The pseudocode in Figure 5.4 describes the steps required to recursively build the hierarchy of a service, given the name of its lowest class found in the manifest.

Once we built the inheritance hierarchy of the service under analysis, all the source files corresponding to the classes in the hierarchy are available for the source-sink search.

The next step is to define ways to identify *declarations* and *invocations* of methods. This is useful since we are performing analysis directly on source files, without executing the code, and so we need a way to understand what declared method is

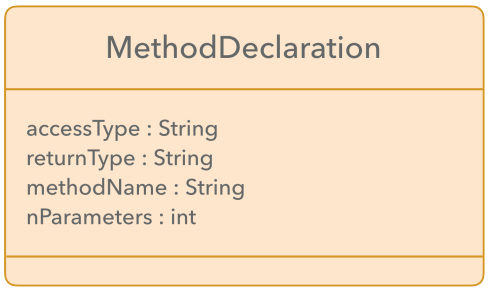

Figure 5.5. Class diagram of MethodDeclaration class.

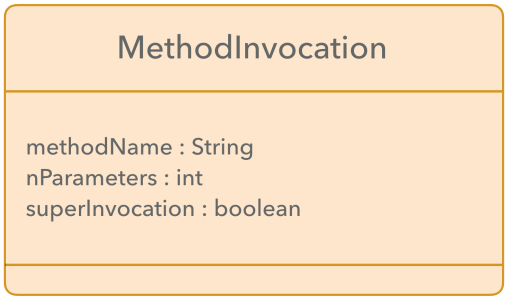

Figure 5.6. Class diagram of MethodInvocation class.

addressed by each invocation. To achieve this we defined the classes *MethodDeclaration* and *MethodInvocation*, that respectively model the signature of a method and its invocation in the source code. Since a declaration holds more information about a method with respect to its invocation (accessibility scope, return type, type of every parameter), we considered a *MethodInvocation* to be referring to a *MethodDeclaration* when the two objects had the same method name and the same number of parameters.

Moreover, the MethodInvocation class holds a *boolean* variable that is set to `true` if the corresponding invocation is explicitly referring to a superclass method declaration through the `super` keyword. This is required to correctly follow the execution flow when a subclass overrides a method, but it requires to execute its superclass implementation.

The detection of a method declaration and a method invocation is achieved through the use of *regular expressions*, that are also exploited for other tasks like:

- The exclusion of comments;
- The detection of the `package` statement, that allowed the identification of the path within the application folders where every source file is stored;
- The identification of every `import` statement, representing the paths to every other classes that are used in the class under analysis.

Finally, the central task of this module is performed by an object of class *FileCursor*, that has the following attributes:

- `int componentState`: an integer whose possible values are `VULNERABLE` (0) or `SAFE` (1), indicating the current state of the analysed component;
- `int currentClassIndex`: the index of the class in the hierarchy that is being analysed;
- `File currentClass`: the File object of the current class;
- `MethodDeclaration currentMethod`: the method that the FileCursor is currently looking for in `currentClass`.

Table 5.1. FileCursor Primitive I

public void hierarchyUp()
`this.currentClassIndex-` `this.currentClass =` H`(this.currentClassIndex)`

Table 5.2. FileCursor Primitive II

public void hierarchyDown()
`this.currentClassIndex++` `this.currentClass =` H`(this.currentClassIndex)`

Moreover, the FileCursor class defines two useful primitives to traverse the given hierarchy.

The main task of the FileCursor class is to scan a source file and keep track of the executed methods chain, labelling the analysed component as vulnerable or safe accordingly. The pseudocode in Figure 5.8 describes the approach used by this class. The following initialisation makes the algorithm start from the lowest class in the hierarchy, searching for the source method. Let:

- `sourceMethod` be the `MethodDeclaration` object relative to `public int onStartCommand(Intent i, int flags, int startId)` or to `protected void onHandleIntent(Intent i)`, depending on the first component of the hierarchy;
- `sinkMethod` be the `MethodInvocation` object relative to `public void startForeground(int id, Notification n)`;
- H be the hierarchy of the component under analysis;
- `cursor` be a `FileCursor` object such that:
  - `cursor.componentState = VULNERABLE`;
  - `cursor.currentClassIndex = size(H)`;
  - `cursor.currentClass = H(cursor.currentClassIndex)`;
  - `cursor.currentMethod = sourceMethod`.

After the analysis, all the services belonging to the list of candidates components are labelled as “safe”, “vulnerable” or “vulnerable with permission”. The last label is assigned to those services in which a source-sink path is not found, with the additional requirement of at least one permission needed to start them as declared in the manifest. According to this distinction, the original list is split in two collections, that are the input of the final component. The services vulnerable with permission are treated as standard vulnerable services for the testing phase, due to the nature of the last component of the tool.

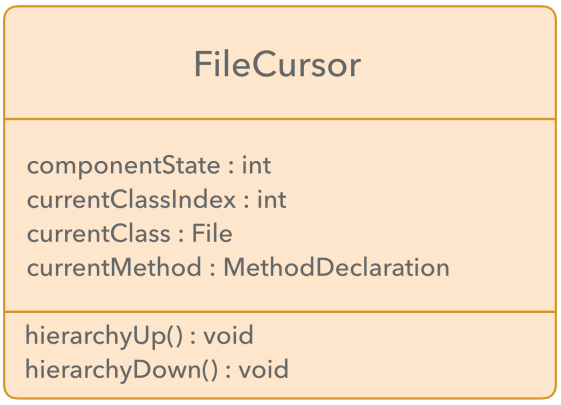

Figure 5.7. Class diagram of FileCursor class.

```
/*Terminate because we reached one of the termination classes*/
if c.currentClass == H(1)
    return

/*Terminate because we found the sinkMethod*/
if c.currentMethod == sinkMethod
     c.componentState = SAFE
     return

Let D be the collection of MethodDeclaration objects whose signature match
c.currentMethod

/*c.currentMethod is not found in c.currentClass*/
if size(D) == 0
     c.hierarchyUp()
     recur()
     c.hierarchyDown()
else
    for MethodDeclaration d in D {
        scan c.currentClass until d is reached
        read d line by line {
            if a MethodInvocation I is found {
                c.currentMethod = I
                if I is a superClass invocation
                    c.hierarchyUp()
                    recur()
                    c.hierarchyDown()
                else
                    recur()
            }
        }
    }
```

Figure 5.8. Pseudocode describing the approach used for the classification of a service.

