Droid Development

Open Mobile Development (OMD)

November 18, 2022

Contents

Chapter 1

Overview

1.1 Main Motivation

Gain a sound understanding how *Android Apps* work. Then you are able to design, implement and analyse apps with regards to security.

Specific aspects of the Kotlin programming language can enhance code quality and efficiency of software development.

Get an idea of how to perform forensic analysis on Android. Selected tools are suggested to inspect the network, the system and apps. Risks are categorised, common vulnerabilities are explained, examples of attacks are given and mitigations are presented.

1.2 Main Contents

Techniques and technologies for app coding:

- The **Kotlin** programming language
- for secure programming,
- command line **tools** to automate tasks,
- the Android system architecture.

Techniques and technologies for app coding:

- **Forensic tools** to analyse the system,
- the apps, and
- the network.

1.2.1 Evaluate Your Know-how

Check out the possible exam questions.

Chapter 2

Part 1 – Kotlin

The Kotlin Programming Language.

2.1 Overview

The overall idea is to become a better programmer: to speak the same language, to know the main **concepts** of app programming for Android. This should help when **communicating** with other developers and it could help to reduce the amount of code to write (**efficiency**) and write more **secure code**.

2.1.1 Code Quality:

- Know your target group: **write code for humans!**
  - Other developers want to read, modify, and reuse your code.
- Stick to the Coding-Conventions
- Sensible naming (semantics!)
  - (+) speaking names for methods, variables
  - (-) avoid generic names such as `demo`, `test`, `dummy`
- Structure (indentation, line length)
  - suggestion for max. length of a method: about half to one page
- Documentation (if necessary)
  - (+) *What* (is it about)
  - (-) NOT: *How* each (line of) code works
- Reference external sources
  - Please honour the work of others.

2.2 How Kotlin is run

- Kotlin/JVM i.e. Java bytecode

- Kotlin/JS i.e. Browsers, Node.js
- Kotlin/Native i.e. Linux, Windows, macOS, iOS, Android, WebAssembly

2.3 From Java to Kotlin

The main differences listed in JAVA-vs-Kotlin:

- **Idiomatic Kotlin** means to let go old Java habits
  - (-) No *getters* and *setters*
  - (+) Use *properties*
  - (-) No multi-line code for *singletons*
  - (+) Use keyword *object*

2.4 Top Features Security

- Immutable and mutable variables
- Null safety, lateinit
- Contracts

2.5 Top Features Commodity

- **Immutable variables** <= whenever possible prefer `val` over `var` for security.
- **Type inference** <= compiler auto-detects type of a new variable
- **Default arguments** for functions <= call the same method with minimal information (what is different from the default)
- **Named arguments** <= so you need little documentation
- **High-order functions** <= pass functions like *normal* parameters
  - lambdas with `it` (vs. `this`)
- **Lazy properties** <= set on first use
- **Observable properties** (vetoable) <= event when current value is going to be changed
- **Data classes** <= no need to create boiler plate code for methods such as constructor, `hasChode` , `toString` , or `copy` .
- **Singletons** <= with keyword `object`
- **Operator overloading** <= for concise, readable code. E.g., use operator `+` instead of method `add` .
- Scope functions: `let` , `run` , `with` , `apply` , `also`

2.6 Concurrency

Check out the prerequisites[1]*, the know-how on processes and threads to be able to understand coroutines properly.*

The main idea is to **avoid** the so called **callback hell** and pretend/simulate a sequential (serial) program-flow. The code is shorter and better readable.

- **Coroutines** are not threads (threads are typically managed by the os), but just code which can be **suspended** and **resumed** later. Coroutines are *not preemptive*, they must (be nice i.e. collaborative and) pause/stop themselves.
  - (-) coroutines provide no real (preemtive) multithreading by default. See remarks below.
  - (+) coroutines are **very light-weight** and fast (no context switching necessary)
  - (+) can simplify asynchronous programming (no need of callbacks). This is called **structured concurrency**.

2.6.1 How to use Coroutines

- Android apps need import

```
import kotlinx.coroutines.*
```

- For using coroutines, we have to mark **suspendable functions** with `suspend`. Suspendable functions are later launched in a given scope within a coroutine *context* (see Scopes and Dispatchers below).

```
suspend fun downloadImages(fromURL:Url){
    ... // download takes a while
}
...
val f = suspend { ...}
...
withTimeout(1000) { ...}
...
```

- Coroutines need a **context** and a **scope**.

[1]**Prerequisites**: Knowledge about real (hardware) parallelism in contrast to quasi parallelism. **Multitasking** is managed by the operating system (keywords: processes, scheduler, dispatcher, context switch, preemtive, inter-process communication overhead). For concurrency within a process, we use **Multithreading** (keywords: threads, sharing the heap memory, race conditions, synchronisation, threadsafe, deadlocks, not termination from outside). Multithreading is faster and cheaper than multitasking, but not that secure. Multithreaded programming with nested threads might lead to **callback hell**.

– **Scopes** (to **launch async code**, i.e. to **launch suspendable functions**):
  * Use **Jobs** to manage (lifecycle of coroutines), i.e. to *launch*, to *cancel*, to *join*. . . * Note: For example, coroutines in Android **lifecycleScope** are canceled when Activity/Fragment is destroyed.

– Scopes: **GlobalScope** (live time of app) and optionally spedify **Dispatcher** : * `Dispatcher.Default` thread pool (typically the # of CPU cores), CPU bound workload * `Dispatcher.IO` pool 64 threads, I/O workloud: disk / network * `Dispatcher.Main` Android, update UI

– (Multiple) operations (coroutines) run within a scope. Optionally, we can block until all coroutines of a scope are done using `join` .
  * E.g.,

```
launch { ... }
GlobalScope.launch { ... }
...
GlobalScope.launch(Dispatchers.IO) { ... }
CoroutineScope(Dispatchers.IO).launch { ... } //
    ↪ same
...
val jb = GlobalScope.launch { ... }
jb.join()
jb.cancel()
...
GlobalScope.async { ... }
GlobalScope.async (start = CoroutineStart.LAZY) {
    ↪ ... }
...
// For Android Activities and Fragments
lifecycleScope.launch { ...  }
...
// e.g.:
job = GlobalScope.launch {
    downloadImages("https://server1.fhj.at/..")
    processImages()
    createThumbs()
    ... }
job.join()
```

– Optionally, coroutines can be started in a special **context** with one of following **CoroutineDispatchers**:
  * `Default` <= CPU intensive work

        * `IO` <= NW, FS, DB
        * `Main` <= use the main thread, e.g. because the UI must be updated!
        * `Unconfined` <= do not use
    - I.e. coroutine might be executed not in the main thread, but within thread pools.
    - Coroutines can easily be cancelled. E.g. be getting *out-of-scope* or manually with `job.cancel`.

