0 - Mobile Android

Android System Overview

Android is based on linux (modified) and made better from SELinux (that does some checks on the access to apps, processes and filesystem).

Android fragmentation

Android systems **are often customized by Original Equipment Manufactures (OEMs)** with for example:

- apps that cannot be removed
- very slow adoption of system updates

With the project Treble we have a better separation between the Android framework and the OEM code in order to speed up the updates (from android 9)

Google Play protect certification

Vendors can partner with google in order to preinstall google apps and certify that they have recent security updates.

- **it includes automated software testing and virus scanning**

Android versioning

Each device runs one Android framework version that **is identified by the API level**

- Android 14 == API/(SDK) level 34

A developer can specify the minimum and the target SDK:

- `< uses-sdk android:minSdkVersion="15" android:targetSdkVersion="26" />`

For example new and updated apps on google play need to target Android 12 or higher.

Android Package Format (APK)

An application is a all-in-one application archive, that is basically **a zip file**.

It is usually not encrypted, except for paid apps on the Play Store.
Unencrypted apks can be dumped from a rooted phone.

They can be installed using google play or "sideloaded" via USB or from the archive of the phone.

APK identification

Each app has a **package name (ID) that is theorically unique**. It is used to:

- identify the app on google Play store
- avoid to install 2 apps with the same package name on the same device

Each app **has a developer signature. This signature is a self-signed certificate** and is used to:

- distinguish developers and not to identify them
- distinguish system apps from normal apps
- used to verify the apps updates (posso aggiornare un app solo se l'apk dell'aggiornamento ha la stessa developer signature)
- nowadays Google Play signs the APKs for the developers

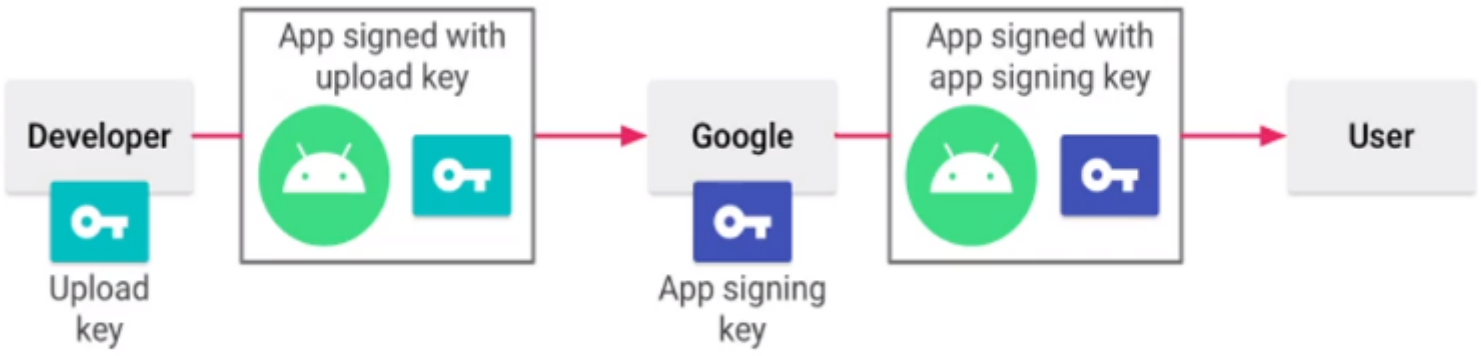


Figure 1. Signing an app with Play App Signing

APK Content

An APK contains:

- **AndroidManifest.xml**
  - used to declare the name, version, permissions, components ecc of the app (usually in binary XML)
- **classes.dex**
  - the classes of the code compiled in dex file format
- **resources.arsc**
  - precompiled resources, such as a binary xml
- **META-INF/**
  - **MANIFEST.MF** that is the manifest file
  - **CERT.RSA** that is the certificate of the app (RSA)
  - **CERT.SF** that is the list of resources and SHA-1 digest
- **lib/**
  - libraries and pre compiled code
- **res/**
  - resources not compiled in resources.arsc
- **assets/**
  - applications assets retrievable with AssetManager (immagini ecc)

Android app permissions

The android app permission are used to explicitly say what are the application permission over the device.

They **are specified in AndroidManifest.xml**:

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

There are 3 types of permissions:

- **normal permissions** that are granted at install time and no confirmation is needed
  - `INTERNET, NFC, BLUETOOTH..`
- **dangerous permissions** that need explicit user approval at install-time or runtime
  - `CONTACTS, LOCATION, STORAGE..`
- **special permission** that require manual activation through system settings
  - `REQUEST_INSTALL_PACKAGES, ACCESS_BACKGROUND_LOCATION ..`

We have also **permission groups**, in this case if the permission is grant then all the permission in the group are grant:

- `android.permission-group.MESSAGES = {android.permission.RECEIVE_SMS, android.permission.SEND_SMS, ..}`

We can also use **signature permissions** that are permission defined by an app A and that can be used also from other apps that share the app A certificate (esempio le posso usare con un ecosistema di app di uno sviluppatore Facebook-insta-whatsapp)

- `my.cool.app.START_FIREWORKS`

The **priviledged permissions** ==that are used for pre-installed apps in /{system, product, vendor}/priv-app.
==It is an allow-list for vendors to customize the system:

- `GET_ACCOUNTS_PRIVILEGED, CALL_PRIVILEGED`

Permission management

From Android 11 apps can get permission one time using "Only this time".

