Rovesti Gabriel

Mobile security simple (for real)

Summary

[Lecture 1: Mobile OSes and focus on Android security 2](#_Toc148079797)

[Lecture 2: Android Architecture: security models and app components 4](#_Toc148079798)

[Questionnaire 1 – Lecture 2 15](#_Toc148079799)

[Challenge 1 – filehasher 17](#_Toc148079800)

[Lecture 3 – RPC/IPC Communication, Binder, Android permissions 20](#_Toc148079801)

[Questionnaire 2 – Lecture 3 27](#_Toc148079802)

# Lecture 1: Mobile OSes and focus on Android security

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 prospects.
* **Community and Recognition:** Building popular apps can lead to recognition and a sense of accomplishment within the developer community.

Incentives vary depending on the platform (iOS, Android, or others) and the business model chosen by the developer.

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:

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

# Lecture 2: Android Architecture: security models and app components

Immagine che contiene testo, schermata, software, design

Descrizione generata automaticamenteAndroid'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:**
   * At the foundation of Android is the Linux Kernel. It provides core services such as hardware abstraction, memory management, 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):**
   * Above the Linux Kernel is the Hardware Abstraction Layer (HAL), which provides 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.
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 JVM and runtime compilation)
   * ART 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 allows Ahead-of-time (AOT) compilation, good garbage collection and sampling in debugging.
   * It’s based on the Core Libraries, which provide the main features for executing an application and then the ART (Android Runtime) itself to run the app (this one was introduced in recent Android Versions, because at the birth of Android, there was no such a thing; this bases itself on the Dalvik Virtual Machine, to have it running quickly valuing performance over the Java Virtual Machine)
   * All the processes in Android are completely isolated, unless they communicate with the bridged communication offered by the OS itself.
4. **Native Libraries:**
   * Native libraries 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):**
   * The Application Framework layer provides developers with a set of high-level APIs and tools to build Android applications.
   * It includes various managers and services, such as:
     1. 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.
  + 1. 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.
* Other managers include the Intent resolver, allowing to request actions and data, the View System, which manages and renders UI components, the Package Manager, managing the installation and permissions of apps, the Notification one for events and the Resource one.

1. **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: Services 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. For example, an app might have a broadcast receiver that responds to incoming text messages.
  + Content Providers: Content providers 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.

This file is the first that is parsed and allows retrieving all the app’s info. Here's an overview of the key elements specified in the AndroidManifest file:

1. **Package Name:**
   * The package name is a unique identifier for the app. It is defined using the **package** attribute in the **<manifest>** element, such as **com.example.myapp**. We can’t have then apps with the same package name on Play Store. It can’t be updated afterwards as name. In this case you must update the whole application.
   * Package names are crucial for distinguishing apps and ensuring they don't conflict with each other on a device. From the signature of the apps themselves, we can distinguish between the versions of apps, whether they are updated or old, given it will be the same package name by the same developer.
   * They often follow a reverse domain naming convention, such as **com.facebook.katana**
2. **Components:**
   * The AndroidManifest file defines the app's components, including:
     + **Activities:** These are the app's user interface screens. Each activity specifies its name, intent filters, and which activity should launch when the app is started.
     + **Services:** Background processes or tasks that run independently of the user interface.
     + **Broadcast Receivers:** Components that listen for system or app-specific events and respond accordingly.
     + **Content Providers:** Components that manage access to app-specific data or shared data with other apps.
3. **Permissions:**
   * The AndroidManifest file lists the permissions the app requires to access specific device resources or perform certain actions.
   * Permissions include access to the camera, location, contacts, internet, and more.
   * When users install an app, they are informed about the requested permissions and can choose to grant or deny them.
4. **Intent Filters:**
   * Intent filters define how an app responds to external events or user actions. They specify which components should handle specific intents (e.g., opening a web link or receiving a text message).
   * Intent filters allow apps to interact with each other and with the system.

The AndroidManifest file is essential for both the Android operating system and the Google Play Store. It helps the system understand how to manage and execute the app and ensures that the app complies with security and compatibility requirements. It also plays a role in app distribution on the Google Play Store by providing necessary information for listing and categorizing the app.