- **Deferred objects**, i.e. Wait for results using `async` and `await` or `awaitAll`:

```
coroutineScope {
    val f = async { ... }
    val g = async { ... }
    ...
    g.await()
    f.await()
    ...
    val deferreds = listOf(
        async { f() },
        async { g() }
    )
    deferreds.awaitAll()
    ...
}
```

- Caveats
    - In coroutine use `delay()`, not `Thread.sleep()`
    - When using `async` we will explicitly `await` the result.
    - When using `launch` the *await* happens implicitly.
    - As coroutines can **only** be started within a **scope**, such as *runBlocking*, the statement `runBlocking{...}` is often used for scripts, but should be avoided in apps.
        * Details: the use of **runBlocking** in demos does NOT kill the script, i.e. to **wait until coroutines finish**. Careful: `runBlocking` **blocks the main thread**. Use inside `main` method. `runBlocking` returns the result of a coroutine.

```
runBlocking { ... }
...
runBlocking {
    ...
    coroutineScope {
        launch { ... }
        launch { ... }
    }
    ...
}
```

2.7 Other Language Features

- Inline functions
- Primary constructors
- Sealed classes, open classes
- Property delegation
- Generics with
  - invariance: ,
  - covariance: ,
  - contravariance:
- **Smart casts** <= compiler knows that within if block a otherwise nullable variable can not be null: `if (nullableVar is Type) {...`
- Range expression
- String interpolation, multi-line strings
- Conditional as expressions
- Powerful collections: filter, map, fold/reduce
- Extension to classes: extension functions

2.8 Not-so-nice/intuitive Features

- Companion objects
- Object expressions

2.9 Kotlin and Java

- Interoperability with Java
  - E.g. with annotations: @JvmStatic, @JvmField

2.10 Getting Started

- Checkout many, many **Kotlin Code Snippets** from https://git-iit.fh-joanneum.at/omd/code-snippets, i.e. a structured code snippet collection for multiple languages.
- https://kotlinlang.org

- LanguageDocumentation
- Koans in the online playground

Chapter 3

Part 2 – Tools

Tools for Android Development, Automation and for Android Forensics (Inspection)

Typical Tasks

- Create apps: compile, package, sign
- Reverse engineering: analyse, decompile

3.1 IDE - Android Studio

Android Studio is suggested. Get familiar with (USB-) **debugging**, inspecting the **LogCat** output and updating the Software Development Kits **SDKs** as well as the **emulators**, the Android Virtual Devices **AVDs**. Under the hood, Android Studio works with **gradle** as build tool to automate the compile, assemble, deploy tasks and more.

3.1.1 From any source code file

- Kotlin / Java / Bytecode Inspection
  - `Tools / Kotlin / Show Kotlin Bytecode`
  - Button **Decompile**

3.1.2 Inspect existing `*.apk` files

- Android Studio: Open apk using menu entry *Profile or debug apk*.

3.1.3 Deployment build

- With Android Studio: `View / Tool Window / Build`
  - create **apk for deployment**: `./gradlew assembleRelease`

- Inspect `./app/build/outputs/apk/release/app-release-unsigned.apk`
- Optionally, use `zipalign` and `apksigner` to sign it with YOUR developer key.

3.2 Command Line Tools (gradle, adb, am, pm)

Idea: we want to find out: which kind of development such as which programming language was used (java, kotlin, react/cordova), which libraries (3rd party, native libs: e.g. facebook integration) are in use, which sensors and permissions (e.g. camera, audio in manifest) are necessary and wheather the app is obfuscated (e.g. use of proguard)?

Advantage: later you might change code, revoke permissions, and resign an app.

3.2.1 Setup / check / inspect tools and paths.

Dependent of your system the commands might differ. For example, set and add the pathes to **android sdk**:

- for Windows:

```
PATH= %PATH%;C:\Users\User\AppData\Local\Android\Sdk\
    ↪ platform-tools
PATH= %PATH%;C:\Users\User\AppData\Local\Android\Sdk\
    ↪ emulator
PATH= %PATH%;C:\Users\User\AppData\Local\Android\Sdk\tools\
    ↪ bin
```

- for Linux and for Mac (and zsh):

```
export ANDROID_SDK_ROOT="~/Library/Android/sdk"
echo $ANDROID_SDK_ROOT

echo 'PATH=$PATH:'${ANDROID_SDK_ROOT}'/tools/bin/' >> ~/.
    ↪ zshrc
echo 'PATH=$PATH:'${ANDROID_SDK_ROOT}'/platform-tools/' >>
    ↪ ~/.zshrc
echo 'PATH=$PATH:'${ANDROID_SDK_ROOT}'/emulator' >> ~/.zshrc
source ~/.zshrc
```

3.2.2 Java for Gradle (and other tools)

- Gradle (and other tools such as *avdmanager*) need **Java 8** (Version 1.8). Install for example the JDK 8 from OpenJDK and set your `JAVA_HOME` environment variable, such as following example

- For Linux:

```
export JAVA_HOME='/usr/local/android-studio-ide
    ↪ -201.6858069-linux/android-studio/jre/jre'
export PATH=$JAVA_HOME/bin:$PATH
```

- For Mac (and zsh):
  - *Note for Mac-Users*: One can list all the installed jdks with the command `/usr/libexec/java_home -V`.

```
export JAVA_HOME="~/Library/Java/JavaVirtualMachines/
    ↪ corretto-1.8.0_302/Contents/Home"
```

- Try out **gradlew**: in the base directory of Android apps (created with Android Studio) check out available Gradle tasks:

  `./gradlew tasks`.

3.3 Prepare your Android phone for development

For development it is much better to use a real device.

- Step-by-step
  - (1) Add **Developer Menu Entry** `{} Developer options`
    * tap 7 times on `Settings / About phone / Build number`
  - (2) Switch on **USB Debugging**
    * Enable `Settings / System / Advanced / Developer options / USB debugging` (*Debug mode when USB is connected*)
  - (3) Connect via USB-Cable
    * try (Find details below)

```
adb devices
```

you get something, like following two devices listed, if a real device is connected and an emulator is running too:

```
List of devices attached
99051FFBZ00D39  device
emulator-5554   device
```

3.4 Main Tool for remote control the device / emulator

Android Debugging Bridge **adb**:

```
adb version
```

For several tools Java 1.8 is required. Find out which path tho Java Android Studio is using and set environment variable `JAVA_HOME`.

3.5 Source Code Checks

- detekt

  ```
  java -jar detekt-cli/build/libs/detekt-cli-1.4.0-all.jar
      ↪  --input <afile.kt>
  ```

- ktlint

  ```
  ./ktlint <afile.kt>
  ```

3.6 Code Optimisation

Code should be shrinked/minified and optimised. It can be obfuscated. The task is performed by the **R8 compiler** and is enabled by default for release builds.

3.6.1 Minify code and shrink resoures

In file `build.gradle` for release build the minifying is enabled by default:

```
...
buildTypes {
        release {
            minifyEnabled true
            shrinkResources true
            proguardFiles getDefaultProguardFile('proguard-
    ↪ android.txt'), 'proguard-rules.pro'
        }
    }
...
```

3.6.2 Custom Obfuscation

Basic obfuscation is switched on with the `minifyEnabled true` flag (see above). Customize obfuscation rules by setting *Progard rules* in file `proguard-rules.pro` :

Note: *The R8 compiler uses the proguard rules for configuring the obfuscation process.*

3.7 Compiling and package to apk

List all tasks available and start debug/release builds with Gradle:

```
./gradlew tasks --all

./gradlew :app:assembleDebug


/gradlew :app:assemble
ls -al ./app/build/outputs/apk/release/
```

3.8 Automate

Use tool to automate. The Android Bridge **adb** allows to remote control emulators and real devices. The package manager **pm** is used for installing apps and the activity manager **am** to startup (activities within) apps.

3.8.1 Create and run emulator

Full featured with Google Playstore:

```
echo no | avdmanager \
        create avd \
        --force \
        --name myCustomInspectionAVD \
        --abi google_apis_playstore/x86_64 \
        --package 'system-images;android-31;
    ↪ google_apis_playstore;x86_64'
```

Note: Google Play does not allow `adb root` .

