Mobile Security & Hacking

The Complete Practical Handbook: 300+ Lessons on Android & iOS Security, Mobile Pentesting, Exploit Development and Bug Bounty – Challenges, Quizzes and Full Solutions

Mobile Security Lab · mobilesecuritylab.com

Mobile Security & Hacking: The Complete Practical Handbook: 300+ Lessons on Android & iOS Security, Mobile Pentesting, Exploit Development and Bug Bounty – Challenges, Quizzes and Full Solutions

First edition 2026

© 2026 Babala Özdemir, Füllinsdorf, Switzerland. All rights reserved. No part of this book may be reproduced or distributed without the written permission of the publisher.

Publisher: Babala Özdemir (Mobile Security Lab), Füllinsdorf, Switzerland · mobilesecuritylab.com

The lessons, examples and exercises were written for the Mobile Security Lab courses with the help of AI tools and reviewed editorially. All companies, apps, people and data in the examples are fictional. **Authorised testing only:** the techniques in this book are for learning, defence and testing systems that you own or have explicit written permission to test. Misuse can be illegal; the publisher and author accept no liability for it. Independent training, not affiliated with or endorsed by any vendor or product mentioned. Despite careful checking, no guarantee of error-free content is given. Feedback is welcome at mobilesecuritylab.com.

How to use this book

15 courses with 291 lessons on mobile security and exploitation: clear explanations, worked examples, hands-on challenges, knowledge checks, final exams and a full solutions appendix.

How the lessons work. Each lesson explains one topic with the reasoning behind it, shows the commands and examples, and often ends with a **Challenge** and a few knowledge-check questions. Try the challenge on your own lab first; worked solutions and answers are collected in the **Appendix: Solutions** at the end of the book. Each course ends with a final exam; the answer key is in the appendix as well.

Build a lab. Practise only on devices, apps and networks that you own or have written permission to test. A throwaway phone or emulator, your own test apps and a separate lab network are enough for almost everything in this book. Where a lesson mentions a cloud device lab, use your own test device or emulator instead, or the online labs at mobilesecuritylab.com.

Authorised testing only. Security techniques can be misused. Use them to learn, to defend and to test systems you are allowed to test, and follow the law of your country.

Interactive version. Every course in this book is also available online at **mobilesecuritylab.com**, including cloud lab devices for hands-on practice.

This volume has 291 lessons.

Contents

Android for Beginners ... 11

How Android Works — Linux, ART and the Sandbox ... 12
Apps, APK and the Play Store Model ... 20
Developer Options and USB Debugging ... 20
Meet ADB — Your Main Tool ... 21
Reading Device Logs with logcat ... 22
Emulator Setup ... 22
Rooting — What It Is and the Risks ... 23
File System and App Data Layout ... 24
Reading an APK — a First Look ... 25
App Components & Intents ... 25
App Signing, Updates & Installing Safely ... 26
Ethics and Legal Boundaries ... 27
The Android Permission Model ... 28
Platform Defences — SELinux, Verified Boot & Play Protect ... 29
Android Versions & Security Patch Levels ... 29
How Apps Talk to the Network — HTTPS & Network Security Config ... 30
Harden Your Own Phone — a Settings Checklist ... 31
OWASP MASVS & MASTG — the Testing Map ... 32
Practice — a First Mini-Audit of Your Own App ... 33
Where to Go Next ... 34
Final exam ... 35

Android Application Security ... 37

Android AppSec & the Cloud Device Lab ... 38
The Toolchain & Lab ... 46
Scope, Rules of Engagement & Threat Modeling an App ... 46
Static Analysis — Decompiling the APK ... 47
Manifest & Build Configuration Review ... 47
Insecure Data Storage ... 48
Secure Storage Done Right — Android Keystore ... 49
Dynamic Analysis with Frida ... 50
Traffic Interception & SSL Pinning Bypass ... 50
Network Hardening — Network Security Config & Pinning Done Right ... 51
Exported Components & Insecure IPC ... 51
Deep Links & WebView Attacks ... 52
Secure IPC, Deep Link & WebView Patterns ... 53
Hardcoded Secrets, Logging & Backend Leaks ... 53
Root Detection & Anti-Tamper Bypass ... 54
Play Integrity & Server-Side Trust Decisions ... 55
The Mobile Backend — IDOR, BOLA & Tokens ... 55
MASVS v2 Methodology & Automated Scanning in CI ... 56
Reporting Mobile Findings — Severity, Evidence & Disclosure ... 57
Final exam ... 59

Android Exploit Development ... 61

Native Exploitation on Android ... 62
ARM64 Assembly Crash-Course ... 69
Setting Up a Safe Native Research Lab ... 69