5.5.4 Automatic intent sender

The final step is to evaluate the labels assigned to services and to compare the expected behaviours with the actual ones on a connected device. In order to complete this task, the *Android Debug Bridge*, or ADB, can be conveniently exploited. It is a tool part of the Android Software Development Kit (SDK) that can be used to send commands to a real or emulated device. These commands are directly derived from the API exposed by system services, so that there is no need to write a complete application for testing purpose. This means that methods like `Context.startForegroundService(Intent i)`, part of the API of the AMS system service, can be directly executed from command line. The required command is:

```
$ adb shell am start-foreground-service -p package_name -c
   service_name
```

The `p` (package) and `c` (component) options emulate the creation of an explicit intent, because the unique couple {package name, component name} is provided, allowing ADB to immediately identify the target service in the target application.

Moreover, the use of ADB allowed us to test even those services that require permissions, because it can execute commands on a device on behalf of the *root* user, that has any permission automatically granted. This is achieved through the command:

```
$ adb root
```

This is the reason why vulnerable services that required permissions in order to be started have been associated to standard vulnerable services in the outputs of the third module.

In order to understand if a process crashed, the *logcat* can be used. It is a record of every event of every process in execution on a device, and it is composed by three different channels: main, system and crash. The crash channel is the one where crashes and errors reports are written, and so it can be analysed by reading it from the most recent entry.

Now that we have a way to execute `startForegroundService(Intent i)` and to check if a process crashed or not, we can automatically perform tests by executing the following steps:

1) Read the list of vulnerable services;

2) For every service `S` in the list:

2.1) Clear the crash channel of logcat;

2.2) Start the main activity of the application hosting `S`;

2.3) Execute `startForegroundService(Intent i)`, with `Intent i` targeting `S`;

2.4) Inject a touch event to dismiss the error message dialog (if any);

2.5) Read from the crash channel of logcat to check the result;

2.6) Loop to 2.1)

After having performed tests on every service labelled as vulnerable for a given application, we can compare the actual results and we can estimate the precision and reliability of the tool.

In order to completely analyse the given application, we decided to perform tests also on the list of the services labelled as safe. This allowed the classification of the results as:

Table 5.3. Tests results classification

Label	**Test Outcome**	**Result Type**
Vulnerable	Crash	**True Positive**
Vulnerable	No Crash	**False Positive**
Safe	Crash	**False Negative**
Safe	No Crash	**True Negative**

5.6 User Manual

In order to be correctly executed, the tool has the following system and software requirements:

- operating system: Unix-based OS or MAC OS;
- Java 8 [32];
- The Android SDK with the ADB extension [33];
- ApkTool;
- Jadx;
- An Android device connected to ADB (only for the evaluation phase).

The tool is delivered as a collection of source files and bash scripts, allowing it to be started directly from the command line. An initial configuration is required, and it mainly involves the filling of variables storing the locations of the external tools used:

- In *vulnerableComponents.sh*, the variables JADX_DIR and APKTOOL_DIR hold the paths to the binary images of Jadx and ApkTool respectively. Moreover, OUTPUT_DIR can be modified to specify a custom location for the outputs of the tool (default is the current folder), while JADX_OUT holds the location of the output of the reverse engineering step;
- In *evaluateServices.sh*, the variable ADB_DIR must be filled with the path storing the binary file for the ADB tool.
- The variables ANDROID_ROOT and SDK_ROOT contain respectively the path to the AOSP Android source and the source code of the version of Android under analysis.

These variables can be populated by running:

```
$ make config
```

After the execution of this command, the file *config.cfg* will be available. Then, the compilation of the source codes for the Java components can be achieved through:

```
$ make install
```

Once the configuration and installation phases are performed, the user can start to use the tool for the analysis of applications.

The presence of various options allows users to perform a complete or partial study of the desired apk file. To perform the analysis corresponding to the tasks of the modules dedicated to the vulnerability detection (modules 1 to 3), the following command is available:

```
###### Components analysis for Application: spotify ######

Found 3 exported components:
Component:
        Name: com.spotify.mobile.android.spotlets.androidauto.SpotifyMediaBrowserService
        Type: S
        Permission: none
        Intent Filters:
                action : android.media.browse.MediaBrowserService

Component:
        Name: com.spotify.mobile.android.spotlets.appprotocol.service.AppProtocolRemoteService
        Type: S
        Permission: none
        Intent Filters:
                action : com.spotify.mobile.appprotocol.action.START_APP_PROTOCOL_SERVICE

Component:
        Name: com.google.android.gms.auth.api.signin.RevocationBoundService
        Type: S
        Permission: com.google.android.gms.auth.api.signin.permission.REVOCATION_NOTIFICATION
        Intent Filters: none



###### Vulnerability analysis for Application: spotify ######

Class SpotifyMediaBrowserService.java: vulnerable
Class AppProtocolRemoteService.java: vulnerable
Class RevocationBoundService.java: vulnerable (requires permission)
```

Figure 5.9. Output of the analysis of the exported services of application Spotify v.8.4.48

```
$ ./vulnerableComponents.sh [-da] apk_path apk_name
```

Where:

- Input `apk_path` is the location where the apk file of the application under analysis is stored;
- Input `apk_name` is a custom name given to the apk that will be used as name of the folder containing the outputs;
- Option `-d` makes the script skip the apk decompilation phase. This can be useful to perform analysis on already decompiled apks;
- Option `-a` makes the script skip the analysis phase. In other words this option only makes the script perform the reverse engineering step.

If an Android application is analysed with this command, the list of potentially vulnerable services will be available at the location specified in OUTPUT_DIR during the configuration.