Older API level and without Google Playstore:

```
echo no | avdmanager \
        create avd \
        --force \
        --name anotherCustomInspectionAVD \
    --package 'system-images;android-25;google_apis;
  ↪ x86_64'
```

Adjust to the api level of your choice, i.e. use -29 instead of -23 . . . . compare to previous created avds: `cat ~/.android/avd/*.ini && cat ~/.android/avd/*.avd/config.ini`

Start up an *android virtual device* AVD:

```
ls -al ~/.android/avd
emulator -avd "myCustomInspectionAVD"
```

3.8.2 Connection to Device / Emulator

- Find out, if a device/emulator is connected using the Android Debug Bridge `adb devices -l`
- and open a shell:
  - Default (works only If just one device/emulator is running): `adb shell`
  - Emulator TCP/IP: `adb -e shell`
  - Device over USB: `adb -d shell`
  - Specific device (here 94WAY0TG87): `adb -adb -s 94WAY0TG87 shell`
  - Specific device given transport id (here 2): `adb -t 2 shell`

3.8.3 Install / Uninstall Apps

Install on device

```
adb install -t app-debug.apk
```

For uninstall, we have to find out the package name

```
adb shell "pm list packages at.fhj"
adb uninstall at.fhj.iit.theshow
```

3.8.4 Run Apps

```
adb shell am start -n "at.fhj.iit.sunsetslideshow/at.fhj.iit
  ↪ .sunsetslideshow.MainActivity" -a android.intent.
  ↪ action.MAIN -c android.intent.category.LAUNCHER
```

3.8.5 Sending System Events, i.e. Implicit Intents to an App

First, connect via telnet to the device/emulator and authorize:

```
cat ~/.emulator_console_auth_token
# e.g.: FOoWnz55R8ge6Jf8

telnet localhost 5554
auth FOoWnz55R8ge6Jf8
```

then:

```
gsm call +438605987
sms send +438605987 "Hello SMS from telnet"

power ac on
power ac off

help
help network

network speed gsm
network speed 5g
```

3.8.6 Sending Events, i.e. Explicit Intents to an App

E.g. sending (**broadcast**) key value (`TAG` / `snow`) to an app (which has registered a broadcast receiver to listen for `at.fhj.slideshow.FILTER` .

```
adb -s emulator-5554  shell am broadcast -a "at.fhj.
    ↪ slideshow.FILTER" -e "TAG" "snow"
```

3.9 Mirror app to your desktop

Install and run mirroring tool on your desktop:

E.g. Download/install https://github.com/Genymobile/scrcpy.

```
scrcpy
```

On the device

```
adb -s 94WAY0TG87 shell am start ...
```

3.10 Inspect the Linux system (as root)

- Older systems (and emulator) support root access:

  `adb root` for android <= 4.2, AP level 17 Jelly Bean)

```
adb shell

whoami
id
ls -al /data
cat /proc/meminfo
cat /proc/version
ps -ef | wc -l
lsof
```

- Output the system events from the system log:

```
adb shell logcat
adb shell logcat | grep at.fhj
```

- use adb in combination with pm an am:

```
pm list libraries
```

- list and dump info on services

```
adb shell dumpsys
adb shell dumpsys -l # to list all services
adb shell dumpsys user
adb shell dumpsys display
adb shell dumpsys battery
adb shell dumpsys ....
adb shell dumpsys package com.android.settings
```

- Use your Linux know how to report:
  - number of ps, find user and process id, report cpu usage:
    * e.g. `ps -ef | wc -l`
    * e.g. `ps -ef | grep at.fhj`
    * e.g. `top`
  - report id, groups, . . . open files of your app process
    * e.g. `groups u0_a139`
    * e.g. `lsof`
  - optional report about loged in user and the system

    * e.g. `whoami` , `id` , `cat /proc/meminfo` , `cat /proc/version` , ...
    * report about app: fs permissions e.g. `ls -al`

- Inspect Security Enhanced Linux (SELinux):
  - Show SELinux labels (to inspect *user/role/type/range* - fields) for processes, files and directories:
    * e.g. `ps -eZ`

```
...
u:r:webview_zygote:s0          webview_zyg+
    ↪ 872     289 1772388  50824 0
    ↪      0 S webview_zygote
u:r:untrusted_app:s0:c116,c25+ u0_a116
    ↪ 7988     289 1274984 103476 0
    ↪       0 S com.android.chrome
...
```

    * e.g. `ls -dZ /storage/emulated/0/*`
    * e.g. `ls -Z /system/app/EasterEgg`
    * e.g. `ls -Z /system/app/EasterEgg/*.apk`

```
u:object_r:system_file:s0 /system/app/EasterEgg/
    ↪ EasterEgg.apk
```

  - Show SELinux labels set for specific file system pathes or for app-domains:
    * e.g. `cat /system/etc/selinux/plat_file_contexts | grep /mnt/sdcard`
    * e.g. `cat /system/etc/selinux/plat_seapp_contexts`

3.11 Inspection of app

Typical tasks, are to find out:

- About the UI
  - Languages used: German, English, Chinese, ...
  - Resources
    * strings
    * layouts
- About the code
  - Programming language: Kotlin?
  - was NDK (native development kit: C, C++) used
  - obfuscated / optimised <= classes named a b c
  - libraries / multidex
    * libs for analytics
    * logging libraries

- About the App Features
  - Manifest:
    - ∗ permissions(!)
    - ∗ activities (which might be started)
    - ∗ minSdkVersion <= can be run as root on virtual device?
    - ∗ Keys (API keys)
    - ∗ android:scheme <= might we use (crafted) "URL"s to start or to send (fake) data
    - ∗ android:allowBackup . . . ?
    - ∗ android:usesCleartextTraffic, ..network config, usw. usf <= security, client certificates, . . .
    - ∗ starts or uses services, broadcast receivers, content provider,. . . ?
- Network, Filesystem
  - uses SD-card
- Privacy, Security
  - hard coded credentials
  - signed by

3.11.1 Download an app (apk) and/or data

Downloading (**pull**) an *apk* after finding the full app name and location by listing and filtering existing apps (by their package name, i.e. the unique app-id).

```
adb -s 94WAY0TG87 shell pm list packages -f | grep fhj
adb -s 94WAY0TG87 pull /data/app/~~vR8Xxo4zn5qHtYeL-7Bm7g==/
    ↪ at.fhj.iit.sunsetslideshow-vRZW-hUzxpTO2VRLLUt21Q==/
    ↪ base.apk /tmp/inspectme.apk
```

```
unzip /tmp/inspectslideshow.apk -d /tmp/
```

Note: see below tool: `apkanalyzer`.

3.11.2 Inspect the backup

- Get app data via backup, then extract and inspect (as shown above):

  ```
  adb backup -apk -oob ....
  ```

  ```
  dd ...  |open ssl ... > backkup.tar
  ```

- A too simple approach is just to **unzip** the apk, because the Manifest is still in an unreadable binary format, the classes are compiled in unreadable *.dex files

```
# Too simple, just extracting the zipped (apk) file
unzip base.apk -d ./extracted/

# Sorry, Manifest still unreadable (compiled, binary):
tree ./extracted |grep  -3 "Manifest.xml"
cat ...Manifest.xml
```

3.11.3 Security-related tools (built-in)

In addition to normal linux tools (*zip*, *file*, *vi*, ...) further tools are recommended:

- **keytool**: find out if self-signed, if debug certificate was used

```
keytool -printcert -jarfile Snapchat_v10.70.0.0_apkpure.
    ↪ com.apk
```

- **jarsigner**: find out who signed the apk

```
jarsigner -verify -verbose -certs Snapchat_v10.70.0.0
    ↪ _apkpure.com.apk | grep CN= | uniq
```

- **apksigner**: find out who signed the apk

```
apksigner verify -print-certs Pocket\ Comics\ Premium\
    ↪ Webtoon_v1.3.3_apkpure.com.apk
```

- **apkanalyser**

```
apkanalyzer files list app-debug.apk
apkanalyzer files ... app-debug.apk
apkanalyzer apk features app-debug.apk
apkanalyzer apk ... app-debug.apk
apkanalyzer manifest permissions app-debug.apk
apkanalyzer manifest version-name app-debug.apk
apkanalyzer manifest ... app-debug.apk
apkanalyzer dex list app-debug.apk
apkanalyzer dex ... app-debug.apk
```

- **aapt2** tool (located in `build-tool` subdir of the android-sdk) makes it easy to view permissions, strings and more:

```
${ANDROID_SDK_ROOT}/build-tools/31.0.0/aapt2 -h

# For example, inspect strings
```

```
${ANDROID_SDK_ROOT}/build-tools/31.0.0/aapt2 dump
    ↪ strings base.apk
${ANDROID_SDK_ROOT}/build-tools/29.0.2/aapt dump xmltree
    ↪  app-debug.apk AndroidManifest.xml``
```