All the apps that are not used for few months reset their runtime permissions.

Denying a permission more than once means "don't ask again" and the system hides the permission request.

Android app Sandboxing

Kernel level application sandboxing

Each app runs with is own unique used ID (uid) and creates a new directory in `/data/data/< app id >` owned by an UID.

It was possible to share linux UID between apps from the same developer. But it is deprecated.
It is based on using arbitrary strings as "sharedUserId" in AndroidManifest.xml until API 29.
It is recommended nowadays the inter-process communication.

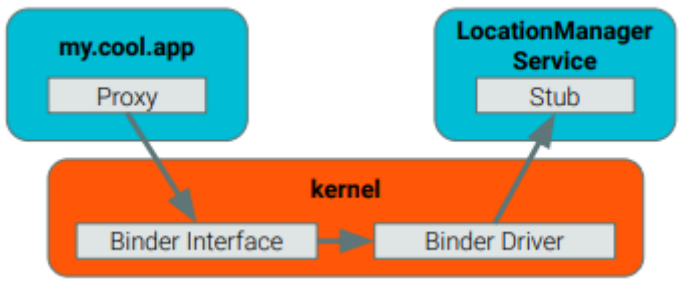
Different SELinux Sandbox levels

We can explicit the level with `targetSandboxVersion [1,2]` in AndroidManifest.xml.
Level 2 is default since API 28 and disallows UID sharing and cleartext traffic.

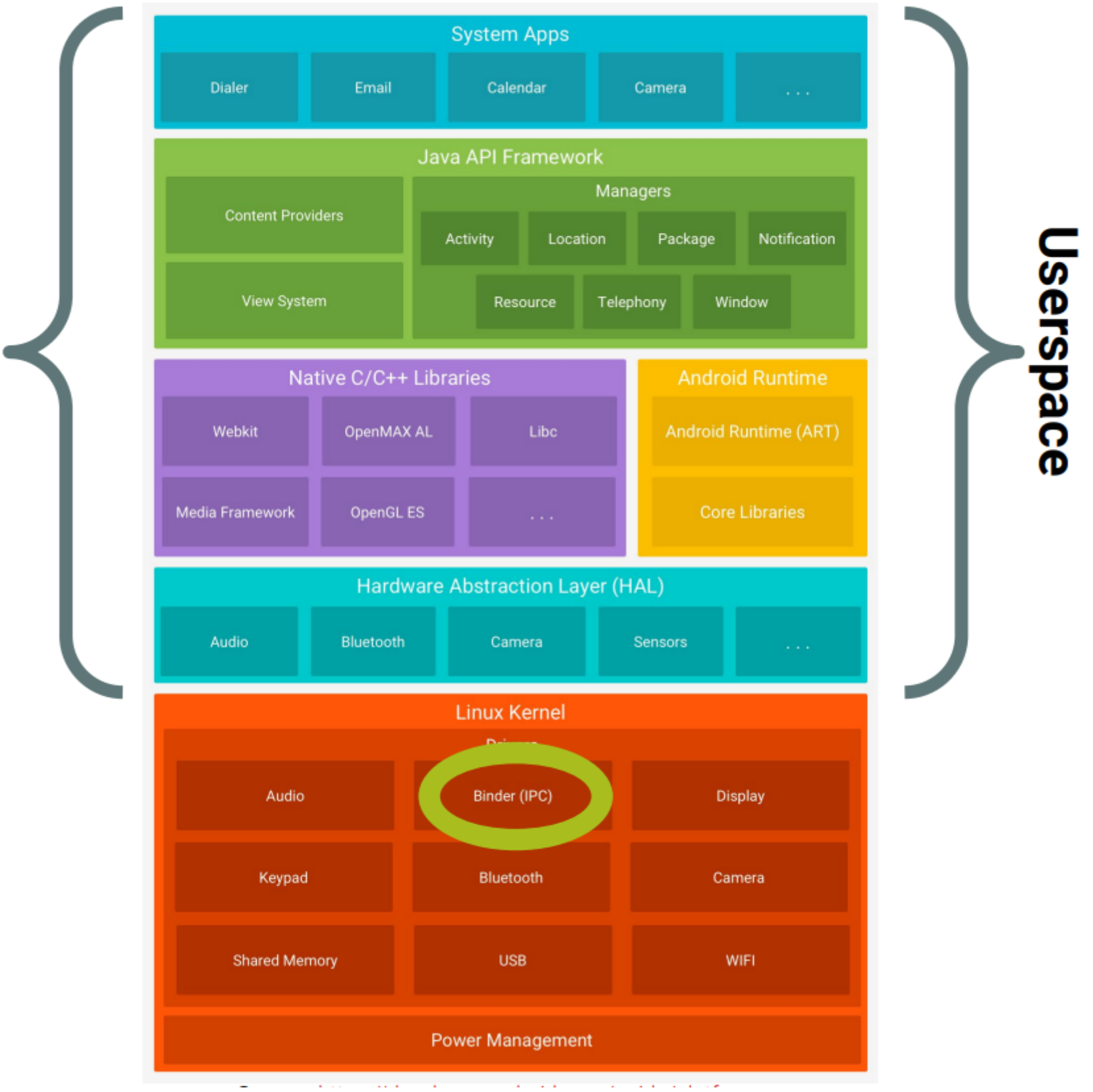
It implements MAC (Mandatory Access Control).
It is not possible to write on `/system` or to have row access on GPS/microphone/radui an is not possible to disabel SELinux.