After that, the evaluation phase can be performed if an Android device (either real or emulated) is connected to the user terminal. Depending on the type of connected device, the connection is established in two different ways. For a real device,

either the USB or the Wi-Fi connection can be used. The device must have the *USB Debug* enabled, available in the *Developers Options* settings. To start and connect an emulated device instead, the following command is needed:

```
$ ADB_DIR/tools/emulator -avd device_name
```

Once the connection is established, the evaluation can take place thanks to the command:

```
$ ./evaluateServices.sh [-f [-n N | -o M]] apk_name
```

Where:

- Input `apk_name` is the same name previously given to the apk;
- Option `-f` makes the script perform the false negative check, testing those services labelled as safe;
- Option `-n` makes the script test only the first N services (can be combined with `-f`)
- Option `-o` makes the script test only the M^{th} service (can be combined with `-f`)

Finally, users can decide to perform a complete analysis of an apk by combining the effects of the previously described commands with the execution of:

```
$ ./evaluateApk.sh apk_path apk_name
```

Where inputs `apk_path` and `apk_name` have the same meaning as in the previous commands. If this script is executed, no options will be used in the execution of *vulnerableComponents.sh* and *evaluateServices.sh*, resulting in both the complete analysis of the application and the evaluation of every service.

5.7 Results

We found that most party applications we examined are not properly aware of the presence of this vulnerability. We then analysed system applications, that are already pre-installed on a device and developed by Google.

5.7.1 Third party applications

We studied the top 30 free downloaded applications from the Google Play Store. We found that:

- Three of them did not expose any service to other applications;
- **Twenty-four applications** exposed at least one service requiring no permissions to be started or bound;
- The three remaining applications only exposed services requiring at least one permission from their client. In all the cases, these were `SIGNATURE` level permissions, thus restricting the use of their services only to system applications or applications published by their same companies.

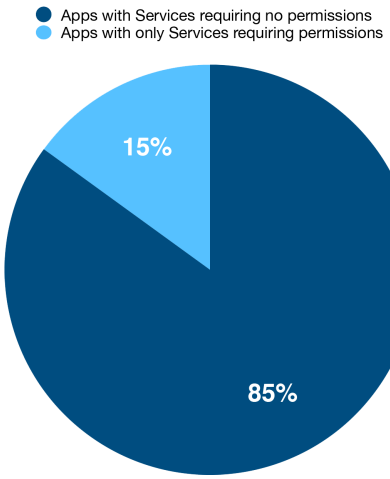

Figure 5.10. Third party apps classification according to declared services.

This difference already shows how the permission mechanism is not well exploited to protect services. Overall, the total number of the analysed services is 120, labelled in the following way by the tool:

- **117** have been classified as *vulnerable*;
- only **3** have been classified as *safe*.

After the tests, every label was confirmed to be correct, meaning that no false positives or false negatives were found.

The 3 Services labelled as safe belong to the applications Facebook, Facebook Messenger and Telegram. However, since these applications also exposed vulnerable services, each of them being able to make each process crash, they cannot be considered completely safe. On the other hand, from a practical point of view, the 3 applications that restrict the use of all their services (6 in total) through `SIGNATURE` level permissions can be considered reasonably safe, because external apps cannot have such permissions granted and so they do not have access to these protected services.

5.7.2 System applications

Despite being developed by the Android team itself, we discovered that system applications suffer from this flaw in the same way. However, in the case of system apps, the implications are usually more serious with respect to third party applications, because system applications usually host very important services, like the `phone` service, in charge of managing the phone calls protocols mechanisms. The 10 analysed system applications are: Camera, Contacts, Google Chrome, Google Maps, Google Music, Google Photo, Google Video, Phone, Settings, YouTube. As with third party applications, we found that:

- The Camera application does not export any service;
- **7 Applications** expose at least one service requiring no permissions to be started or bound;
- Among the 2 applications that only expose services requiring a permission (Contacts and Settings):

  a) none of them require at least one `NORMAL` level permission;

  b) only Contacts requires one `DANGEROUS` level permission;

  c) the Settings app only requires `SIGNATURE` level permissions.

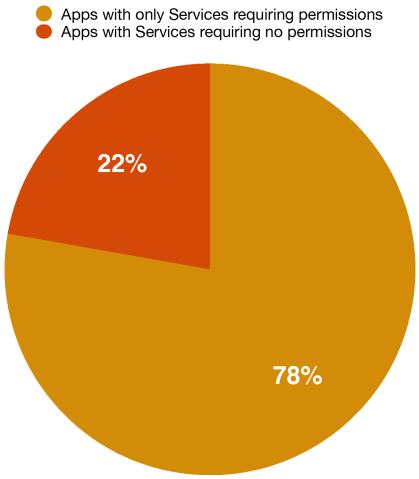

Figure 5.11. System apps classification according to declared services.

The performed tests demonstrated how system apps are slightly more robust with respect to third party applications, for two main reasons:

1. Some applications (like the Google Music application) make use of two different processes when in execution. Generally the first process is dedicated to UI management, while the second to heavier and more time consuming computations;

2. Most system services hosted by system applications are designed to automatically restart after they are killed for unexpected reasons.

The performed tests highlighted the following results:

- **64 Services** were labelled as vulnerable, with:
    1. 60/64 (93.75%) of them being true positive, thus leading to the ANR error dialog window;
    2. 4/64 (6.25%) of them being false positive, because only one of the two processes of the analysed application crashed.
- No services were labelled as safe.

5.7.3 Consequences on Phone app

Among the vulnerable system applications, the Phone app is one of the most relevant because it hosts the `com.android.phone` process.

This process manages ingoing and outgoing phone calls, as well as the strength of the received phone signal. The application hosts 7 services, of which 4 are exported. Among them:

- Service `com.android.phone.otasp.OtaspActivationService` does not require any permission to be started;
- Services `com.android.phone.TelephonyDebugService`, `com.android.services.telephony.sip.SipConnectionService`, `com.android.services.telephony.TelephonyConnectionService` require `SIGNATURE` level permissions.

The starting of each of these 4 services in the foreground through `Context.startForegroundService(Intent i)` always results in the crash of the `phone` process. One major consequence of the crash is the immediate loss of the phone signal. This means that if a malicious application obtains the dangerous-level permission `READ_PHONE_STATE`, it can understand if a phone call is currently ongoing in the device and programmatically crash the `phone` process to immediately close the communication channel. However, since the `phone` process is restarted by the system_process as soon as it is killed, Android automatically makes the phone call start again. This can ignite a infinite loop in which the malicious application keep closing the phone call as soon as it is restarted, making the device impossible to use without a reboot.

Given the previous considerations on the ease of the users to grant many permissions to downloaded applications, the described scenario can be easily implemented in a real life application, representing a serious threat to the system.

Chapter 6

Real implementation cases

We shall now proceed to analyse how a malicious application can trigger this vulnerability.