Unlike traditional console-based programs, Android apps do not have a central "*main*" function. Instead, they consist of various components, and the Android operating system 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, and more.

When you click an app icon in the launcher, we link the MainActivity of an application, then using its UI and its components. We have no command line for input, only in a debugging phase.

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:

* **Registering a Listener:** Developers register event listeners (also known as event handlers or callbacks) for specific UI elements or system events. For example, you might register a click listener for a Button.
* **Callback Invocation:** When the associated event occurs (e.g., a button is clicked), the registered callback is invoked, allowing the app to respond to the event. This is where developers write the code to handle 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.

We can define a main one, but we also do 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. 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).

Below we have the Android Activity Lifecycle:

Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamente

To be complete, each one will be briefly described above (not required, but just to give more notions):

1. **onCreate()**
   * This method is called when the activity is first created.
   * You should perform one-time initialization here, such as setting up the user interface and initializing variables.
   * The activity is not visible to the user at this stage.
2. **onStart()**
   * Called when the activity becomes visible to the user but is not yet in the foreground (i.e., not yet interacting with the user).
   * This is a good place to start or resume background threads or services that should be active when the activity is visible.
3. **onResume()**
   * This method is called when the activity comes into the foreground and becomes interactive.
   * It's a suitable place to start animations, acquire resources, and register listeners for user input.
4. **onPause()**
   * Called when the activity is about to lose focus but is still partially visible.
   * You should release resources that are no longer needed, stop animations, and unregister listeners to conserve system resources.
5. **onStop()**
   * Called when the activity is no longer visible to the user.
   * You can release resources and perform cleanup operations that are not needed when the activity is not visible.
6. **onRestart()**
   * This method is called when the activity is restarting after being stopped but not destroyed.
   * It provides an opportunity to reinitialize components that were stopped in **onStop()**.
7. **onDestroy()**
   * Called when the activity is being destroyed or is about to be removed from memory.
   * You should perform final cleanup here, release resources, and unregister any remaining listeners.
8. **onSaveInstanceState(Bundle outState)**
   * Called when the activity is about to be destroyed but might be recreated in the future.
   * You can save critical data to the **outState** bundle to restore the activity's state when it is recreated.
9. **onRestoreInstanceState(Bundle savedInstanceState)**
   * Called after **onCreate()** if the activity is being recreated from a previously saved state.
   * You can retrieve the saved data from the **savedInstanceState** bundle to restore the activity's state.

A Service in Android is a component that performs tasks in the background, independently of the user interface, and often for an extended period. Services are used when you need to execute operations that should continue running, even if the user switches between different activities or exits the app altogether.

Services 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 are designed to respond to events such as incoming SMS messages, battery state changes, screen on/off events, network connectivity changes, and more. They serve as event handlers for system-level and app-specific events.

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

Inter-Process Communication (IPC) mechanisms built on top of the Binder component in Android, which is a core component allowing communication between different processes overall managed by the Android OS itself. It's used to pass data and messages between components running in separate processes on an Android device. The following IPC mechanisms are built on top of the Binder:

1. **Intents:**
   * Intents are a messaging mechanism in Android used for communication between components, whether they are in the same app or different apps and deliver commands and data to components.
   * Intents can carry both commands and eventually data, making them versatile for various communication scenarios between components. Usually the receiver replies back, and they’re not really designed for communication
2. **Messengers:**
   * Messengers are objects that support message-based communication between processes.
   * They are built on top of Binder and provide a higher-level interface for sending and receiving messages.
   * Messengers are commonly used when an app wants to perform actions or share data with a service running in a separate process.
3. **Content Providers:**
   * As mentioned earlier, Content Providers are components that expose a cross-process data management interface.
   * They use the Binder for IPC to allow other apps or components to query and modify data.
   * Apps can access and manipulate data from a Content Provider in a separate process by sending queries and requests via the Binder.
4. **AIDL (Android Interface Definition Language):**
   * AIDL is a language used to define interfaces for communication between Android components running in different processes (creating a client-server communication). Its files define the interface methods and data types that can be used across processes
   * It enables a client (usually in one process) to call methods on a remote object (in another process) as if it were a local object.
   * Android's IPC mechanism uses AIDL to generate code that handles the low-level Binder communication, making it easier for developers to implement cross-process communication.