Platform architecture

Apps use Binders to communicate with other (system) apps passing objects and file descriptors.



The Kernel passes UID and the caller can check the permissions.



Android intents

The android intents **are used to pass messages between the components of the apps** (example launch an activity, start a service or send a broadcast message).

They are based on **target (optional), action and data**

An app can **intercept those intents using an Intent Filter** (esempio se faccio share di una foto vedo che Google Drive risponde e mi esce tra le appe su cui posso condividere)

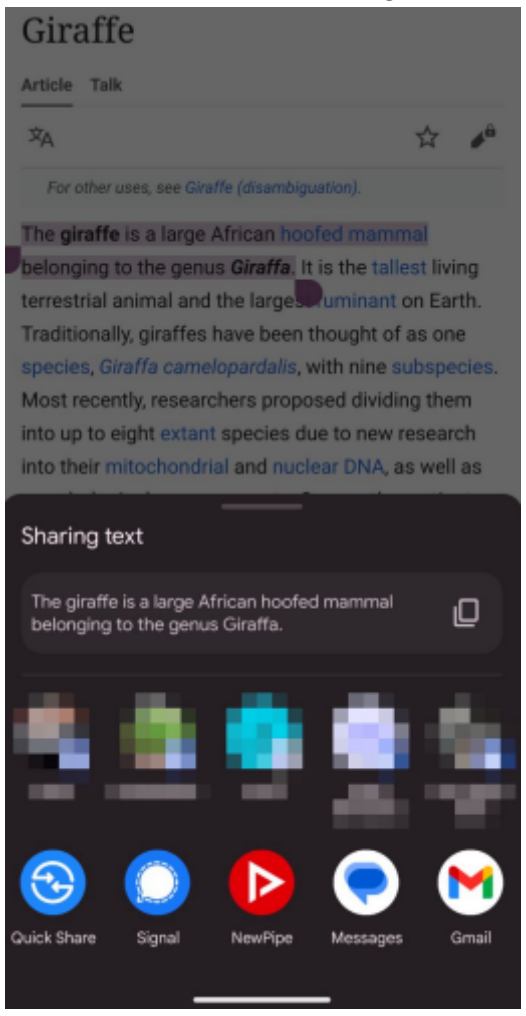
```
<activity android:name="ShareActivity" android:exported="false">
        <intent-filter>
                <action android:name="android.intent.action.SEND"/>
                <category android:name="android.intent.category.DEFAULT"/>
                <data android:mimeType="text/plain"/>
        </intent-filter>
</activity>
```

Implicit intent

We just specify the action and optional some data.

- example `Intent.ACTION_SEND, Intent.ACTION_VIEW ..`

The OS will choose the target or will allow the user to choose the app that can handle the intent.



Explicit intent

We specify the target component to talk to.

APP components

Activities

The **activity is the entry point for and user interaction, basically a single screen with a GUI**.

They are composed from Views and code.

The **Main activity defined in AndroidManifest is launched on app start**.

External apps can start arbitrary activities of the app A if the app A is defined as `exported`

- it is by default false but if the Activity defines an intent filter the it becames exported.