3.12 Selected security-related tools (3rd party)

3.12.1 Static Analysis

- Drag and drop `*.apk` files to Android Studio for inspection.
- **apktool**: reverse engineer Android apps with https://tools.kali.org/reverse-engineering/apktool, e.g. decompile *.dex.

```
apktool -o extracted-apktool d base.apk

# now inspect:
tree extracted-apktool/smali*
cat extracted-apktool/smali_classes2/at/fhj/omd/android/
    ↪ slideshow08b/R.smali
cat extracted-apktool/smali_classes4/at/fhj/omd/android/
    ↪ slideshow08b/model/GPS.smali
cat extracted-apktool/AndroidManifest.xml
```

- **jadx**: dex to java decompiler https://github.com/skylot/jadx

```
jadx -v -r -d extracted-jadx -ds extracted-jadx-sources
    ↪ base.apk

# now inspect the sources
tree extracted-jadx-sources/at
```

Note: In the decompiled bytecode one might find `@Metadata` annotations within some classes. The Kotlin compiler adds annotation (metadata information) to the bytecode to be used by Kotlin libraries at runtime. This information *adds features*, which might not be available with the Java programming language.

```
@Metadata(d1 = {"\u0000 \n\u0002\u0018\u0002\n\u0002...\
    ↪ u0010\b\u001a\u00020\¨t\u0006\n"}, d2 = {"Lat/fhj/
    ↪ ims/ShowActivity;", "Landroidx/appcompat/app/
    ↪ AppCompatActivity;", "()V", "onCreate", "", "
    ↪ savedInstanceState", "Landroid/os/Bundle;", "
    ↪ playSong", "v", "Landroid/view/View;", "
    ↪ app_release"}, k = 1, mv = {1, 5, 1}, xi = 48)
/* compiled from: ShowActivity.kt */
```

```
public final class ShowActivity extends
    ↪ AppCompatActivity {
```

- dex2jar: https://github.com/pxb1988/dex2jar
- JD GUI: decompile java with http://java-decompiler.github.io.
- apkparser: similar to apkanalyzer: https://github.com/hsiafan/apk-parser
- smali/baksmali: assembler/disassembler, i.e. for decompiling **old dex** files for Dalvik https://github.com/JesusFreke/smali
- VS Code Plugin **APKLab** https://github.com/surendrajat/apklab to open `*.apk` and inspect resources as well as code.

3.12.2 Dynamic Analysis

- **Frida** https://frida.re allows to patch apps by hooking into API calls, to *rewrite methods* (using JavaScript scripts).

3.12.3 Inspect Network Traffic

- **Burp** Suite https://portswigger.net/burp to intercept network traffic and view request and responses.
- **mitmproxy** Man-In-The-Middle proxy https://mitmproxy.org to inspect HTTPs encrypted traffic on the fly (if the mitmproxy CA certificates can be installed on the client device). Mitmproxy can be instrumented (automated) using a Python API.

Chapter 4

Part 3 – Android Apps

Android Applications within the Linux Operating System.

The general structure of applications within the Android operating system and frameworks. The interplay of static resources and source code to make apps work. Ways of interaction with the operating system, such as life cycle callbacks to save state and/or communicate with other (system) apps.

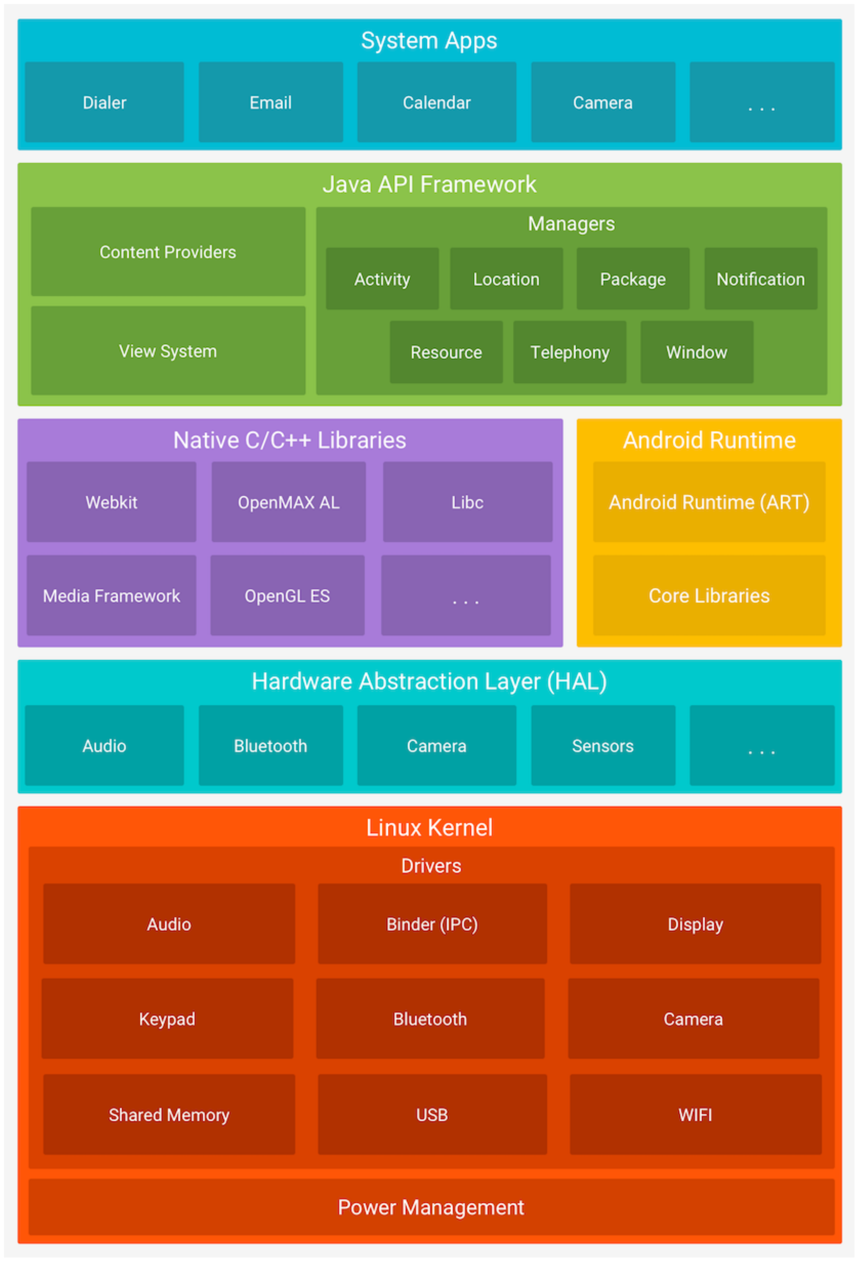
4.1 Android Applications

4.1.1 Layers

- Five Layers
  - (5=Top) Apps
    - (5a) **System Apps**: Dialer, Email, Cal, Cam,...
    - (5b) Custom Apps: Installed by users
  - (4) **Java API Framework**: Content Provides, View System, Managers (Activity/Location/Package/Notification/Resource/Telephony/Window)
  - (3) NativeLibs **OR** ART
    - (3a) Native C/C++ Libraries: Webkti, Libc, Media Framework, OpenGL ES, OpenMAX AL, ...
    - (3b) **Android Runtime** (ART) with Core Libraries
      - supports Ahead-of-Time (AoT) compiling (in comparison to the previous Dalvik java virtual machine with Just-In-Time (JIT) compilation).
  - (2) **Hardware Abstraction Layer** (HAL) for Audio/BT/Cam/Sensors/...

– (1=Bottom): (**S**ecurity **E**nhanced) **Linux Kernel**

* ∗ Drivers: Shared Memor, Binder (IPC), Audio, Display, Keypad, Bluetooth, Camera, USB, WIFI
* ∗ Power Management



The image is taken from Android Developers / Platform /Technology /

Platform Architecture https://developer.android.com/guide/platform

4.2 Android Apps

4.2.1 Manifest

- Permissions

4.2.2 Resources

- Internationalisation (i18n) for supporting multiple languages
- XML Layout files
- R

4.2.3 Source Code

Kotlin Source code.