The only parameter of the `Context.startForegroundService(Intent i)` method is an intent object that must target the service that the calling application is requesting to start. In Chapter 2 we analysed the difference between explicit and implicit intents, and the concept of intent filter. To recap:

- **Intent Filters** are sets of values declared (statically) in the manifest or (dynamically) directly in the application source code. They consists of strings labelled as actions, category, data, type and extras;
- **Explicit Intents** are created with the couple {package name, class object}, that uniquely identify a component within an application;
- **Implicit Intents** are created assigning to the fields actions, category, data, type and extras of a new intent the values corresponding to the ones declared by the intent filter of the target service. Thanks to the intent resolution process, the OS can understand which component is aimed by the intent.

```
public Intent getExplicitIntent(){

    Context appContext = getApplicationContext();
    Intent i = new Intent(appContext, ServiceToStart.class);

    return i;

}
```

Figure 6.1. Explicit intent definition for a service declared in the same application that starts it.

We show how a malicious application can decide to leverage either implicit or explicit intents in order to ignite the vulnerability and obtain the same exact results.

```
public Intent getImplicitIntent(){

    Intent i = new Intent();
    i.setAction("android.intent.action.SET_ALARM");
    i.addCategory("android.intent.category.DEFAULT");

    return i;

}
```

Figure 6.2. Implicit intent definition with one action and one category.

We will exploit the capabilities of a very powerful API available in some programming language, and some hidden features of two classes. In both the situations, the attack could be executed without restrictions or exceptions depending on the target applications.

6.1 Using explicit intents

6.1.1 Introduction to Reflection API

In order to build an explicit Intent, an object of the special Java class *Class* is required. These objects are especially used with the *reflection API*, a feature of some programming languages (Java and Kotlin included) that allows the introspection of classes at runtime. Indeed, a Class object is the representation of every aspect of a given class: its attributes (represented by objects of class Field), methods (represented by objects of class Method), constructors (objects of class Constructor), implemented interfaces, extended classes and so on.

This means that given the name of a class, we can retrieve every information about it by loading the corresponding class object and exploiting reflection API methods like `Class.getDeclaredFields()` and `Class.getDeclaredMethods()`.

Alongside the inspection of the properties of a class, one of the most interesting features of reflection is the capability of executing every method a class declares. This is achieved with the following steps:

1. Load the *class* object of the class implementing the desired method;
2. Obtain the constructor object through `Class.getDeclaredConstructor(Class... params)`. The object `params` is an array with all the class objects corresponding to the classes of the arguments needed by the constructor, in declaring order;
3. Instantiate a new object of the class with the method `Constructor.newInstance(Object... args)`. This object will be referred to as the *receiver*. The `args` array now holds all the actual objects used by the constructor;

4. Obtain the Method object of the requested function we want to execute through `Class.getDeclaredMethod(String name, Class... params)`. The `params` array contains again the class objects of the parameters needed by the method in correct order;

5. Execute `Method.invoke(Object receiver, Object... args)`. The `receiver` is the previously obtained instance, and `args` is an array with all the parameters requested by the method.

Figure 6.3 shows the implementation of the described steps to achieve the invocation of the method `obtainMessage(int what)` of class `Handler`.

These steps imply that in order to execute a function from a class, a receiver of that class is needed. However, if the method we want to execute is a *static* method, the receiver parameter is `null`. Executing the constructor to obtain a receiver instance can be a time consuming task, because it can imply calling other methods or exploiting some properties of the passed parameters, that thus must be properly populated. That is why focusing on static methods can be really convenient, as we will understand in these sections.

6.1.2 How class objects are retrieved

One of the standard ways to obtain the *class* object of a specific class is through the static method `Class.forName(String name)`, where the `name` parameter is the complete name of the required class. This means that if we want to load the class object of the Java `String` class, we need to refer to it as `java.lang.String`. This method uses another important class in the context of reflection, named *ClassLoader*. As their name suggests, objects of this class have the responsibility to retrieve class instances. Every class loader holds a path to a specific location in the device filesystem that represents the starting point for the search of class objects, that are stored in .apk, .jar, .zip and .dex files. For this reason, every downloaded application has a default ClassLoader, assigned by the OS at installation time, whose class path is set to the location of the .apk file of the application itself.

This means that the default class loader of every application is able to load all the classes available in that application. System applications instead have access to those classes that are private or hidden to every other app. Android enforces this protection through the standard accessibility keywords that define the scope of a class (like `private` and `protected`) and through annotations directly in the source code, that prevent the compilers from accepting class names that it does not recognise as part of the official documentation (one example is the @hide annotation).

If a third party application tries to load objects of these protected classes, the result is a `ClassNotFoundException`. One consequence of this mechanism is that there exists a hierarchy among class loaders, with some of them being more powerful with respect to others, being able to load classes that other class loaders cannot see.

```
public void reflectionInvocation(){

    try {

        //Handler class object
        Class handlerClass = Class.forName("android.os.Handler");

        //Retrieve the Constructor object for Handler class
        Class looperClass = Class.forName("android.os.Looper");
        Constructor handlerCt = handlerClass.getDeclaredConstructor(looperClass);

        //A Looper object is needed for the instantiation
        Looper myLooper = getMainLooper();
        Handler myHandler = (Handler) handlerCt.newInstance(myLooper);

        //Retrieve the Method object for Handler.obtainMessage(int what)
        Class intClass = int.class;
        Method obtainMsg = handlerClass.getDeclaredMethod("obtainMessage",
            intClass);

        //An integer value is needed to invoke the method
        int value = 1;

        //Method invocation
        Message myMsg = (Message) obtainMsg.invoke(myHandler, value);

    } catch (Exception e){

        e.printStackTrace();

    }

}
```

Figure 6.3. Use of reflection API to invoke a method of class Handler.

6.1.3 Exploiting class loaders

The properties of class loader objects emphasised so far imply that for an external application it is impossible to retrieve the class object of a target service hosted by another application, because its *default* ClassLoader does not have the ability to load classes of different apps. This makes the instantiation of an explicit intent impossible.