We can have many use cases: communications between the same application, having care to specify which app this belongs to.

1. **Starting Another Activity in the Same App:**
   * Use Case: An activity within the same app (A) wants to start another activity (A.Y) within the same app.
   * Example: A login activity (A.Login) wants to start a user profile activity (A.Profile) after successful authentication.
2. **Starting an Activity in Another App:**
   * Use Case: An activity within one app (A.X) wants to start an activity in another app (B.Z).
   * Example: A music player app (A.Player) wants to open a specific song (B.Song) in a third-party music streaming app (B.Music).
3. **Sending Data Between Components:**
   * Use Case: A component (e.g., an activity, service, or broadcast receiver) within one app (A.X) wants to send data to a component in another app (B.Z).
   * Example: A messaging app (A.Chat) wants to send a chat message to a recipient's messaging app (B.Chat).

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.
  + *Any kind of event is anyway handled by the Android OS*, not needing the sender itself.
  + You can see this example, where there is no ambiguity because everything is explicitly told

Immagine che contiene testo, schermata, Carattere, bianco

Descrizione generata automaticamente

* 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").
  + You can see this example

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

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.

It is 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.

Now let’s specify about the Android Security Model, composed by:

**1. Sandbox Model:**

* Android follows a sandboxed security model, where 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). Also, an app can have multiple GIDs (Group ID).
* 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:**

* Android uses a permission-based system 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 app's manifest 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
* Users can grant or deny permissions on a per-app basis, giving them control over how apps access their device's resources.

**3. App Signature:**

* Android apps are signed with digital certificates to verify their authenticity and integrity. 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

**4. SELinux (Security-Enhanced Linux):**

* Android incorporates SELinux, 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
* 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
* It adds an extra layer of security by enforcing access control rules even if an app's permissions are compromised. Enforcing mode is applied to system daemons, while permissive mode is applied to apps

**5. Verified Boot:**

* Verified Boot is a security feature that ensures the integrity of the Android operating system and prevents booting compromised or tampered images. This 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
* 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

## Questionnaire 1 – Lecture 2

Immagine che contiene testo, Carattere, schermata, algebra

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

## Challenge 1 – filehasher

Useful general commands:

*adb logcat*

*adb logcat -s MOBIOTSEC* (for challenges)

1) filehasher

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:  
  
*// 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:

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

<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
* 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

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*

Things to know for the challenge (presentation by Alberto Lazari on the solution):

* Messages/Compose/Settings Activity
* Intents to communicate between activities
* Implicit intents specific actions for apps installed or user installed

# Lecture 3 – RPC/IPC Communication, Binder, Android permissions

From the eyes of an application:

* Android is based on Linux
* Each app has its own Linux user ID
  + It shares many features, but a unique one is the user identifier (with no mapping between the physical user and the user identifier, like it happens in Linux)
* Each app lives in its own security sandbox
  + Standard Linux process isolation (for permissions)
  + Restricted file system permissions

To the new application just installed, will be associated with its own UID (also the number of it has a specific meaning; over 1000 is for system apps, over 10000 is for third-party apps). Also, every app has a private directory (*/system* for system apps, */data* for user-installed apps), called “internal storage”, having no ways for other apps to access it). There’s also a particular feature called *sharedUserId*, which allows apps to share the user ID, possible if they have the same signature (so, developed by the same people).

As soon as an app gets assigned a particular UID, it will be executed, coming with its own sandbox and its own process. We see the Android OS as a combination of different sandboxes, where each one will get assigned to a particular application. 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 then?