- Source code is compiled into classes and assembled in **classes.dex** files. Multiple dex files are called **multidex**. The *.dex files and the resources are put into an Android Package* **.apk**. A more flexible way is to use Android bundles **.aab** allow the Android Play Store to dynamically customise an App before download for a given device.

4.2.4 Code to interact with the User Interface (UI)

Resources, such as UI elements are defined in XML layout files. They are assigned unique ids (which can be found in the auto-generated `R` file. After UI elements are loaded from XML files into memory they can be accessed via code. Kotlin source code uses `findViewById` to get a reference (and to access/modify/add listeners) to UI elements.

4.2.5 Navigation (data passing)

- Start **activities** using **intents**

`startActivity()` , `startActivityForResult()`

- Implicit vs. explicit intents

Implicit intents are dynamically found at runtime. Apps register available activities (see **intent filters**) which can be called by other apps. For example your app needs routing. Therefore an external app might be called using an **implicit intent**. The system finds the proper app for it (e.g. Google Maps or Open Street Map).

- Navigation

  - Tab
  - Stack
  - Master-detail

4.2.6 Life-cycle

- During an apps **life cycle** it changes between the states: *resumed*, *paused*, and *stopped*. Each transition triggers a callback. If needed, developers provice code to be executed for a specific callback, such as:
  - onCreate
  - onStart (onRestart)
  - onResume
  - onPause
  - onStop
  - onDestroy
- Layout, Views and Interaction

Layout is specified in XML files. The views are composed by nested views and widgets. Activities might exchange parts of the UI, called **fragments**. Activities, as well as fragments have their own **Life Cycles**. Use the **callbacks** to react on events, such as app is leaving the foreground.

For interaction swipe **gestures** and **taps** are used, so better try this with your fingers on *real* devices.

4.2.7 Usability

- Never block the User Interface (UI). Use concurrency mechanism (Kotlin coroutines) to prevent blocking the main thread.
- Save state! Use life cycle callbacks to preserve (and restore previous) state. The user should not even notice if an app was terminated between usage.
- Reduce user input. Use sensors. Use haptic and/or sound feedback. Provide support for deep linking to start apps from a (mail/messenger) message, QR code or web page.
- Test on real devices. Do not perform usability tests in the emulator using the point-and-click metaphor of a mouse.

4.2.8 Save battery

Use life cycle callbacks to reduce memory and CPU usage. Apps requiring fewer memory might be keept longer alive and are not terminated by the system. Apps requiring much energy, might be deinstalled by users.

4.2.9 Services

Apps can be started as long running forground or background services.

4.2.10 Sensors and Actuators

- Sensors
  - Motion/Position: Gyroscope, Accelerator, GPS
  - Camera
  - Barometer (relative height), Temperature
  - Ambient Light
- Actuators
  - Vibration

4.2.11 Local and Push Notifications

- Local notifications: listen to specific notification events or timer-notifications.
- Push notifications might be used with the Google **Firebase** infrastructure.
- Compare:
  - Cloud Synchronisation with **Firebase**

4.3 Interaction between (system) apps

- Interprocess Communication (IPC)
- An app can register to listen for system events (broadcast receivers), such as SMS, phone calls, calendar alerts.
- **Deep Linking** with URI schemes. This allows one app to call (specific parts of) another app using a crafted URI, such as:

```
adb shell am start -W -a android.intent.action.VIEW -d "slideshow://
↪ holiday/2023/17" slideshowapp.ims.iit.fhj.at
```

Chapter 5

Part 4 Android APIs

Modern Android Development (MAD) and selected step-by-step instructions for development using Android libraries and APIs.

5.1 Part 4: Android APIs

5.1.1 Designing / Structuring an App

Larger apps might be structured by features or might have a layered architecture. Structure by using modules. Modules can be loaded on demand.

5.1.2 MAD

Use the Modern Android Development **MAD** design patterns / **architectural components**. MAD are layers, frameworks, libraries which work upon the basic Android functionality, as discussed in Part 3 – Android Apps). Components are **lifecycle aware** (discussed below).

Use the libraries of **Android Jetpack**.

- Requirement in `build.gradle.kts` :

```
...
dependencies {
    val lifecycle_version = "2.2.0"

    implementation("androidx.lifecycle:lifecycle-
    ↪ livedata-ktx:$lifecycle_version")
    implementation("androidx.lifecycle:lifecycle-
    ↪ viewmodel-ktx:$lifecycle_version")
    ...
}
```

- **ViewModel**
  - Preserve (cache) app data, even when view changes (i.e. app rotates to landscape). *Advantage: No reload/fetch of data is necessary when UI configuration changes, e.g. navigate between fragments.*
  - Optionally, persist data between app recreation (start/stop).
    * Prepare a view model:

```
data class DiceUiState( ..)
class DiceRollViewModel : ViewModel() {
    // Expose screen UI state
    private val _uiState = MutableStateFlow(
    ↪ DiceUiState())
    val uiState: StateFlow<DiceUiState> =
    ↪ _uiState.asStateFlow()

    // Handle business logic
    fun rollDice() {
        _uiState.update { ....
    ...
}
```

    * Create your view model (when activity is created) and write code how the UI should be updated with data from the model

```
class DiceRollActivity : AppCompatActivity() {
    override fun onCreate(...){
        val viewModel: DiceRollViewModel by
    ↪ viewModels()
        lifecycleScope.launch {
            repeatOnLifecycle(Lifecycle.State.
    ↪ STARTED) {
                viewModel.uiState.collect {
                        ...// Update UI elements
                }
            }
        }
    }
}
```

- **LiveData**
  - To observe data changes. Data objects notify UI elements (views) when underlying data/databases change. *Notes: no manual lifecycle handling required.*

        * Create (empty) instance of `LiveData` (to hold data)

```
class NameViewModel : ViewModel() {

    val currentName: MutableLiveData<String> by
   ↪ lazy {
        MutableLiveData<String>()
    }
```

        * When activity is created, **create an observer**. Then **start observing**.

          “‘kotlin class NameActivity : AppCompatActivity() { private val model: NameViewModel by viewModels()

```
  override fun onCreate(...){
      ...
      val nameObserver = Observer<String> {
   ↪ newName ->
          nameTextView.text = newName
      }
      model.currentName.observe(this,
   ↪ nameObserver)
      ...
  }
  ```
```

- **DataBinding**
    - Bind **declarative** UI elements to variables (data sources) of your app.
        * No need for `findViewById`. Code is generated.
            · Like always, define UI elements in XML

```
<LinearLayout ... >
    <TextView android:id="@+id/username" />
     ....
    <Button android:id="@+id/confirmbutton"
   ↪ ... />
</LinearLayout>
```

            · During project compilation, the binding library creates code, i.e. variables (within `binding`) to access UI elements.