The JNI Boundary — Where Bugs Live ... 70
Debugging Native Code — Tombstones, Symbols & LLDB ... 71
Finding & Triggering a Memory Bug ... 71
Android Mitigations & Modern Chains ... 72
The Android Sandbox — UIDs, SELinux & seccomp ... 72
Reversing JNI/NDK Libraries ... 73
Native Memory-Corruption Bug Classes ... 74
What Each Mitigation Means for You ... 74
Return-Oriented Programming on ARM64 ... 74
Fuzzing Native Libs with AFL++ ... 75
Catching Bugs Early — HWASan, UBSan & MTE ... 75
Triaging a Crash for Exploitability ... 76
Secure JNI & NDK Coding Patterns ... 77
Hardening Your NDK Build — Flags You Can Verify ... 78
Reading Android Security Bulletins & Patch Levels ... 79
Reporting Native Bugs — Impact, CVSS & Coordinated Disclosure ... 80
Final exam ... 82

Android Fuzzing & Vulnerability Research ... 84

From Bug Hunting to Vuln Research ... 85
What to Fuzz on Android ... 92
Fuzzing Fundamentals ... 92
Building a Safe Fuzzing Lab: Emulators, Devices and Scope ... 93
Building a Fuzzing Harness ... 94
AFL++ Setup & Native Harnesses ... 94
libFuzzer and cc_fuzz in AOSP ... 95
Corpus, Seeds & Mutations ... 96
Structure-Aware Fuzzing: Dictionaries, FuzzedDataProvider and Grammars ... 96
Sanitizers: ASan, UBSan, MSan ... 97
Measuring & Growing Coverage ... 98
Planning Campaigns and Reading Fuzzing Metrics ... 98
Fuzzing Java and Kotlin Logic ... 99
Fuzzing System Services & Binder ... 100
Crash Triage & Exploitability ... 101
Root-Cause Analysis: Symbolization, Tombstones and Debuggers ... 101
From Crash to CVE ... 102
From Bug to Fix: Patching and Android Platform Mitigations ... 102
Continuous Fuzzing in CI and Final Review ... 103
Final exam ... 105

iOS for Beginners ... 107

How iOS Works — Architecture and the Sandbox ... 108
The iOS Security Model ... 115
Apps, IPA and the App Store Model ... 115
Code Signing, Provisioning Profiles & Entitlements ... 116
Permissions, Privacy Prompts & Privacy Manifests ... 117
Settings, Profiles and Developer Mode ... 118
Connecting a Device to Your Mac (Xcode) ... 119
Jailbreak — What It Is and the Risks ... 120
Updates, Lockdown Mode & Hardening Your Test Device ... 120
Simulator vs Real Device for Testing ... 121
Building Your iOS Testing Lab ... 122
A First Look at App Files and Structure ... 123

App Data Containers — Where Apps Keep Their Files ... 124
URL Schemes, Universal Links & Inter-App Communication ... 125
Network Security Basics — HTTPS & App Transport Security ... 126
Reading Logs & Crash Reports Safely ... 127
Ethics and Legal Boundaries ... 128
Keychain, Data Protection & the Secure Enclave ... 128
OWASP MASVS & MASTG — the Testing Map ... 129
Where to Go Next ... 130
Final exam ... 132

iOS Application Security ... 134
iOS AppSec & the Cloud Device Lab ... 135
OWASP MASVS & MASTG — Scope and Test Plan ... 143
The iOS Security Architecture in One Lesson ... 144
Jailbreak, Tooling & Getting the Binary ... 145
Keychain & Insecure Storage ... 145
Secure Storage Done Right — Keychain & Data Protection ... 146
Runtime Hooking & Jailbreak-Detection Bypass ... 147
Traffic Interception & Pinning Bypass ... 147
Secure Networking — ATS & Pinning Done Right ... 148
Static Analysis & the Mach-O Binary ... 149
Entitlements, Info.plist & Privacy Manifest Review ... 149
Anti-Frida & Anti-Hook Bypass ... 150
URL Schemes, Universal Links & IPC ... 151
WKWebView Security Review ... 151
Side-Channel Data Leakage ... 152
Local Authentication & Biometrics — Secure Patterns ... 153
The Mobile Backend — IDOR, BOLA & Tokens ... 154
App Integrity — App Attest & DeviceCheck ... 154
Reporting iOS Findings — MASVS, Severity & Retest ... 155
Final exam ... 157