However, our study found a mechanism to programmatically load *every class* of the OS, as well as of every installed application on the device. We exploit a particular subclass of ClassLoader, called PathClassLoader. One of the two constructors of this kind of object has the following signature:
`public PathClassLoader(String classPath, String libPath)`.
The parameters represent:

- `classPath`: it is the string with the path pointing to an .apk, .jar or .zip

file containing a *classes.dex* file, or directly to a classes.dex file. This is the location that this PathClassLoader will use to load class objects;

- `libPath`: in the same way, ClassLoaders can load native libraries (.so files). This is the path used by this PathClassLoader to load such libraries. We will examine the implications of this parameter in Chapter 8.

This means that by providing the right path, we can decide to use the PathClassLoader instance to load every class we need, through the inherited method `ClassLoader.loadClass(String className)`, instead of using the default class loader of our application.

Surprisingly, we noticed that these objects can be instantiated even providing protected paths. Since Android is a Unix based OS, it uses the classical access control model of every Unix-like system. This implies that every file has a owner as well as reading, writing and executing permissions for the classes user, group and others. For example, the most important classes that the system_process instantiates when it is first executed are stored in a .jar file found at the path `/system/frameworks/services.jar`. This file is owned by root, and no other user or group has read, write or execution privileges on it. However, through a PathClassLoader, we are able to access the file and use it to load every class it holds. In this way, objects of classes like ActivityManagerService can be instantiated, and they can be used with standard reflection methods.

6.1.4 How Context objects are retrieved

The second component of an explicit intent is an object of the abstract class *Context*. Normally, every application has access to its own context through the methods `Context.getApplicationContext()` and `Context.getBaseContext()`. These methods return an object of class Context holding all the information of the calling application. However, in order to make the explicit intent target an external application, the context of that particular app must be retrieved. The easiest way to achieve this task is through the method `Context.createPackageContext(String packageName, int flags)`. This method returns the context object of the application identified by its unique package name, that is publicly available for every app that can be downloaded from the Google Play Store. The second parameter can be a combination of zero or more different flags, with the following meaning:

- `CONTEXT_INCLUDE_CODE` (integer value 1): If this flag is set, the returned object will also contain the class loader able to load classes from the requested application. However, this could raise a SecurityException, as it happens with system applications, whose class loader cannot be retrieved in this way;
- `CONTEXT_IGNORE_SECURITY` (integer value 2): if this flag is combined with `CONTEXT_INCLUDE_CODE`, the class loader of the requested app will always be included;

- `CONTEXT_RESTRICTED` (integer value 4): this flag makes `createPackageContext()` return a restricted context.

One of the most interesting properties contained by a context object retrieved in this way is the exact path of the .apk file of the requested application. The attribute holding this information is called `mPackageInfo`, and it is an object of class LoadedApk. Thanks to the method `Context.getPackageCodePath()`, its value can be retrieved.

6.1.5 Creating the explicit intent

Now that we are able to obtain the {`Context`, `Class`} objects pair, we can instantiate the explicit intent that will be used as parameter of `Context.startForegroundService(Intent i)`. This can programmatically be achieved for every installed application following these steps:

1. Obtain the context of the requested app;
2. Retrieve the .apk file location on the device;
3. Create a new PathClassLoader, whose class path is the previously obtained path;
4. Obtain the class object of the service we want to start. The name can be read from the `AndroidManifest.xml` file;
5. Create a new explicit intent;
6. Pass this intent to `startForegroundService()`.

Figure 6.4 shows a real implementation case with the explicit intent targeting one of the vulnerable services of the messaging application Whatsapp.

The method `createPackageContext()` throws a `PackageManager.NameNotFoundException` exception if the requested package name cannot be found. If this happens it means that the application that the attacker is trying to kill is not installed, so that this procedure can even be exploited to understand which apps are available on the device, and error situations (like trying to kill a non installed application) can be silently handled thanks to the try-catch block.

6.2 Using implicit intents

The procedure to start a service using an implicit intent does not involve the use of class loaders, because the class object is not needed anymore. Indeed, the OS recognises which component in which application an implicit intent is targeting by comparing the intent fields action, category, data, type and extras with the ones declared in the intent filter of every component. For this reason, implicit intents

```
public void startWhatsappService(){

    Context wContext;
    String wCodePath;
    PathClassLoader wLoader;
    Class wService;
    Intent i;

    try{

        Context wContext = createPackageContext("com.whatsapp",
            CONTEXT_IGNORE_SECURITY);
        String wCodePath = wContext.getPackageCodePath();

        wLoader = new PathClassLoader(wCodePath, null);
        wService = wLoader.loadClass(
            "com.whatsapp.accountsync.AccountAuthenticatorService");

        i = new Intent(wContext, wService);
        startForegroundService(i);

    } catch (Exception e){

        //Handle the exception...

    }

}
```

Figure 6.4. Explicit intent creation procedure for application Whatsapp.

can be used only when the target service declares an intent filter. Moreover, we already highlighted how the same filter can be declared by different components, possibly leading to ambiguities in the choice of the target of the intent. Despite the presence of some techniques aimed to reduce these ambiguities, like the possibility to define custom values for the intent filter fields, Android is constantly discouraging the use of implicit intents in general. Indeed, starting from Android 5.0 (API level 21), it is not possible to call `Context.bindService(Intent i)` with an implicit intent, mainly for security reasons [22]. In the same way, the execution of `Context.startService(Intent i)` and `Context.startForegroundService(Intent i)` with implicit intents generates a `IllegalArgumentException` on the application that is starting the service that reads as: "*java.lang.IllegalArgumentException: Service Intent must be explicit*".

6.2.1 Faking the Context

We now introduce a procedure that allows a malicious application to use implicit intents to start a service in the foreground and ignite the termination of its process, regardless of its API level. This mechanism is a specific exploitation of a wider approach that allows the modification of the context objects that are returned by

the methods `getApplicationContext()` and `getBaseContext()`. The reflection API is again useful in this case, because it allows the *modification* of the values of every field within objects. However, standard reflection methods cannot be directly applied on objects of class Context. The reason is that Context is an abstract class, and reflection needs a concrete receiver object to work on. Android uses the hidden class *ContextImpl* to implement the methods defined in Context. For this reason, the first thing we need to do is to find a way to perform a cast from the superclass, Context, to the subclass ContextImpl. Fortunately, the latter class defines the static method `ContextImpl.getImpl(Context context)`, that returns the ContextImpl object corresponding to the passed parameter. As previously said, the presence of a static method is really useful since no receiver of ContextImpl class is required. Since the class object of ContextImpl is accessible through reflection, we can inspect this class and obtain the method object corresponding to `getImpl()`. Finally, we can invoke it on a `null` receiver, passing the object returned by either `getBaseContext()` or `getApplicationContext()` as parameter.