```
private lateinit var binding:
   ↪ ResultProfileBinding
```
```

```
override fun onCreate(...) {
    ..
    binding = ResultProfileBinding.inflate(
    ↪ layoutInflater)
    val view = binding.root
    setContentView(view)
}
```

  · Use variables (here `username` and `confirmbutton`) in code

```
binding.username.text = viewModel.name
binding.confirmbutton.setOnClickListener {
    ↪ viewModel.userClickedConfirm() }
```

  – Compare one-way and two-way data binding with `@={}` notation.

```
<CheckBox
    android:id="@+id/rememberMeCheckBox"
    android:checked="@={viewmodel.rememberMe}"
/>
```

- **Repository**

  – Persistence via O/R-Mapping for a local SQL database: Use **Room** and specify a *database*, *entities* and *DAOs*.

```
@Entity
data class User(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = "first_name") val firstName:
    ↪ String?,
    @ColumnInfo(name = "last_name") val lastName:
    ↪ String?
)
```

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): List<User>
    ...
```

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

```
val db = Room.databaseBuilder(
    applicationContext,
    AppDatabase::class.java, "database-name"
).build()
 val userDao = db.userDao()
val users: List<User> = userDao.getAll()
```

  - Access Remote Data Source (webservice): **Retrofit**

```
private val retrofit = Retrofit.Builder()
    .addConverterFactory(ScalarsConverterFactory.
   ↪ create())
    .baseUrl(BASE_URL)
    .build()
```

```
val listResult = MarsApi.retrofitService.getPhotos()
```

- **WorkManager**
  - For async tasks
  - Scheduled in the background

```
val continuation = WorkManager.getInstance(context)
    .beginUniqueWork( ...)
    .then(...)
    .then(...)
```

- **Dependency Injection** (DI)
  - **Hilt**
    * Define **one** container for the app

```
@HiltAndroidApp
class ExampleApplication : Application() { ... }
```

  - Where you want to use DI, annotate your custom (Application/ViewModel/Activity/Fragment/View/Service/BroadcastReceiver) classes, like:

```
@AndroidEntryPoint
class ExampleActivity : AppCompatActivity() { ... }
```

- With the annotated class, annotate a variable to inject dependency:

```
@Inject lateinit var analytics: AnalyticsAdapter
```

- Define the bindings

```
class AnalyticsAdapter @Inject constructor(
  private val service: AnalyticsService
) { ... }
```

- The (single) object/instance/service you want to inject must be provided

```
interface class AnalyticsServiceImp { ...
class AnalyticsServiceImp ... (): class
    ↪ AnalyticsServiceImp {  ...}
...
```

- **Dagger** with auto-code-generation to manage complex dependencies.

• **Navigation**

- Manage the navigation flow within the app.
- Define/create a navigaton graph for **one single activity**.

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="..."
            xmlns:app="..."
            android:id="@+id/nav_graph">

    <fragment
        android:id="@+id/view_pager_fragment" ... >
        <action
            android:id="@+id/
    ↪ action_view_pager_fragment_to_plant_detail_fragment
    ↪ "
            app:destination="@id/plant_detail_fragment"
    ↪ ... />
    </fragment>
</navigation>
```

- Trigger navigation from Code. Note, a navigation host (Fragment/View/Activity) has a `NavController` with method `navigate`.

```
override fun onClick(v: View) {
val amount: Float = ...
val action =
    SpecifyAmountFragmentDirections
        .
    ↪ actionSpecifyAmountFragmentToConfirmationFragment(
    ↪ amount)
v.findNavController().navigate(action)
}
```

- **Declarative UI**
  - Write UI in code with **Jetpack Compose**

```
@Composable
fun PlantListItemView(plant: Plant, onClick: () -> Unit)
    ↪  {
...
Text(
    text = plant.name,
    textAlign = TextAlign.Center,
    modifier = Modifier
        .fillMaxWidth()
        .padding(vertical = dimensionResource(id = R.
    ↪ dimen.margin_normal))
        .wrapContentWidth(Alignment.CenterHorizontally)
)
```

- Of course, the is much more:
  - *Paging* to load information gradually. Paging includes the use of *RecyclerView* adapters.

Note: It is important to note, that the architecture components (listed above) are components, which are lifecycle-aware. That means, code to react on lifecycle changes is written in other places. I.e. one can move code to react on lifecycle state changes away from the activities or fragments direct into the component.

5.2 Secure Coding

Use Kotlin (see Part 1 Kotlin) it is more secure. For example, it provides **Null Safety**. Find more on secure coding in Part 6 Security / Attacks and Mitigations.

5.3 Step-by-step Tutorial

Starting with an app idea the development steps are explained step-by-step.

5.3.1 Part 4 — Step 1 A Slideshow App

Why we need an app? Why we need a slideshow app?

Read Part 4 - Step 1 Intro.

5.3.2 Part 4 — Step 2 Prototyping a Slideshow

How to spare many hours of rework?

Read Part 4 - Step 2 Prototyping.

5.3.3 Part 4 — Step 3 Slideshow App-Structure and Incemental Development

How to setup the first app structure and then improve your app step by step?

Read Part 4 - Step 3 App Structure and Incremental Development.

5.3.4 Part 4 — Step 4 Idiomatic Kotlin for Coding Slideshow Functionality

Use Idiomatic Kotlin for the data structures and the algorithms during Slideshow Develoment. Provide some holiday slides as demo data and search, filter, transform the given data set, e.g. the list of images.

Read Part 4 - Step 4 Idiomatic Kotlin.

5.3.5 Part 4 — Step 5 GUI — User Interface of the Slideshow

Presenting slides and interacting with an slideshow app.

- Part 4 - 5a Activities, Intents, Fragments: Display static slideshow images
- Part 4 - 5b Android App Life-cycle, Broadcast Receivers: Use adb to broadcast slide number. Show slide with given number.
- Part 4 - 5c Navigation: Show image details for a single slide
- Part 4 - 5d Data Binding: Fewer code with binding variables to UI. Timer, Layout: Present slides as images with titles
- Part 4 - 5e Interaction: Allow users to rate a slide
- Part 4 - 5f Filter: Add functionality to show sunset imgages only
- Part 4 - 5g Save State: Remember the last viewed slide when reopening the app

5.3.6 Part 4 — Step 6 Fetch Data

Never block the user interface! Use concurrency to fetch data in the background.

- 6a Do not block the UI, WorkManager: Load data in the background

- 6b Dependency Injection with Hilt and Dagger: Load data from data stores
- 6c Retrofit: Fetch data from remote web services

5.3.7 Part 4 — Step 7 Caching for Performance (and UX)

- Room, Alarm Manager: cache data and check for update regularly.

5.3.8 Part 4 — Step 8 Location Based Services

- Geo Location and Distance: Show location where photo was taken.
- Location: Point of Interests (POIs), Distance
- Map: Pins, Landmarks
- Detect **In range**

5.3.9 Part 4 — Step 9 Components

Code features as reusable components.

- Zoom animated components: rate slide with subtle blinking / pulsing stars

5.3.10 Part 4 — Step 10 Sensors and Actuators

- Use the camera to take images.

5.3.11 Part 4 — Step 11 Security

- Secure Hardware and Security APIs: use biometric authentication
- Secure Coding: refactor and improve code

5.3.12 Part 4 — Step 12 System Services

- Contact Store: get email of contact to share slides with.
- Content Provider: offer slides to other apps

5.3.13 Part 4 — Step 13 Cloud

- GCM Cloud and Sync Services: synchronise comments via cloud

5.3.14 Part 4 — Step 14 Notifications

Local and remote (push) notifications to remind the user of events.

- Local and/or remote Notifications: notify users that someone added a comment to a slide.

5.3.15 Part 4 — Step 15 Machine Learning (ML)

- Machine Learning ML: auto catagorise an image and show matching category (sunset, beach, water)

5.3.16 Part 4 — Step 16 Augmented Reality (AR)

- Augmented Reality (AR): project the slides onto a wall.

. . . .to be continued. . . .

Chapter 6

Part 5 – Android System Architecture

Selected details on security aspects of Android Apps within the operating system.

6.1 System Architecture

Prerequisite: For the layered Android platform architecture see Part 3: Android Apps.

6.1.1 Booting, the Operating System, and the Kernel

- Trusty TEE (trusted execution environment)
- Verified Boot for *chain of trust* and rollback protection: e.g. using a digitally signed boot image
- System Partition
- Full-disk encryption
- File-based encryption
- No swapping partition (i.e. you might run out of memory)
- SEforAndroid: Customised Security Enhanced (SE) - Linux providing mandatory access control (MAC) with *Lables* (security contexts); Additional to DAC[1] in standard Linux.
- Process isolation
- Secure Inter Process Communication IPC (see below)

[1]Discretionary Access Control (DAC): i.e. Linux rwx permissions. Compare to Role-Based Access Control (RBAC), where permissions are assigned to a specific role, users are assigned roles by administrators.