Mobile Penetration Testing ... 159
The Mobile Pentest Methodology ... 160
OWASP MASVS & MASTG ... 167
Scoping & Threat Modeling ... 168
Building Your Mobile Test Lab: Devices, Emulators and Proxies ... 168
Test Accounts, Test Data and Evidence Handling ... 169
Static Analysis & Storage Review ... 170
Android Manifest and Component Exposure Review ... 170
iOS Info.plist, Entitlements and ATS Review ... 171
Hard-Coded Secrets in App Packages ... 172
Reviewing Cryptography, Keystore and Keychain Use ... 173
Dynamic & Network Testing ... 174
Verifying TLS Configuration and Certificate Pinning ... 174
Authentication, Biometrics and Session Controls ... 175
WebViews and Deep Links Review ... 176
Testing the Mobile Backend Within Scope ... 177
Resilience and Privacy Checks ... 178
Automating Mobile Security Testing ... 179
Reporting a Mobile Pentest ... 179
Remediation Guidance and Retesting ... 180
Final exam ... 182

Mobile Anti-Tamper & Resilience (L1→L5) ... 184
MASVS-RESILIENCE — Why Apps Defend Themselves ... 185
The OWASP MASVS & MASTG Standard ... 192
Threat Modelling Resilience Controls ... 192
The Analyst Method — Locate, Neutralise, Verify ... 193
Tooling Concepts — jadx, Ghidra, Frida, LLDB ... 194
Static Assessment: Locating a Control ... 194
Dynamic Assessment: Observing Behaviour ... 195
L1 — Root & Jailbreak Detection ... 196
Emulator & Instrumented-Environment Detection ... 196
L2 — Obfuscation & String Hiding ... 197
L3 — Anti-Hooking, Anti-Frida & Anti-Debug ... 197
L4 — Integrity & Signature Verification ... 198
Certificate Pinning as a Resilience Control ... 199
L5 — Layered Defences & Defence-in-Depth ... 199
Defender's Playbook — Make Tampering Harder ... 200
Protecting Keys & Secrets on Device ... 200
Remote Attestation in Practice ... 201
Tamper Telemetry & Server-Side Response ... 202
Writing a Resilience Assessment Report ... 203
Final Exam — Assess an App's Resilience ... 203

Mobile Malware Analysis ... 205
Mobile Malware Analysis & the Analyst Mindset ... 206
Mobile Malware Types & Families ... 213
Building a Safe Mobile Analysis Lab ... 214
Sample Intake, Hashing & Chain of Custody ... 214
Android APK Triage Workflow ... 215
Dropper & Loader Patterns ... 216
Permission & Accessibility Abuse — Reading the Signals ... 216
Command-and-Control Indicators ... 217
Static Triage: Manifest, DEX, Strings & YARA ... 218
YARA Rules That Work — Testing, Scope & False Positives ... 218
Dynamic & Sandbox Behaviour Analysis ... 219
Network IOC Extraction ... 220
Sharing Indicators — Defanging, STIX, MISP & TLP ... 220
iOS Malware & Configuration-Profile Abuse ... 221
iOS Device Triage — Backups, Sysdiagnose & MVT ... 222
De-obfuscation Concepts ... 223
Case Studies — Public Mobile Malware Campaigns ... 223
From Analysis to Defence — Containment & Prevention ... 224
Writing the Analysis Report ... 225
Exam — Mobile Malware Analyst ... 225

Mobile DFIR (Forensics & Incident Response) ... 227
Mobile DFIR — Why Phones Are the Crime Scene ... 228
Forensic Process & Chain of Custody ... 235
Building Your Mobile Forensics Lab & Toolkit ... 236
Seizure, Isolation & Device Lock States ... 236
Android Acquisition — Logical & Filesystem ... 237
Android System Artifacts — Usage, Packages & Logs ... 238
iOS Acquisition & Backups ... 239
iOS System Artifacts — KnowledgeC, Biome & Unified Logs ... 239

App Data Artifacts — SQLite, Plists & SharedPrefs ... 240
Messaging Artifacts — SMS, iMessage, WhatsApp & Signal ... 241
Location & Media Artifacts — EXIF, Photos.sqlite & Wi-Fi ... 241
Timeline Building ... 242
Deleted-Data Recovery Concepts ... 243
Cloud & iCloud Artifacts ... 243
Incident Response on a Compromised Device ... 244
iOS Compromise Triage with sysdiagnose & MVT ... 244
Android Compromise Triage — Apps, Permissions & AndroidQF ... 245
Tool Validation, Verification & Peer Review ... 246
Reporting — From Artifacts to Conclusions ... 247
Exam — Mobile DFIR Analyst ... 247

Exploit Development ... 249

Beyond ret2win — the Real Problem ... 250
Crashing It in GDB ... 256
Finding the Exact Offset ... 257
The Calling Convention (RDI, RSI, …) ... 258
What Is a ROP Gadget? ... 258
Finding Gadgets ... 259
Building the ROP Chain ... 259
Scripting Exploits with pwntools ... 259
Bad Chars & the movaps Trap ... 260
Why ROP Exists — NX / DEP ... 261
ASLR, PIE & the Need for Leaks ... 261
Format Strings as a Leak Primitive ... 262
GOT, PLT & ret2plt Leaks ... 262
ret2libc & Popping a Shell (Concept) ... 263
ret2csu: Controlling rsi and rdx ... 264
Stack Pivoting ... 264
Sigreturn-Oriented Programming ... 265
one_gadget: Instant Shell in libc ... 266
How Defenders Stop ROP ... 266
Final exam ... 268