Once we retrieve the concrete implementation of the context of our application, we can access all its field and modify them at our will. In particular, we can access the field `targetSdkVersion`, and modify it at runtime by providing a value lower than 21. In this way, when the intent to start the service is passed to the method `ContextImpl.validateIntentService(Intent i)` during the execution of `startForegroundService()`, Android will consider our application as being developed for API levels lower than 21. Since the starting and binding of services with an implicit intent was allowed in these older versions, the intent will pass the security check, and the service will be started normally. The log of the application requesting the start of the service will display a warning message saying: "*Implicit intents with startService are not safe*", followed by the details of the intent, but the application will be executed normally and the intent will be delivered correctly.

Chapter 7

Proposed solutions

7.1 General considerations

The analysis of the vulnerability highlighted that the current Android design is missing at least two key mechanisms:

1. How an application can understand if one of the services that it declares is being started in the foreground or not;
2. How an application can infer if a service can be started in the foreground.

Indeed, by providing applications with such capabilities, the application starting the service could properly choose to execute `Context.startService(Intent i)` or `Context.startForegroundService(Intent i)`. Moreover, the application hosting the service could call `Service.startForeground(int id, Notification n)` only if the service is being started in the foreground, avoiding the crash of the process and allowing it to keep running safely.

For these reasons we present some potential solutions that address and solve these problems without a big overhead in terms of computation and complexity.

7.1.1 New onStartCommand() prototype

An immediate solution is to modify the prototype of the `Service.onStartCommand(Intent i, int flags, int startId)` callback into `Service.onStartCommand(Intent i, int flags, int startId, boolean isForeground)`. The reason why this solution would be easily affordable is that the system_process already keeps a track of how a client is starting a service (background or foreground) through a boolean variable. By propagating this variable to the requested service, the hosting application is provided with a mechanism to understand if `Service.startForeground(int id, Notification n)` must be called or not.

Of course, this solution implies the modification of a widely used callback, so this could represent a problem in terms of backward compatibility. However, the old prototype could still be used by applications that target API levels 25 and lower,

while the new one should be enforced for applications targeting API level 26 and higher. This technique has already been adopted in Android Oreo with the method `Activity.findViewById(int id)` that changed its return type from `View` to the generic `<T extends View>` [34].

7.1.2 Leveraging the permissions system

Another approach is represented by the exploitation of the Android permissions system. We propose the introduction of a new *dangerous* level permission, that will allow the application willing to start a service to know if such service can be started in the foreground or not.

With this approach, the application defining the service should include this permission with the attribute `android:permission` in the `AndroidManifest.xml` file. In this way, the app is announcing to the system_process that all the services that require the new permission are ready to be started as foreground services, and so they correctly execute
`Service.startForeground(int id, Notification n)`. Since the new permission belongs to the dangerous level, the first time that an application will try to execute `startForegroundService()`, the user will be prompted with a dialog asking to allow or deny the starting of the service in the foreground. By definition, a foreground service executes operations that the user is aware of [35], so the dialog window would represent an enforcement of this acknowledgement. Figure 7.1 shows the execution flow achieved by the introduction of this new dangerous level permission.

7.1.3 Final considerations

Each of the proposed solutions targets only one portion of the issue. In particular:

- The new prototype of `onStartCommand()` solves the problem of allowing the hosting application to know if one of its services is being started through `startService()` or `startForegroundService()`. This gives the app the chance to call `startForeground()` only when needed. However, the requesting application can still freely decide which of the 2 methods to use to start the service;
- The new dangerous level permission makes the requesting application aware of how the requested service can be started. The drawback is that the hosting app would not know whether to call `startForeground()` or not.

Ideally, a combination of the two approaches would address both the problems, providing a complete solution to this design flaw.

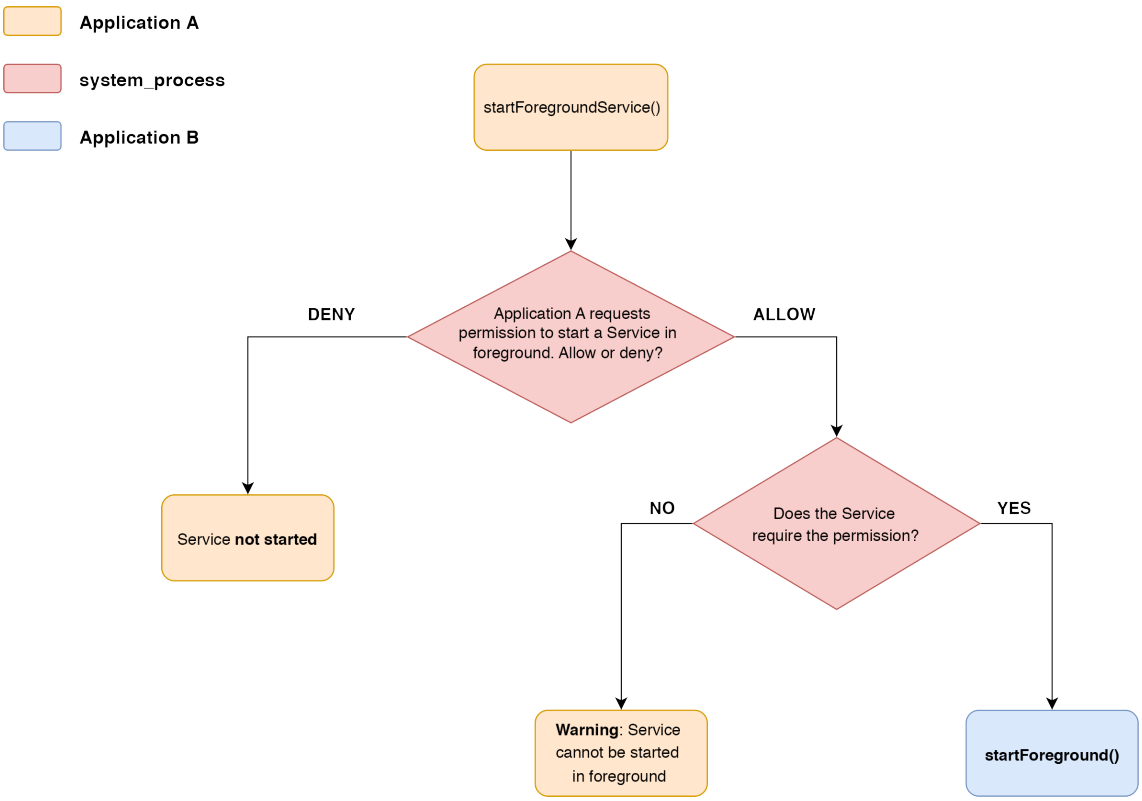

Figure 7.1. The execution flow generated by the introduction of a new dangerous level permission never leads to the crash of the application hosting the service.