6.1.2 Starting Apps

- Apps are run in virtual machines (ART VMs) forked (spawned) from the **Zygote** process. I.e. every spawned process has it's own ART VM.
- Code is shared (by the `zygote` process) for performance reasons

6.1.3 Selected Security Features for (Custom) Apps

- App signing, certificates to verify apks
- Every (Android App-) process is forked from a special process called `Zygote` (for startup performance reasons).
- App sandbox (see above)
- User-based permissions model
  - app equals Linux user (id)
  - apps within the same (linux) group are allowed to access common (file system) resources
- APIs for Encryption, Keystore

6.1.4 About ART

Android runtime (ART) supersedes the Dalvik virtual machine. ART provides:

- Ahead of time compiling (***AOT***).
- ART precompiles Dalvik bytecode into native code, into *Linux* ELF shared object `*.oat` files.
  - compilation happens internal on device with tool `dex2oat`
  - compilation happens on system upgrade
  - compilation happens on installation of app
- Android Optimized Application `*.oat` files are available since Android 5.x Lollipop. Check out (`/data/dalvik-cache/`, and the tool `/system/bin/dex2oat`)
- Garbage collection (in Oreo) with a *Concurrent Copying Garbage Collector*

6.1.4.1 Update for ART with Android 7:

- **Hybrid compilation**, i.e. combination of ahead-of-time, just-in-time, and profile-guided copmilation.
- Details: On the first runs (with interpreted code and possibly JIT compilation) a profile of an app is created. Only then, frequently used code is compiled by a compilaton daemon when the device is idle (and charging). On subsequent starts of the app the precompiled code-parts get used and, if necessary, further code is scheduled by the JIT compiler (info added to the profile) for compilation by the compilation daemon.

6.1.4.2 History (about Dalvik):

- Dalvik used special (non-standard) java bytecode. The bytecode is stored as (Dalvik Executables DEX) `*.dex` file. Dalvik used **JIT** just-in-time compiling.
- *Details:* `*.java` is compiled to `*.class` files (= bytecode). BUT: opcodes are different on Dalvik, so it is necessary to convert the `*.class` files to `*.dex` files. On Android devices, we find the optimised dex (`*.odex`) files in a cache directory (created after first launch).

6.1.5 About IPC

Inter-process comunication (IPC) is limited basically to Intents (and Services) using the *Binder*. Explicit intents start a specific app. Implicit intents call other apps which are registered. **Binder** allows to bind apps to running services.

Note: Unlike desktop operating system, there is **no support** for signals, sockets or pipes (on higher level APIs).

Internal Note 1: IPC is also possible using so called a *bound service*: (a) by extending the *iBinder interface*, (b) implementing a *Messenger* (class), and (c) AIDL. For (c) AIDL, the IPC can be done implemented via client server communication using the Android Interface Definition Language to un-/marshall objects.

Internal Note 2: Furthermore, API level 27 and the Native Development Kit (NDK) suppport anonymous shared mem (useing a file-handle to `/dev/ashmem` which are passed to other processes/apps via `android.os.ParcelFileDescriptor`).

Chapter 7

Part 6 – Attacks and Mitigations

7.1 Know the risks

There are several (open source) projects which lists typical problems (risks) and best practice (mitigation strategies).

7.1.1 Required terms / background

- Attack Vectors
- Security Weakness

7.1.2 Know the possible risks and learn about the assets to protect.

Most common problems and risks are discussed at:

- Open Web Application Security Project (OWASP)
  - OWASP Mobile Security
  - Selected Risks from OWASP Mobile Risks Top 10 (2016):
    - M1: Improper Platform Usage
      - => E.g. Stick to Android Guidelines, API, Kotlin, Libs,...
    - M2: Insecure Data Storage
      - => E.g. Use secured shared data, encrypted filesystem and files, switch off verbose logging, ...
    - M3: Insecure Communication
      - => E.g. Use TLS, trusted CA providers, ...
    - M4: Insecure Authentication

        - => E.g. Use two factors, ...
      - M5: Insufficient Cryptography
        - => E.g. Use up-to-date Crypto-API, Algos/Ciphers and key lengths, consider SHA1, MD5 insecure, ...
      - M6: Insecure Authorization
        - => E.g. Use minimal privileges for web services (no admin user/roles),...
      - M7: Client Code Quality
        - => E.g. Use tools to check code for quality and security, ...
      - M8: Code Tampering
        - => E.g. Try to detect modified code (or rooted environment), ...
      - M9: Reverse Engineering
        - => E.g. Obfuscate your code, ...
      - M10: Extraneous Functionality
        - => E.g. Perform code reviews to find hidden features,...
- App security improvement program (ASI)
  - Flags security issues to developes on Google Play.
- Check Security Level with Mobile App Security Requirements and Verification (**MASVS**), Mobile Security Testing Guide (MSTB), Mobile App Security Checklist

7.1.3 Stick to standards and best practices.

Learn about defense strategies, improve apps, the secure environment.

- Mobile Application Security Verification Standard (**MASVS**)
- **OWASP** Mobile Security Testing Guide

7.2 Selected Attacks and Mitigations

- Attackers use **Reverse Shells**
  - E.g. with the Netcat tool

7.2.1 System and App Permissions

Attacks

- Startup an activity (e.g. start activity via `am` tool) which is not the main activity (possibly circumventing authorisation required to view the activity).

Mitigation

- Manifest
  - Do not allow to start intents externally: `android:exported="false"`.

  - *Signature-based permissions* to communicate between two of your apps. Checks for same signing key.

7.2.2 Code, Programming Language & APIs

Attacks

Apps are never secure. The algorithms, the code is never proven correctly. Often, evil-crafted user input will break apps.

- Development, GIT

Mitigation

- Use Kotlin, it is (slightly) more secure.
- Protect credentials. Never store API keys in the code base (git).
- Run code metrics. Use external tools, such as
  - *ktlint*, or
  - *detect*.
- Testing
  - Setup test cases.
  - Check code coverage.
- Coding with Crypto APIs
  - Basic (high level)
    - see EncryptedSharedPrefs
    - see BiometricAuthentication
  - Advanced
    - Algorithms
    - PseudoRandomNumberGenerator (PRNG)
    - Hashes
    - Message Digest
    - Message Authentication Code (MAC)
    - Symmetrical (AES, stream vs block, padding, initialisation vectors)
    - Asymmetrical (RSA, public/private keypairs)
    - Signatures
    - Certificates
    - Keystore (PKCS12, Alias)

7.2.3 Source Code Protection

Attacks

- Code Tampering
- Reverse Engineering

Mitigation

- Make Reverse Engineering harder with Code Obfuscation.

7.2.4 Secure Storage

Attacks

- Extract sensitive data from shared preferences.

Mitigation

- Use encryption provided by the system
  - EncryptedSharedPrefs
  - EncryptedFile

7.2.5 Use Cryptography

Use the **Crypto API** for hashing, signing and encrypting data. Never even try to write crypto algorithms on your own!

Attacks

- Authenticate within the app using no/weak crypto, e.g. by "guessing" passwords from md5 hashes.

Mitigation

- Keystores to generate keys and use Crypto APIs.
  - Keystore, keychain
  - Generate key pairs

Attack

- Inspect backups in the file system. The plain text data might hold credentials.

Mitigation

- **Encrypt data** when storing into the file system.

7.2.6 Authentication

Attacks

- No or broken authentication (leaked passwords).

Mitigation

- Biometric Authentication (Fingerprint)

7.2.7 Network

Required terms / background

- Certificate, Root Certificate
- Certificate Authorities (CAs)
- Certificate Chains
- Limitations of debug certificates

Attacks

- Unsecured networks allow to inspect data in transit.

Mitigation

- Transport Layer Security TLS. Always use **https**.
- In Manifest: `android:usesCleartextTraffic "false"`
- Network Security Configuration file: to specify a *custom CA* or for *certificate pinning*. Settings are done in `network-security-config`.
  - **Client certificates**
  - Optional, not suggested by Google
    - Custom CAs
    - *Certificate pinning*
    - *Public Key Pinning*

Chapter 8

Possible Exam Questions

In Section Overview: Droid Development

The answers to following questions can be found in the Study Material.