Malware Analysis & Reverse Engineering ... 270

Malware Types & Building a Safe Analysis Lab ... 271
Malware Families & TTPs ... 277
Static Analysis: Strings, Hashes & Headers ... 278
Inside the PE File Format ... 279
Analyzing Malicious Documents ... 280
Dynamic Analysis: Detonating in a Sandbox ... 281
Decoding C2 & Beacon Traffic ... 282
Disassembly & Decompilation ... 282
Analyzing Shellcode ... 283
Obfuscation, Packers & Unpacking Basics ... 284
Anti-Analysis & Evasion ... 285
Windows Internals & Persistence Techniques ... 286
Memory Forensics for Malware ... 286
Automating Triage with Python ... 287
Writing YARA-Style Detection Rules ... 288
Behavioural Detection with Sigma ... 289
Mapping to MITRE ATT&CK ... 290

IOCs & Writing a Clear Analysis Report ... 291
Threat Intel & Attribution Caveats ... 291
Final exam ... 293

Bug Bounty Mastery ... 295

The Bug Hunter's Method ... 296
Recon & Mapping the App ... 304
Your Proxy Is Your Workbench ... 304
Information Disclosure ... 305
IDOR — Broken Access Control ... 306
API Access Control — BOLA & Mass Assignment ... 306
Broken Authentication & Sessions ... 307
Business Logic Flaws ... 307
SQL Injection — Auth Bypass ... 308
SQL Injection — Dumping Data ... 308
Command Injection ... 309
XML External Entities (XXE) ... 309
Server-Side Request Forgery ... 310
Insecure File Upload ... 310
Reflected XSS ... 311
Stored XSS ... 311
Cross-Site Request Forgery ... 312
Open Redirect & Its Chains ... 312
Writing a Report That Pays ... 313
Final exam ... 314

Pentest Reporting & Methodology ... 316

Why the Report Is the Product ... 317
The Engagement Lifecycle ... 324
Scoping, Rules of Engagement & Authorisation ... 324
Field Notes & Activity Logging During the Test ... 325
Scoring Risk — CVSS & Likelihood×Impact ... 326
CVSS v4.0, EPSS & Context-Aware Severity ... 327
Evidence & Reproduction Steps ... 328
Finding Taxonomy — CWE, OWASP & ATT&CK Mapping ... 329
Business & Technical Impact ... 330
Attack Chains & Narrative Writing ... 330
Actionable Remediation ... 332
Anatomy of a Professional Report ... 332
Writing the Executive Summary ... 333
Technical Writing Style for Findings ... 334
Reporting Mobile App Findings (MASVS & MASTG) ... 335
Red Team & Purple Team Reports ... 336
Report QA, Limitations & Sanitising ... 337
Reporting Platforms & Automation ... 338
Readout Meetings, Disputes & Retesting ... 339
Capstone — Write a Full Finding ... 340
Final exam ... 342

OSINT — Open-Source Intelligence ... 344

What OSINT Is ... 345
The Legal & Ethical Framework ... 353
Advanced Search & Dorking ... 354

Archives, Caches & Historical Data ... 355
Corporate & Business OSINT ... 355
People-Focused OSINT ... 356
Emails, Usernames & Exposure Checks ... 356
Research Accounts & Attribution Management ... 357
Domain & Infrastructure Recon ... 358
OSINT for Threat Intelligence ... 358
Surface, Deep & Dark Web — Awareness ... 359
Social Media Gathering ... 359
Images & Geolocation ... 360
Document & Image Metadata Analysis ... 360
Maps, Satellite & Geolocation at Scale ... 361
Source Verification & Beating Disinformation ... 362
Automation & Workflows ... 362
Protecting Your Footprint ... 363
Writing the OSINT Report ... 363
Final exam ... 365

Appendix: Solutions ... 367

Android for Beginners

The Linux under Android — ADB, APKs and a safe test setup · 20 lessons

- How Android Works — Linux, ART and the Sandbox
- Apps, APK and the Play Store Model
- Developer Options and USB Debugging
- Meet ADB — Your Main Tool
- Reading Device Logs with logcat
- Emulator Setup
- Rooting — What It Is and the Risks
- File System and App Data Layout
- Reading an APK — a First Look
- App Components & Intents
- App Signing, Updates & Installing Safely
- Ethics and Legal Boundaries
- The Android Permission Model
- Platform Defences — SELinux, Verified Boot & Play Protect
- Android Versions & Security Patch Levels
- How Apps Talk to the Network — HTTPS & Network Security Config
- Harden Your Own Phone — a Settings Checklist
- OWASP MASVS & MASTG — the Testing Map
- Practice — a First Mini-Audit of Your Own App
- Where to Go Next