Chapter 8

Conclusions

Our work exposed a design flaw in the Android OS that potentially makes every application vulnerable to a DoS attack. For this reason, we decided to report this issue directly to the Android security team, through the Google IssueTracker platform [7]. After some time, our report was assigned to a developers team, and it is under investigation at the time of writing.

8.1 Fix in Android P

The next version of Android is scheduled to be released in Autumn 2018. While its official code name is not known yet, we currently refer to it as *Android P*, following the alphabetical order that gave name to all the prior versions. Programmers have access to the Android P Developer Preview, that allows them to install the latest beta version of Android P and to start adapting their applications to the new API level 28.

Among the new features of Android P, we can find one hint that suggests the intention to fix the foreground services issue. Indeed, Google decided to introduce a *new permission* called `FOREGROUND_SERVICE`. This permission belongs to the normal level permissions, and "*Allows a regular application to use Service.startForeground.*" [36].

However, despite the several modifications that the documentation could receive in the next months prior to the final release, we argue that this new permission would not solve the problem at all. The reasons are mainly two:

1. The new permission is a *normal* level permission. This means that applications requesting it will automatically have this permission granted, denying to users any possibility to avoid the start of a vulnerable service in the foreground;

2. According to the current documentation, the permission must be requested by applications willing to execute `Service.startForeground(int id, Notification n)`. In other words, the permission must be asked by the application that declares and hosts the service. Instead, as we proposed in Chapter 7, we think that a

permission should be granted to the applications that *request* the start of a service hosted by another application.

In the end, if this permission will become part of the official Android framework with the release of Android P without any modification, the highlighted issue about foreground service will still be present.

In this case, the only way that developers have to avoid their applications to be programmatically crashed is to immediately execute `startForeground()` in all their services, even those not designed to run in the foreground, and even if the application requesting their start executed the standard `startService()`.

8.2 Future works

Our work on the current state of the reliability of the Android OS highlighted some potential flaws that could represent the starting point for future researches.

8.2.1 Context faking exploitations

Specifically, we highlighted how an application can fake every field of the context objects returned by methods `getApplicationContext()` and `getBaseContext()`. To the best of our knowledge, the combined exploitation of the *ContextImpl* class and the reflection API to achieve this result has not been explored yet. Since the context is the most used way for the identification of an application in the OS, this technique could represent the first step of a bigger attack vector.

8.2.2 ClassLoaders exploitations

We also gave evidence of the strength of objects of class *PathClassLoader*, a subclass of the abstract class ClassLoader. Thanks to these objects, the protections enforced by Android on specific classes can be bypassed, giving access to a whole new set of methods and functionalities thanks to reflection API. Through class loaders, we were able to demonstrate the feasibility of the implementation of attacks exploiting the vulnerability about foreground services.

However, our work also found another interesting capability of these objects. We recall that one of the constructors of a PathClassLoader has the signature: `public PathClassLoader(String classPath, String libPath)`. The `classPath` argument gave us the possibility to load classes from every location in the device filesystem storing a *classes.dex* file (like .apk, .jar and .zip files). The second argument, `libPath`, plays a similar role: it represents a location containing one or more *shared objects* (.so) files that can be linked to our application at runtime. These files are libraries that store the implementation of every *native* methods used in Android. A native method is a function implemented in `C` or `C++` languages that is invoked in a Java class. The conversion from Java variables and classes to C/C++ structures is managed by the *Java Native Interface*, or JNI, through a set of conventions and

rules that allows the exchange of data from one environment to the other and vice versa.

It follows that in order to execute a native method, an application must have access to the .so file with the implementation of the required function. If this condition is not satisfied, the result is an exception whose error message reads as "*No implementation found for int com.android.server.input.InputManagerService.nativeInjectInputEvent()*" (taking as example the method `int nativeInjectInputEvent()` of the InputManagerService class). Given the direct interfacing of the system_process with every aspect of the device (sensors, power management, touch screen, bluetooth receivers and so on), *system services* make a wide use of native methods that are implemented in specific .so files in the device. This means that we need a way to load such native libraries to completely take advantage of the capability of loading protected classes like the system services. The standard way that an application has to load a .so file is through the static method `System.loadLibrary(String libName)`, where the library denoted by `libName` must be located at the path stored in the `libPath` argument of the *default ClassLoader* of the application.

However, starting from Android 7.0, specific native libraries have been declared as *private* [37], so that the execution of the `loadLibrary()` method now results in the error "*java.lang.UnsatisfiedLinkError: dlopen failed: ...*". All the libraries containing the implementation of the native methods executed by the system services are part of this new set of private libraries: this means that the default class loader of a standard application cannot load these .so files.

Our study found a new way of overcoming this protection mechanism through the exploitation of a PathClassLoader whose library path is properly set. This method allows the dynamic loading of every .so file present in a device, private libraries included. In this way the implementation of the desired native method is retrieved by the OS, and the function can be actually executed.

For these reasons, the exploitation of PathClassLoader objects to gain access to protected classes and libraries (and thus methods) can potentially pave the way for new discoveries in the field of Android security.

8.3 Final considerations

Over the years, the continuous growth of Android has pushed this operating system to a very strong level of reliability. However, its worldwide diffusion represents its main limitation. It is users responsibility to keep their devices updated to the latest version of the OS, in order to protect their data from threats that are now considered to be obsolete or completely extinct, but that could easily harm devices running older versions of the operating system.

In our work, we analysed various sections of this OS, and we demonstrated how classes addressing apparently disjointed tasks can be combined to achieve unexpected results.