Hint: Explain your answers to a fellow student!

8.1 Apps and Architecture

- How does the **architecture of Android apps** differ from typical iOS-Apps?
- What is the main entry point into an app. Explain what the *manifest* file is used for. Explain, what are *activities* and what are *intents*?
- Explain and give examples of permissions granted at install time. Compare to runtime permissions.
- Could you explain terms such as activities, fragments, and views.
- Which unique ids and which certificates are necessary for development, signing and distribution?

8.2 Kotlin Programming

- How and where is Kotlin code executed?
- How does garbage collection work?
- Explain some of the key language features of Kotlin. For example: **type inference**, strong typing.
- Discuss compile-time vs. run-time errors/exceptions.
- What ar nullable types.
- Kotlin idioms (*idiomatic programming*): compare two of selected features (such as data classes, smart casts, operator overloading, generics,

coroutines, default values, lazy properties, singletons, not-null shorthands, if-expressions...) to other languages.

- Can you show the usage of object-oriented programming (including initialisers, inheritance, **properties**, **extensions**, ...) in Kotlin?
- Are `calss`es or `data class`es **passed by value**? Explain the (dis-) advantages of **pass by reference**.
- What is special with **Nullable types** and with the **save call operator** `?.` ?
- What are **default arguments** for functions?
- Fuctional programming: explain the use of **high-order-functions** and **lambdas**.
- Can I explain and demo advanced features of the Kotlin programming language: generics, **operator-overloading**, ...
- Could you explain the usage, the advantages and disadvantages of advanced Object Oriented Programming (OOP) with Interfaces and Extensions?
- Explain the differences between **primary** and **secondary constructors**.
- Delegates: what could be an usage of **observable** properties?
- Why would many consider `val` more secure than `var`?
- Argue for the usage of `==` over `===`.
- Compilation (on demand): could you explain the work of the Android Runtime (**ART**), which combines the advantages of *just in time* compiling with *ahead of time* compiling.

8.3 GUIs, Navigation, and Accessibility

- Explain what is necessary to create interfaces for Android Apps.
- Explain the use of an `activity` and a `fragment`.
- How can developers assign code for life-cycle methods (view appears) and user interactions (swipe gestures, tap a button).
- Name ways for **navigation**, and explain how to **pass data** from one activity to another.
- Name ways for **navigation**, and explain how to **pass data** from one fragment to another.
- Code generation: explain how (two-way) **data binding** works.
- Could you explain the most important **life-cycle** events?

8.4 Concurrency and Web Services

- How to keep the UI responsive with background tasks for longer operations?
- How to retrieve data in the background from web services?
- Explain the use of **suspendable functions** with **coroutines** to prevent blocking UIs and to improve performance?
- How and why are ReSTful web services called **asynchronously** (in concurrent threads)?

- Explain ways of **JSON serialisation** and the elaborate on the problem of type checking?
- How and when would you use **threads** over **coroutines**?

8.5 Saving State and Persistency

- How do **shared preferences** work? Give an example.
- Name three different ways to persist data.
- How to setup the O/R-Mapping (ORM) and how to update a schema?
- For which kind of data is it necessary to use secure storage?
- Explain the disadvantage of keeping data in a keystore?
- What is the difference between eager and lazy loading!
- Explain the main features of **firebase** platform, expecially authentication and the cloud firestore.

8.6 Location Based Services

- How to minimize user input? How to suggesting location, movement and context?
- What is necessary to use maps and add custom landmarks and points-of-interest (POIs)?
- How to use existing web services to improve user-experience?
- Could you explain the usage, the advantages and disadvantages of Location Based Services concerning privacy?
- Maps: what differences are between *Overlays*, *Markers*, and *Tiles*?
- Which accuracy can we expect from different ways of determining the location of a user?
- Could you name the idea and application of **geofencing**?
- What are the use and (practical) limitations of the **(reverse) geocoding**?

8.7 Sensors and Actuators

- Explain the advantages and disadvantages (according to a context) of polling versus callbacks to retrieve sensor data.
- How and when to employ different sensors (and different frameworks)? (Proximity, Ambient Light, ...) to provide feedback to users?
- Could you explain the usage, the advantages and disadvantages as well as the proper technical terminology for at least two actuators and five sensors?
- What is special (concerning the performance, power, connection time, kind of connection, security, ...) about bluetooth low energy (**BTLE**)?

8.8 Operating System Security and Insecurity, Forensics, App Analysis and Secure Coding.

- Explain the **Android Virtual Machine** and how it is embedded into the **Android Stack**.
- Consequences of **sandboxing** an app. How are restrictions/permissions enforced.
- What does it mean to code in a secure way? Give 3 examples!
- How and where to save client data (user credentials) in a secure way?
- Why and how is code quality related to security?
- Explain approaches for testing.
- What means code integrity and how to enforce code integrity.
- How can usability build trust, and could you as developer care for user privacy?
- Which steps are necessary/possible for an `*.apk` debugging session?
- (OWASP) **risks**: give examples for *insecure data storage*, *insecure communication*, *insufficient cryptography*, and *insecure authorisation*.
- Explain the deployment certificates needed to put an app to the App Store?
- Could you draw the architecture and flow during usage of **biometric sensors**?
- Which limitations and restrictions are given for **inter-app communication**?
- Name ways for inter-process communication (**IPC**) and known is uses?
- Differences between **certificate** and **public key** pinning.
- Explain the use of **URL schemes** in Android?
- Could you discuss the different approaches of **separating apps** on Android and on iOS including the consequences?
- Can you explain the terms and specific tasks of **Secure Elements**.
- Draft the steps to **decompile** `.apk` files.
- Name relevant aspects of the Android *permission model*. Detail on differences of **install-time** (normal/signature), **runtime** (prompting users), and **special** (platform/OEM) permissions.
- Limitation of **code obfuscation**. What does it mean that the R8 compiler shrinks, obfuscates and optimises an app.
- **Certificates**: Explain the terms (and differences): *client certificate*, *root certificate*, *certificate authority*, *self-signed certificate*, and *certificate chain*.
- Name at least three ways to harden your app.
- Compare **static** to **dynamic** (malware) **analysis**.

8.9 System Services and Social Frameworks

- Name and explain kinds of data exchange (not) possible between apps and the system, such as clipboard, file system, inter-process communication (IPC). Explain consequences to security and privacy. How are restrictions enforced?

- How to integrate App logic with system services such as AddressBook, PhotoAlbum or Calendar? Could you sketch the steps for accessing contacts?
- In which ways it is possible to integrate social media services for sharing or login?
- Which way of communication from Android to Android watches can be implemented?
- Explain the basic concepts of selected IPC methods, such as **binder, signals, sockets/ports, streams, pipes, or shared memory**.

8.10 Cloud Services, Monitoring, and Testing

- Which Google cloud services are available to developers and how to integrate them into own apps? Name limits and consequences, state ways to store documents transparently in the cloud.
- Can you explain different ways of synching to cloud storage?
- Which privacy and security issues have to be considered?
- How to **test asynchronous code**?
- What are the requirement for stable **UI-Tests**? What are the limits for UI-Testing?
- Which drawbacks concerning performance, security, footprint, cross platform usage are to expect with different cloud service providers?
- How could a cloud based password store be safe?

8.11 Local and Remote (Push) Notifications

- What are the important elements of the Google **push notification architecture**?
- Could you name the main differences between the Apple, Google and Microsoft Push Services?
- State ways to consume the notification inside the client app?
- Can local and/or remote push notifications mitigate the problem of battery draining background processes?
- Could you draft first the **registration process**, the creation and use of credentials/logins/tokens, and then the flow of notifications. Explain the flow (step-by-step) and show how to setup push notification on the server?
- How and where can push notification services be attacked? Consider the security implications for developers, especially the app server with custom setup and custom logic.
- How **background services** work with Android compared to push notifications.