How Android Works — Linux, ART and the Sandbox

Lesson 1 · Start Here

Android looks like magic until you see what is underneath: a Linux kernel, a runtime that executes your apps, and a sandbox that gives every app its own identity. Once you can picture that stack, the phone stops being a mystery and becomes a system you can reason about, inspect and test safely.

This is a beginner, setup-focused course. You will learn how Android is built and how to stand up a safe test environment — not how to break into anything. Attacking apps comes later, in courses like Android Application Security and Mobile Pentest. Here the goal is understanding, so that when you do pick up offensive tools they make sense instead of feeling like spells.

> **Authorised use only**
> Everything in this course is for your own devices, your own apps, and an emulator you control. Never apply these techniques to a phone or app you do not own or are not explicitly authorised to test. Rooting a device or inspecting an app you do not own can void warranties, breach terms of service and, in some places, break the law.

Android is Linux wearing a different face

At the very bottom of every Android phone is a **Linux kernel** — the same family of kernel that powers servers and the machines you met in a Linux course. It manages the hardware, memory and processes. What makes Android *Android* is everything stacked on top: a runtime called **ART** (the Android Runtime) that executes each app, a set of framework services that draw screens and handle calls and sensors, and the apps themselves.

Think of it as three honest layers. The kernel is the foundation. The runtime and framework are the house built on it. Your apps are the furniture inside. A security mistake can happen on any floor, but most beginner-friendly bugs live upstairs, in the apps and their data — which is exactly where this course keeps you until you are ready to go lower.

Figure 1. The Android stack. The kernel at the bottom is Linux; the sandbox at the top is just Linux user permissions applied to apps. – Four stacked layers from bottom to top: Linux kernel, native libraries and HAL, Android runtime and framework, and apps, with the sandbox spanning the app layer.

The sandbox is just Linux permissions, applied cleverly

Here is the single most important idea in Android security, and it is beautifully simple. When an app is installed, Android gives it its own **Linux user id (UID)** and its own private data directory under `/data/data/<package>`. Because Linux file permissions already stop one user from reading another user's files, apps cannot freely read each other's data. The famous "app sandbox" is mostly this: one app, one UID, one private home.

So when you later hear that an app "stored a token insecurely", what that usually means is the token sat in plaintext inside that private directory. Safe from *other apps* on a normal phone — but not safe from someone who has rooted the device, or from a backup that copies the file off. Understanding the sandbox tells you exactly why that matters and where its edges are.

> **Pro tip**

You can see the UID yourself. On a device or emulator you own, `adb shell dumpsys package com.example.app | grep userId` prints the UID assigned to that package. Two apps signed by the same developer key can opt to *share* a UID, which is a subtlety worth remembering later.

How a lesson in this course works

Every topic follows the same rhythm, so you always know what to do. You read the theory, you see the tools and the method, then you do it yourself on your own device or emulator. If you get stuck, every task has a worked solution with the exact steps.

- **1 Theory** understand how the piece fits
- **2 Tools** what to install and why
- **3 Method** the exact commands
- **4 Do it** try it on your own device
- **5 Solution** step-by-step if stuck

Hands-on: meet ADB and talk to a device

Your main tool throughout this course is **ADB**, the Android Debug Bridge. It is a command-line program on your computer that talks to a device or emulator over USB or TCP. It is not an attack tool — it is the official developer utility from Google — but it is also the first thing every tester reaches for, because it lets you list apps, read files you are allowed to read, pull data and run shell commands.

Let's walk through a first, completely safe session. You will install the tools, connect a device you own with USB debugging turned on, and ask it a few questions.

1. **Install platform-tools.** Download the official `platform-tools` package from developer.android.com; it contains `adb`. No full Android Studio needed just for this.
2. **Enable USB debugging.** On a device you own: Settings → About phone → tap Build number seven times to unlock Developer options, then turn on USB debugging. (Lesson 3 covers exactly why this is a developer-only switch.)
3. **Connect and confirm.** Plug in over USB, accept the authorisation prompt on the phone, and run `adb devices`.
4. **Ask the device questions.** List packages, check the Android version, open a shell.