Bibliography

[1] Wikipedia, "Proxy pattern diagram." https://commons.wikimedia.org/wiki/File:Proxy_pattern_diagram.svg, 2007, [Online; accessed March-2018]

[2] Google, "Activity | Android Developers." https://developer.android.com/reference/android/app/Activity#activity-lifecycle, 2018, [Online; accessed March-2018]

[3] Google, "Content providers | Android Developers." https://developer.android.com/guide/topics/providers/content-providers, 2018, [Online; accessed March-2018]

[4] Google, "Intents and Intent Filters | Android Developers." https://developer.android.com/guide/components/intents-filters#Types, 2018, [Online; accessed March-2018]

[5] Google, "Services overview | Android Developers." https://developer.android.com/guide/components/services#LifecycleCallbacks, 2018, [Online; accessed March-2018]

[6] Google, "Bluetooth | Android Open Source Project." https://source.android.com/devices/bluetooth/#architecture-android-80, 2017, [Online; accessed March-2018]

[7] V. Chiaramida, "Google IssueTracker — Missing startforeground() causes crashes in almost every app." https://issuetracker.google.com/issues/78603020, 2018, [Online; accessed 05-June-2018]

[8] Statista, "Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 1st quarter 2018." https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/, 2018, [Online; accessed May-2018]

[9] Google, "Kotlin and android." https://developer.android.com/kotlin/, 2017, [Online; accessed May-2018]

[10] Google, "Codenames, Tags, and Build Number." https://source.android.com/setup/start/build-numbers#platform-code-names-versions-api-levels-and-ndk-releases, 2018, [Online; accessed May-2018]

[11] Google, "Permissions overview | Android Developers." https://developer.android.com/guide/topics/permissions/overview#normal_permissions, 2018, [Online; accessed May-2018]

[12] Google, "Request App Permissions | Android Developers." https://

developer.android.com/training/permissions/requesting, 2018, [Online; accessed May-2018]

[13] Google, "Android — 8.0 Oreo." https://www.android.com/versions/oreo-8-0/, 2017, [Online; accessed May-2018]

[14] Google, "Android 8.0 Behavior Changes | Android Developers." https://developer.android.com/about/versions/oreo/android-8.0-changes, 2017, [Online; accessed February-2018]

[15] NortonOnline, "Hundreds of malicious apps are showing up on the google play store, disguised as legitimate applications." https://us.norton.com/internetsecurity-emerging-threats-hundreds-of-android-apps-containing-dresse html, 2016, [Online; accessed May-2018]

[16] Z. Wang, C. Li, Y. Guan, Y. Xue, and Y. Dong, "Activityhijacker: Hijacking the android activity component for sensitive data", 2016 25th International Conference on Computer Communication and Networks (ICCCN), Aug 2016, pp. 1–9, DOI 10.1109/ICCCN.2016.7568487

[17] Y. Fratantonio, C. Qian, S. Chung, and W. Lee, "Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop", Proceedings of the IEEE Symposium on Security and Privacy (Oakland), San Jose, CA, May 2017

[18] K. Wang, Y. Zhang, and P. Liu, "Call me back!: Attacks on system server and system apps in android through synchronous callback", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, 2016, pp. 92–103, DOI 10.1145/2976749.2978342

[19] Armis, "The Attack Vector "BlueBorne" Exposes Almost Every Connected Device." https://www.armis.com/blueborne/, 2017, [Online; accessed March-2018]

[20] H. Lockheimer, "Android and Security." https://googlemobile.blogspot.com/2012/02/android-and-security.html, 2012, [Online; accessed June-2018]

[21] J. Oberheide, "Dissecting the android bouncer." https://jon.oberheide.org/blog/2012/06/21/dissecting-the-android-bouncer/, 2012, [Online; accessed May-2018]

[22] Google, "Android 5.0 Behavior Changes | Android Developers." https://developer.android.com/about/versions/android-5.0-changes#BindService, 2014, [Online; accessed March-2018]

[23] Google, "Activitymanager | Android Developers." https://developer.android.com/reference/android/app/ActivityManager.html#getRecentTasks(int,%20int), 2018, [Online; accessed March-2018]

[24] Google, "Activitymanager | Android Developers." https://developer.android.com/reference/android/app/ActivityManager.html#getRunningAppProcesses(), 2018, [Online; accessed March-2018]

[25] Google, "Activitymanager | Android Developers." https://developer.android.com/reference/android/app/ActivityManager.html#getRunningServices(int), 2018, [Online; accessed March-2018]

[26] Google, "Windowmanager.layoutparams | Android Developers." https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html#TYPE_APPLICATION_OVERLAY, 2018, [Online; accessed February-2018]

[27] Google, "Service | Android Developers." https://developer.android.com/reference/android/app/Service.html#startForeground(int,%20android.app.Notification), 2018, [Online; accessed February-2018]

[28] W. Ryszard and T. Connor, "Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps." https://ibotpeaches.github.io/Apktool/, 2012, [Online; accessed March-2018]

[29] Skylot, "jadx - Dex to Java decompiler." https://ibotpeaches.github.io/Apktool/, 2013, [Online; accessed March-2018]

[30] I. R. LLC, "Apkmirror." https://www.apkmirror.com, 2014, [Online; accessed March-2018]

[31] A. Henry, "Why Does This Android App Need So Many Permissions?." https://lifehacker.com/5991099/why-does-this-android-app-need-so-many-permissions, 2013, [Online; accessed May-2018]

[32] O. Corporation, "Download Free Java Software." https://www.java.com/en/download/, 2018

[33] Google, "Android Debug Bridge (adb | Android Developers." https://developer.android.com/studio/command-line/adb, 2018, [Online; accessed February-2018]

[34] Google, "Android 8.0 Behavior Changes | Android Developers." https://developer.android.com/about/versions/oreo/android-8.0-changes#fvbi-signature, 2017, [Online; accessed May-2018]

[35] Google, "Services overview | Android Developers." https://developer.android.com/guide/components/services, 2018, [Online; accessed February-2018]

[36] Google, "Manifest.permission | Android Developers." https://developer.android.com/reference/android/Manifest.permission#foreground_service, 2018, [Online; accessed June-2018]

[37] Google, "Android 7.0 behavior changes | Android Developers." https://developer.android.com/about/versions/nougat/android-7.0-changes#ndk, 2016, [Online; accessed Marc-2018]