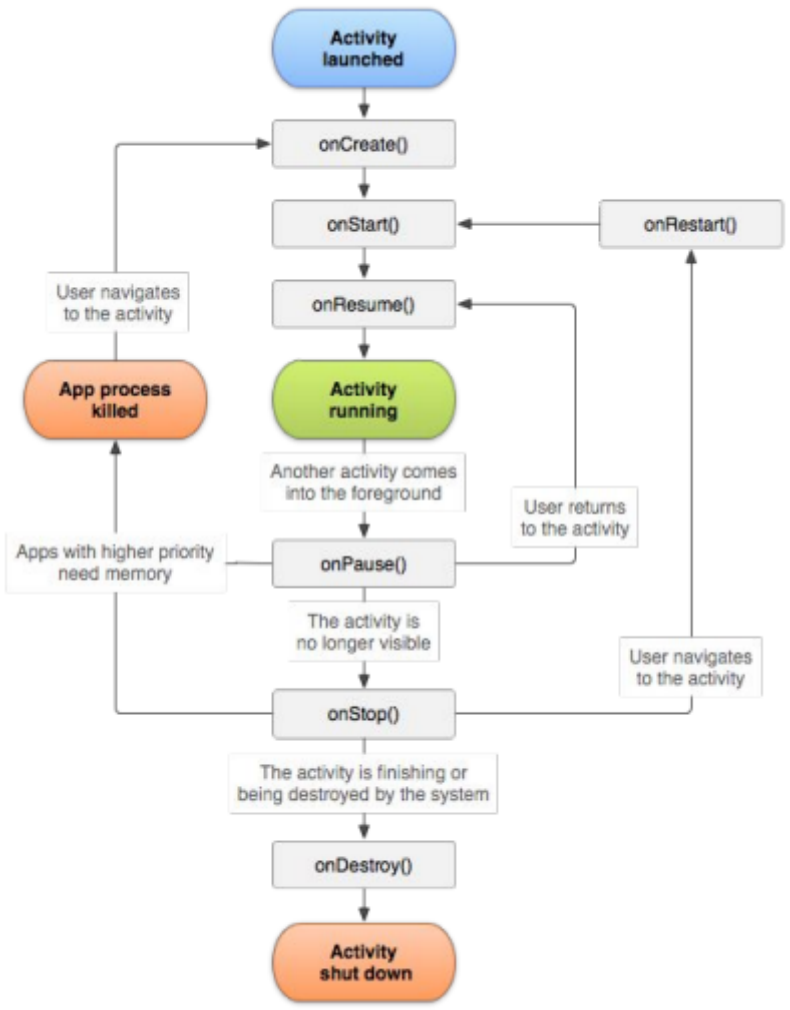
```
<application ...>

        <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=“.GalleryActivity" android:exported="true" …>
        </activity>

</application>
```

Each activity has a life cycle:



Content Providers

It is used to **manage the access on the app data from other external apps**
The access can be protected by permissions.

It is mostly based on (fiel-based) SQLite database, but this can lead to SQLinjections.

```
<application ...>
        <provider android:name=".MyContentProvider"
        android:readPermission="my.cool.app.READ_POEMS"...>
        </provider>
</application ...>
```

Broadcast receivers

It is used to respond to system-wide events in publish/subscribe way

- for example SMS received, phone boot, network state changed

It is another entry point to apps because it is delivered also when the app is not running.
It can be declared statically (in AndroidManifest.xml) for explicit broadcasts or also registered dynamically at run time.

```
<application …>
         <receiver android:name=".OnBootReceiver" android:exported="true">
                 <intent-filter>
                         <action android:name="android.intent.action.BOOT_COMPLETED"/>
                 </intent-filter>
         </receiver>
</application>
```

Services

Services are used to perform ==**long-running operations in the background without GUI** ==(network, music player ...)

```
<application ...>
        <service android:name=".BackgroundService" …> </service>
</application>
```

Foreground services

They are visible by user with a notification (music player) and are running also when the user switches to a different app

Background services

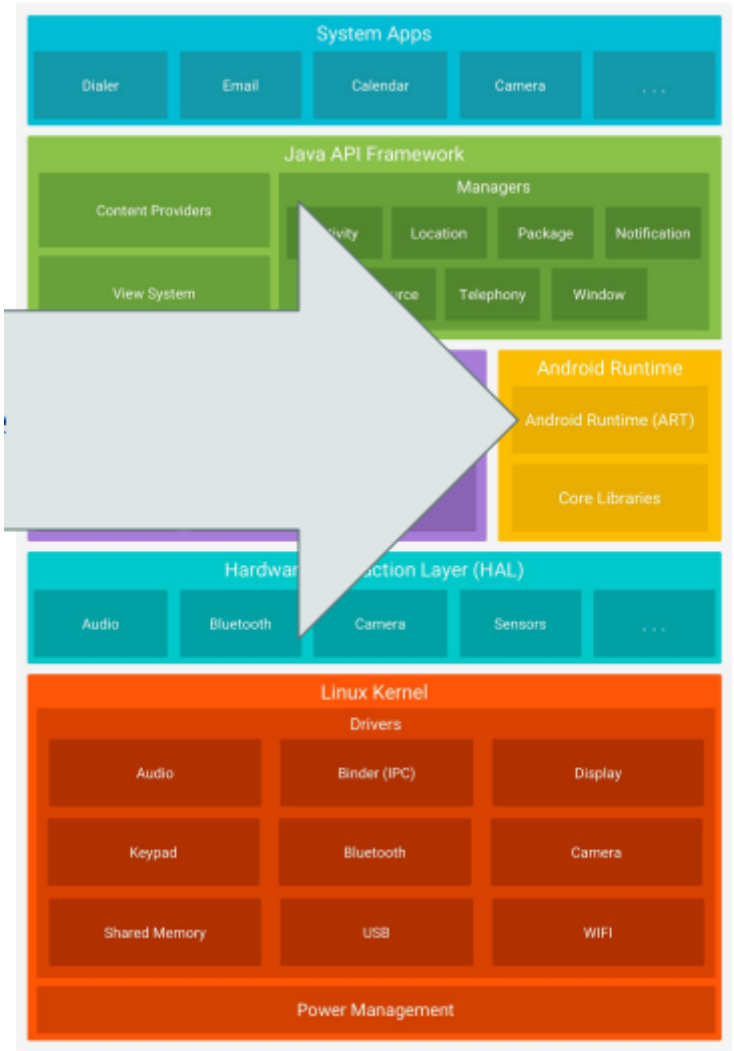
They aren't visible and have a limited functionality with API level > 26 (Oreo 8.0)

Bound services

They are an alternative to broadcasts and are based on a client-server architecture.