The session looks like this. Comments explain what each line does:

```
# is my device connected and authorised?
adb devices
# List of devices attached
# emulator-5554   device

# what Android version is running?
adb shell getprop ro.build.version.release
# 14

# list third-party (user-installed) packages only
adb shell pm list packages -3
# package:com.example.testapp

# open an interactive shell on the device
adb shell
# $ whoami
# u0_a123      <- that is the app-style UID, not root
```

What you should see: your device listed as `device` (not `unauthorized`), a version number, and a short list of packages. The `whoami` output being something like `u0_a123` rather than `root` is the sandbox in action — even the shell is an unprivileged user on an unrooted device. That is normal and correct.

> **If a device shows "unauthorized"**
> The phone has not accepted your computer's key. Unlock the screen and look for the "Allow USB debugging?" dialog, tick "Always allow from this computer", and confirm. If no dialog appears, toggle USB debugging off and on, or run `adb kill-server && adb start-server`.

Where an app keeps its things

Knowing the layout of the file system saves you endless confusion later. An app has code and data in separate places, and the data directory is the one testers care about most.

Path	What is there	Who can read it
/data/app/<pkg>/base.apk	The installed APK — code and resources	Anyone (it is the shipped app)
/data/data/<pkg>/shared_prefs/	SharedPreferences XML files (settings, sometimes tokens)	Only the app's UID (or root)
/data/data/<pkg>/databases/	SQLite databases	Only the app's UID (or root)
/data/data/<pkg>/files/	Arbitrary app files	Only the app's UID (or root)
/sdcard/ (external)	Shared media and downloads	Many apps — never put secrets here

Notice the pattern: the private directory is protected by the sandbox, while external storage is shared. A common beginner-level app bug is writing a token or export file to `/sdcard/`, where other apps can read it. You now know why that is a mistake.

Build a lab you can break safely

You do not practise surgery on a patient, and you do not practise Android testing on your daily phone. You need a target you can reset, root, fill with test data and wipe without consequences. There are two good options, and most learners use both for different reasons.

Emulator

The Android Emulator that ships with Android Studio is free, disposable and snapshot-friendly. You can create a fresh virtual phone in minutes, take a snapshot before you change anything, and roll back instantly if you break it. Google-API images make rooting and file access straightforward for learning. The trade-offs: it has no real sensors, performance depends on your computer, and some apps that insist on real hardware or Play Integrity will refuse to run. For nearly everything in this course, the emulator is the right first choice.

Spare device

A cheap second-hand phone that you factory-reset and dedicate to testing behaves exactly like real hardware, because it is real hardware — genuine sensors, real performance, real network behaviour. That matters once you start testing apps that behave differently on emulators. The trade-offs: rooting a physical device is riskier and can be permanent, warranties go away, and you must keep it off your personal accounts. Keep it as a clean, accounts-free test rig, never your backup daily phone.

Whichever you pick, the golden rule is the same: nothing on your test device should be something you would miss. No personal accounts, no real photos, no banking app you actually use. A lab is a place to make mistakes cheaply.

Under the hood: how an app becomes a running process

You keep hearing "each app gets its own UID". It is worth seeing *when* that actually happens, because it demystifies the whole sandbox. When Android boots, the Linux kernel starts the first user-space process, `init`, which in turn starts a special process called **zygote**. Zygote loads all the common framework classes once and then simply waits.

When you tap an app icon, the system does not start a program from scratch. The `system_server` asks zygote to **fork** — to clone itself. The clone inherits the already-loaded framework in memory (which is why apps start quickly and share pages), is assigned the app's dedicated UID, has its sandbox applied, and then ART begins executing that app's code. So "one app, one UID" is not a metaphor; it is a real Linux process with a real user id that you can see.

Figure 3. Every app is a fork of zygote with its own Linux UID. The sandbox is created at the moment of that fork. – A tree: the kernel starts init, which starts zygote, which forks into separate app processes each with its own UID.

You can watch this yourself. On a device or emulator you own, `adb shell ps -A | grep com.example` shows the app as a process owned by a `u0_a…` user, side by side with `zygote` and `system_server`. Seeing the real process is the moment the sandbox stops being a slogan and becomes something concrete.

Hands-on 2: install an app and look inside its APK

Lesson 9 goes deep on reading an APK; here is a gentle first taste so the file format is not a surprise. You will install a test app you own, find its APK on the device, pull it, and list what is inside. Everything here is read-only and completely safe.