We have two levels of security inside the Android architecture:

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

* The vertical separation, called *security boundary*, is kind of practical and it’s automatical 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 the amount of memory dedicated to the user (coming with restrictions; more important action can be done with usage of signatures, given inside the kernel space).
* The component allowing such communication between spaces is the Binder. Let’s start with an example, like storing a file on the hard drive:
  + The process can’t access to the physical hard drive (would be too dangerous)
  + The process must ask the OS
  + The developer uses high level APIs, with languages like Kotlin/Java/C/C++, like this one:

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

* + Under the hood, again the process needs to ask the OS, and everything will be translated in a set of system calls (for example in Java: *Java 🡪 libc 🡪 syscalls),* like the ones below:

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

The component able to translate these calls from higher-level to architecture ones is the *Binder*, keeping in mind the differences between the usual ones (useful for reverse-engineering purposes or just to know what goes on under the hood):

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

Immagine che contiene testo, schermata, Carattere, menu

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere, nero

Descrizione generata automaticamenteOther references can be found here: <https://syscalls.w3challs.com/> or invoking the “man syscall” command, seeing the manual for architecture the registers or viceversa:

We rely on the Binder to not having to write everything manually, translating APIs to set of system calls. Ot all requests are as easy as opening a file, like accessing the current location, sending SMS, display in UI, playing sounds, talking to other apps, etc.

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

* App invokes Android API (still within the app’s sandbox, running on underprivileged process)
* Then, the service getting called use the privileged API at the system level

The Binder allow for:

* Remote Procedure Call (RPC)
* Inter-Process Communication (IPC)

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:

Immagine che contiene testo, schermata, diagramma, linea

Descrizione generata automaticamente

This way this interaction is allowed by the AIDL language (*Proxy/LocationManagerService* running in the user space, *Stub/LocationManager* running inside the non-privileged space). As soon as the API gets called, the Binder intersects and forwards the location to the privileged process. As soon as it’s requested, it will return the right location value, going bac to the client component.

Everything is done by the Android OS, however it’s important what goes on under the hood. In this case, the Proxy and Stub are automatically generated starting from AIDL, where the Binder does multi-purpose syscalls to talk to drivers and its driver dispatches messages and returns replies. We have a lot of Managers (Activity/Package/Telephony/Resource/Location/Notification/Resource) to deal with).

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 🡪 B.X):

Immagine che contiene testo, schermata, diagramma, Carattere

Descrizione generata automaticamente

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:

Immagine che contiene testo, diagramma, schermata, linea

Descrizione generata automaticamente

Can an app always do these things? No, it can’t.

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

The Android framework defines a long list of permissions and an application can request it according to the security-sensitive level (sensitive = open Internet connection/send SMS/accessing location/user contacts, etc.), mapping for each application the right resources.

Right here, there are many examples of permissions:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

There are many Permission Protection Levels:

* 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
  + As said, location, contacts, etc. These permissions require manual permission from the user
* Signature
  + Basically a way for two different apps to declare a permission and made available for the two apps (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: If device's API level >=23 (Android 6) AND app's targetSdkVersion >= 23:

* 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:

Immagine che contiene testo, schermata, biglietto da visita, design

Descrizione generata automaticamente

Instead, If device's API level <23 OR app's targetSdkVersion < 23:

* 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:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

We then have runtime vs install-time prompts:

* Runtime
  + Pros: Users can install apps without giving all permissions
  + Pros: Users have contextual information to decide accept/reject
  + Pros: 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
  + Cons: No contextual info to take informed decisions

Usually, permissions are organized in groups and organized at a group level (e.g. User grants X -> all permissions in X's group are automatically granted if an app's update asks for them).

The following are an example:

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente

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:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Apps can also define custom permissions, like:

Immagine che contiene testo, Carattere, bianco, schermata

Descrizione generata automaticamente

System permissions are defined the same way; 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:

Immagine che contiene testo, schermata, Carattere, algebra

Descrizione generata automaticamente

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

## 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

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

b. is exported by default if it is an activity

c. 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)?

a. Android app permissions and GIDs are unrelated; they serve different security purposes within the Android ecosystem.

b. Each Android permission corresponds to a unique GID, allowing apps with specific permissions to access resources and services within their associated GID.

c. 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.

## 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*](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>  
        <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:

lass MainActivity : ComponentActivity() {  
  
 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)*

Recostructing it in order:

*> `FLAG{Gutta\_cavat\_lapidem\_non\_vi\_sed\_saepe\_cadendo}`*