Running apps

Running apps are managed bu the Android Runtime (ART)



JAVA but not Java VM

The android source code is written in java or kotlin but we don't run it with the Java VM

The Dalvik VM was the key element of the Android runtime:

- little memory, no swap and for slow CPUs
- the architecture is based on registers (java VM is based on stack)

The code is compiled in this way:

- **java source -> java bytecode (.class) -> Dalvik (.dex)**

On boot there is a process called Zygote that loads the framework and the runtime:

- when an app is started the Zygote is forked (very performant)

Old Dalvik VM vs New Runtime (ART)

Dalvik

In dalvik VM:

- all **.dex files are generated by .class files**
- there is just-in-time **(JIT) compiler is used for performance** (compila il codice durante il runtime)
- we have also a **optimized “odex” Ahead-of-Time compilation** (compila il codice prima dell'avvio)
- we have **larger memory footprint**
- from android 5.0 not so used

Android Runtime (ART)

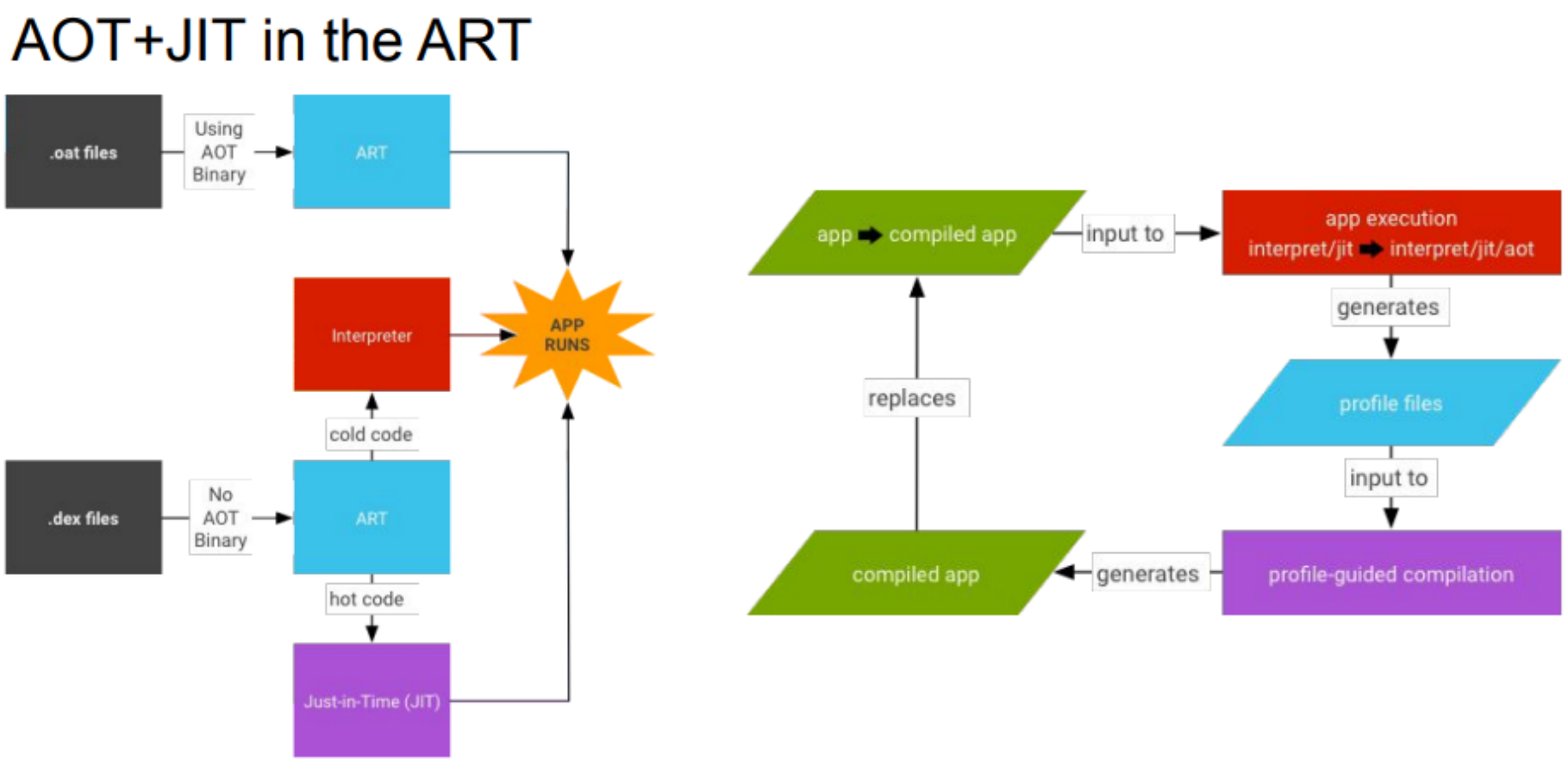
In ART we:

- still use **DEX bytecode format**
- have the **Ahead-of-time (AOT) compiler** (ovvero un compilatore che converte il codice prima dell'avvio)
- generate native **ELF (.oat file) on the app installation for all "hot methods"** (i metodi usati più frequentemente)
- a speed up because **it is optimized for the current hardware**
- could use also **JIT compiler or Interpreter**

ESAME:
The ART supports both ahead-of-time compilation and just-in-time compilation.
The ART runs code that was compiled from Java or Kotlin source code.
The ART is a register-based VM.

NON é VERO CHE: The ART requires developers to compile applications to native code (ELF).

Since Android 7.0 we can have AOT + JIT:



L'utilizzo di una combinazione di AOT e JIT offre diversi vantaggi.
L'AOT garantisce un'avvio più veloce dell'applicazione, poiché il codice nativo è già compilato e pronto per l'esecuzione.
Il JIT, d'altra parte, ottimizza il codice durante l'esecuzione, migliorando le prestazioni complessive dell'applicazione.

Native code

Native code is code written in C or C++ and that can be interpreted using the Java Native Interface (JNI).

Developers can implement their own native libraries with the Android NDK.

In general we have to avoid it because increases the app complexity.

However **it makes security analysis trickier**:

- everything runs in the same sandbox
- there are no barriers between java and native code
- Java and C/C++ can communicate via JNI
- It can be used to overwrite the ART in memory

WebViews (each app can be a browser)

We can use UI that displays local or remote html. Generally they are used to format text and load external web content.

It can be used to implement Cross Platform Apps like how it happens in Flutter.

Nowadays JS is disabled by default (to avoid XSS and so on)

KERNEL SECURITY

The kernel is secured by the the bootloader that is stored in the ROM.

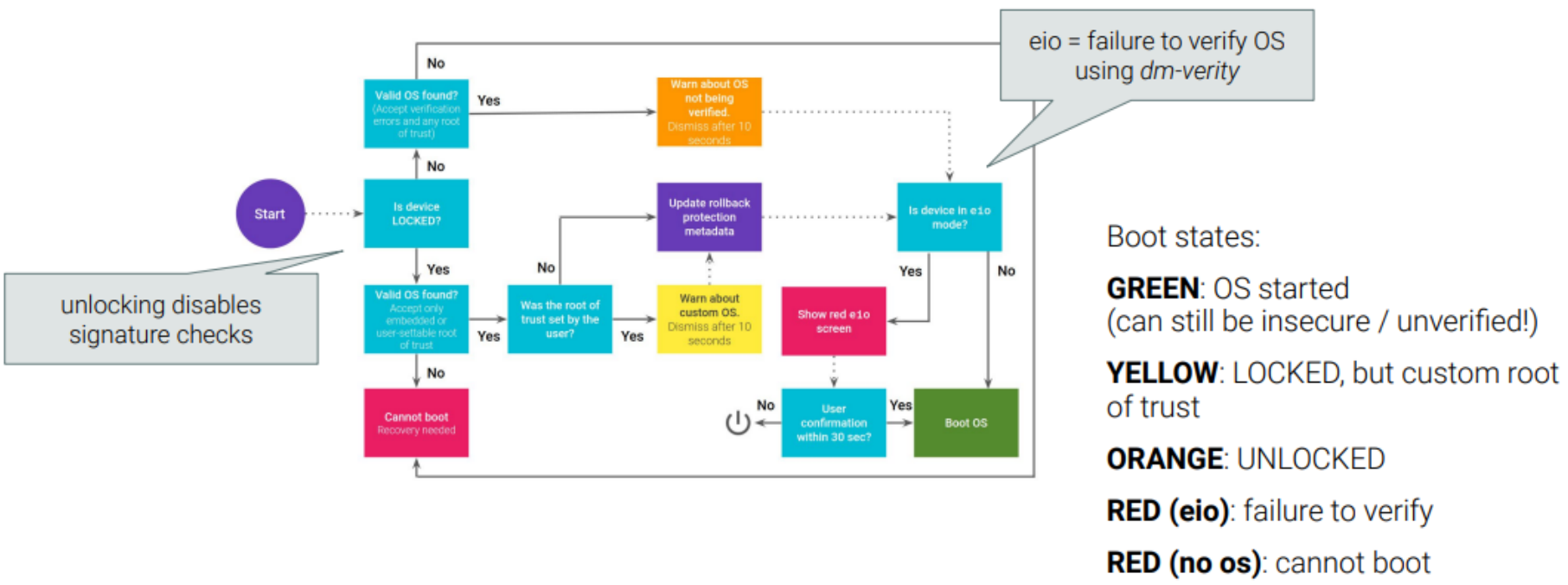
The device can be in 2 states:

- LOCKED
- UNLOCKED

When the state changes all user data is deleted.