1. **Install your test app.** From your computer, install an APK you built or are allowed to use.
2. **Find where it landed.** Ask the package manager for the installed APK's path.
3. **Pull it back.** Copy the APK to your computer, where you can inspect it freely.
4. **List the contents.** An APK is a ZIP, so you can list its entries without any special tool.

```
# install a test app YOU own
adb install ./app-debug.apk
# Performing Streamed Install
# Success

# where did it go on the device?
adb shell pm path com.example.testapp
# package:/data/app/~~AbCd==/com.example.testapp-1/base.apk

# pull it to your computer
adb pull /data/app/~~AbCd==/com.example.testapp-1/base.apk ./base.apk

# an APK is just a ZIP — list what is inside
unzip -l ./base.apk
```

The listing shows the entries you now recognise — the kinds of files every APK carries:

```
   Length      Name
 --------      ----
     3284      AndroidManifest.xml
  1842560      classes.dex
   204800      resources.arsc
      ...      res/...
      ...      lib/arm64-v8a/libnative.so
      ...      META-INF/CERT.RSA
```

What you should see: a manifest, one or more `classes.dex` files, a resource table and a signature under `META-INF/`. That is the shape of every Android app.

> **What went wrong if you see…**
> `INSTALL_FAILED_ALREADY_EXISTS` means the app is already installed — reinstall with `adb install -r`. `INSTALL_FAILED_NO_MATCHING_ABIS` means the APK's native libraries do not match the emulator's CPU architecture; use an APK built for your device/emulator ABI. A signature-mismatch error means a different build is installed — uninstall it first with `adb uninstall com.example.testapp`.

When setup fights back: a troubleshooting table

Almost every beginner loses an hour to one of these before they lose a minute to a real security concept. Keep the table handy.

Symptom	Likely cause	Fix
`adb: command not found`	platform-tools is not on your PATH	Add the platform-tools folder to PATH, or run `adb` from inside it
Device listed as `offline`	Stale connection or a sleeping phone	Replug, unlock the screen, or run `adb kill-server && adb start-server`
Emulator will not start	Hardware virtualisation disabled	Enable virtualisation (VT-x/AMD-V) in firmware, or use a system image for your host
No "Build number" to tap	OEM moved it in Settings	Look under About phone → Software information on many devices
`adb devices` empty on a real phone	USB cable is charge-only, or debugging is off	Use a data cable and enable USB debugging in Developer options

Common beginner mistakes

> **Common mistake**

- Test on your daily phone and root it on a whim.
- Assume the emulator and a real device behave identically.
- Think "the sandbox" is magic, not Linux permissions.
- Install random APKs from unknown sites to practise.
- Leave USB debugging on and unlocked in public.

Better

- Use an emulator or a dedicated spare device for testing.
- Know the differences (no Play Services, easy root, snapshots).
- Understand UIDs and private dirs so bugs make sense.
- Build your own small test app, or use intentionally vulnerable lab apps.
- Treat debugging as a sensitive setting; turn it off after.

The defender's view, even for beginners

Even at this level you can think like a defender. The sandbox and permission model exist to protect users, and most real-world mobile incidents come from apps that worked against those protections — storing secrets where they did not belong, or asking for permissions they did not need.

- **Keep secrets out of shared storage** – private app directories exist for a reason; never write tokens to `/sdcard/`.
- **Request the fewest permissions** – each permission is a promise and a risk; ask only for what the feature needs.
- **Trust the server, not the device** – treat the phone as an environment the user controls, not a vault.
- **Turn off debugging in release builds** – a debuggable production app hands testers (and attackers) far more access.
- **Keep the device updated** – kernel and framework patches close the lower-floor bugs you are not touching yet.
- **Explain each permission to the user** – a clear rationale at request time builds trust and reduces over-granting.
- **Prefer scoped, modern storage APIs** – they keep app data private by default instead of leaking onto shared storage.
- **Never share a UID without reason** – two apps sharing a UID share a sandbox, so a bug in one exposes the other.

A rough picture of where your time goes

As a beginner, most of your effort is setup and reading, not breaking. The chart below sketches that balance for a typical learner working through this course. It is a teaching illustration, not measured data.

Where a beginner's time goes in this course (illustrative)

- Reading / theory – 75/100
- Tool setup – 60/100
- ADB practice – 55/100
- Reading an APK – 40/100
- Emulator fiddling – 35/100

Figure 2. ADB has three parts: the client you type into, a background server, and the adbd daemon on each device. One command can drive a real phone or an emulator. – A diagram showing the adb client on a computer talking through the adb server to adbd on a phone and emulator.

How to practise so it actually sticks

Reading about Android is not the same as knowing it. The learners who progress fastest build a tiny, repeatable habit loop instead of marathon sessions. After each lesson, do the smallest possible hands-on version of what you just read, immediately, on your lab device: one command, one file, one observation. The goal is not to finish quickly but to turn every idea into something you have touched with your own hands.

A good rhythm is a three-step loop. First, run the command from the lesson and read its output slowly instead of skimming. Second, change one thing — a different package name, a different flag — and predict what will happen before you press enter. Third, write one sentence in your own words about what you saw. That last step is what moves a fact from "I read it" to "I know it", and it is the single habit that separates people who finish this course from people who bounce off it.