The bootloader checks the device state and the OS signature using the dm-verity.

The dm-verity creates a cryptographic signature for the main system files during the OS installation or update. These signatures are then checked each time the device is opened.



Physical confidentiality

The idea here is that the **cryptographic keys and certificates are stored in special hardware**.

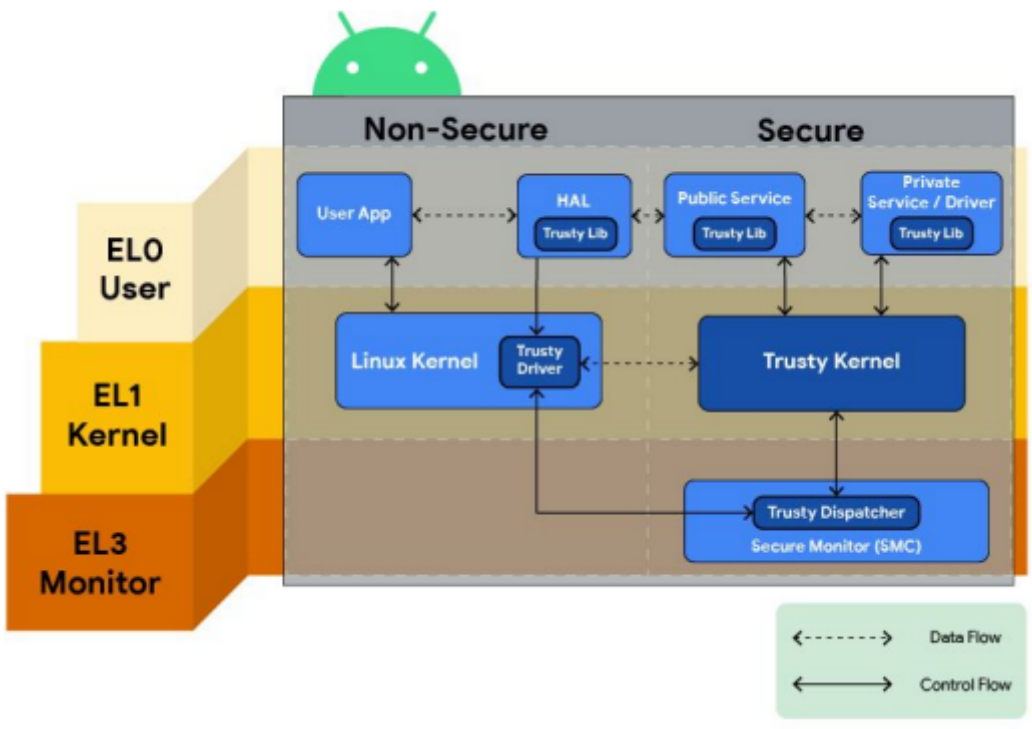
They are generally part of the main ARM CPU as **Trusted Execution Environment (TEE)** and has:

- a secure (virtual) CPU
- secure memory
- secure cryptographic processor
- secure random number generator

Trusty TEE

The secret keys are contained in the TEE and cannot leave it.

The idea is that the non-secure world sends data to encrypt or decrypt to the secure world.



Nowadays is difficult to develop secure services with Trusty.

App attestation

It is a system service that checks the APK signature with TEE component that is also used to check OS integrity:

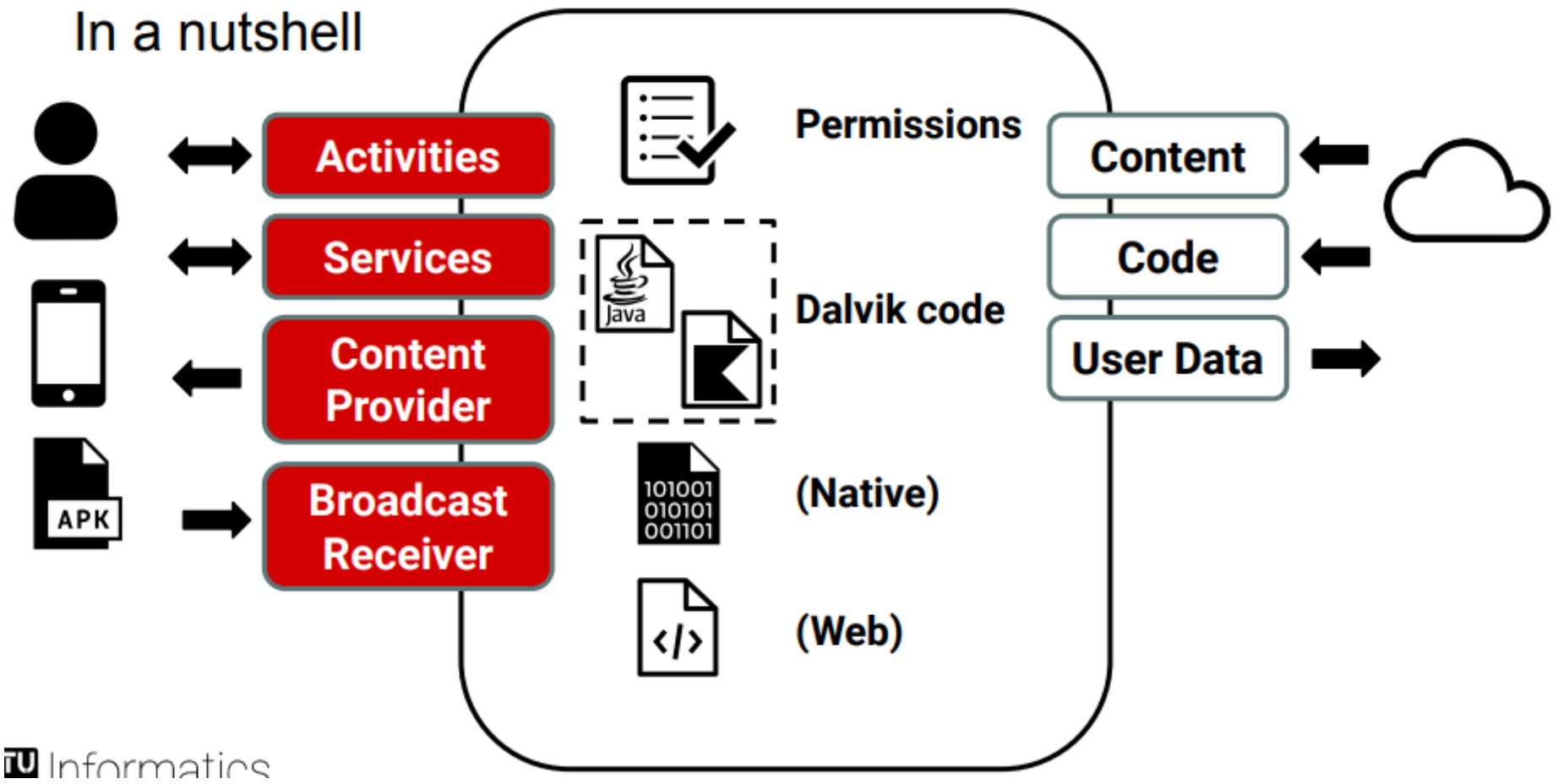
1. **server generates a challenge**
2. **the app requests TEE to sign the challenge**

1. TEE checks the integrity
2. TEE signs challenge, APK signature and boot state
3. **the app sends the signed challenge to the server**

It is generally implemented by Google Play Protect and needs Google Services.

What about fingerprints?

Is not possible for apps to collect fingerprints because the fingerprint sensor keeps the fingerprint model locally and only exposes functions through the Hardware Abstraction Layer (HAL).



OWASP TOP 10 MOBILE SECURITY ISSUES

M1 Improper Credential Usage

It involves:

- hardcoded credentials,
- insecure credential transmission,
- insecure storage...

The remedy is to avoid hardcoding them and handle them properly.

M2 Inadequate Supply Chain Security

It involves:

- relying on insecure third-party components,
- malicious insiders
- lack of security awareness

The remedy is to use:

- a secure software development lifecycle,
- a secure signing and distribution
- and alway keep up to date and to monitor vulnerabilities.

M3 Insecure Authentication/Authorization

It involves the using of:

- security by obscurity, hidden endpoints
- weak passwords, or storing them in cache
- single factor authentication, or ignoring it fro password reset

The remedy can be:

- assume that all APIs are public
- only allow strong passwords and don't store credentials in a weak way

M4 Insufficient Input/Output Validation

It involves loading serialized object without validate them but in general all kind of not validate input (SQLi, XSS, XXE, ecc)

The remedy are:

- always validate input
- check integrity of data

M5 Insecure Communication

It involves all stuff related to the communication like:

- ignoring TLS errors
- using SMS for multi factor authentication
- temporal communication and temporarely unencrypted data storing

Remedy:

- always use TLS and secure channels

M6 Inadequate Privacy Controls

It is related to laks of Personal Identifiable Information (PII) and the issues involve confidentiality, integrity and or availability.

Remedy:

- don't use at all PII (not sensitive ones) or delete them on request by the user or when they are not used.

M7 Insufficient Binary Protection

The idea is that the more popular an app is the more likely it is someone will reverse it.

Remedy:

- obfuscation can be very effective.
- Kerckhoffs principle: the security of a program should not rely on the security of the code

M8 Security Misconfiguration

It is related to all security misconfiguration such as:

- mantain a backend without authentication for testing and never turn it off
- request all permission than necessary

Remedy:

- check always the default values and if they are secure
- only request necessary permissions and release them when not used

M9 Insecure Data Storage

It involves all problem related to the data storage:

- not using built-in secure mechanisms to store sensitive data.

Remedy:

- limit logging of data
- check configurations of storage, encryption, access control

M10 Insufficient Cryptography

It is based on the cryptografy:

- deploying the crypto you rolled yourself
- using outdated libraries, setting, protocol versions

- not using salting for passwords
- transmitting keys in plaintext

Remedy:

- strong algorithms and models, long keys and trust implementation
- key management (HSM) and regular updates
- use key derivation functions (KDF) instead of hashing