Keep a lab journal
Open a plain text file and log every command you run, with a one-line note on what it did and anything surprising. It takes seconds, makes findings reproducible later, and becomes your personal cheat sheet — the same discipline professional testers use to write reports they can stand behind.

Questions beginners actually ask

These three come up in every first week. Short, honest answers, so they stop nagging you while you learn.

Is it dangerous to turn on USB debugging?
It is a powerful switch, not a disaster. While it is on, any computer you authorise can run commands against your phone, so leaving it on in public or authorising a machine you do not trust is the real risk. On your own development device it is fine; just turn it off when you are done and never tap "Always allow" on a computer that is not yours. Treat it like leaving a door unlocked: fine at home, unwise at a cafe.

Do I need to learn Linux before Android?
A little helps a lot. You do not need to be an expert, but being comfortable with a shell, file paths, permissions and users will make Android click much faster, because the sandbox and file system are Linux concepts. If terms like UID, `chmod` and `/data` are new, spend a short while with a Linux basics resource alongside this course; the two reinforce each other.

Why learn how Android works before learning to attack apps?
Because offensive tools make no sense without the model underneath. If you do not know that each app has its own UID and private directory, a finding like "the token was world-readable" is just noise. The whole point of this course is that when you later open an offensive course, every technique lands on a structure you already understand, so you learn attacks as logical consequences rather than memorised recipes.

Check yourself

Four questions before you move on

1. What sits at the very bottom of the Android stack?

A. The ART runtime
B. A Linux kernel

C. The Play Store

2. How does the app sandbox mainly keep one app from reading another's files?

A. Each app gets its own Linux UID and private directory
B. Apps run on separate CPUs
C. Encryption of every file

3. Your device shows as `unauthorized` in `adb devices`. What is wrong?

A. The phone has not accepted your computer's debugging key
B. You need to root it first
C. The battery is low

4. Where should an app never store a secret token?

A. Its private `/data/data` directory
B. Shared external storage (`/sdcard/`)
C. The Android KeyStore

5. Which process do apps get forked from, giving each its own UID?

A. system_server
B. zygote
C. adbd

6. You rename a pulled APK to .zip and list it. Which entry holds the compiled app code?

A. resources.arsc
B. classes.dex
C. AndroidManifest.xml

Key terms

- **Kernel** – The Linux core that manages hardware, memory and processes at the base of Android.
- **ART** – The Android Runtime that executes each app's compiled bytecode.
- **Sandbox** – App isolation built from a unique Linux UID and private data directory per app.
- **UID** – The Linux user id Android assigns to an app so permissions can keep apps apart.
- **ADB** – The Android Debug Bridge — the command-line tool that talks to devices and emulators.
- **adbd** – The daemon running on a device that the adb client connects to.
- **APK** – The Android application package: a zip of compiled code and resources that gets installed.
- **SharedPreferences** – A simple key/value store saved as XML in an app's private directory.
- **Developer options** – A hidden settings menu, unlocked by tapping Build number, that enables USB debugging.
- **Emulator** – A virtual Android device run on your computer, ideal for safe practice and snapshots.
- **zygote** – The warm template process Android forks to start every app, so each one begins fast with its own UID.
- **system_server** – The core system process that hosts most framework services and asks zygote to start apps.
- **classes.dex** – The file inside an APK holding the app's compiled Dalvik bytecode.
- **External storage** – Shared space such as /sdcard readable by many apps — never a place for secrets.

Android is Linux underneath, and the sandbox that protects your data is just Linux permissions giving every app its

own identity.Hold this picture and ADB in your hands — the rest of the course builds straight on both.

Apps, APK and the Play Store Model

Lesson 2 · Foundations

An app is a bundle you can open up

An Android app ships as an **.apk** file (Android Package). Despite the name, an APK is just a **ZIP archive** — you can literally unzip it. Inside you'll find a predictable set of parts, which makes Android far friendlier to explore than a closed binary.

What's inside every APK

- `AndroidManifest.xml` — the app's ID card: package name, components, and the **permissions** it requests.
- `classes.dex` — the app's compiled code, in Dalvik bytecode.
- `res/` — resources like layouts, images and strings.
- `lib/` — native `.so` libraries, one folder per CPU architecture.
- `resources.arsc` — a compiled table of the app's resources.

The Play Store model

Apps are normally distributed through the **Google Play Store**, which reviews and signs distribution. Every app is cryptographically **signed** by its developer, and Android checks that signature on install. Permissions are the other half of the model: an app must *declare* what it wants (camera, location, contacts) and the user grants or denies it.

Tools

Any unzip tool (`unzip`, or your file manager). We'll look inside a real APK properly in the "Reading an APK" lesson.

Method

1. Pick an app you install from Play and read its permission list before installing.
2. Ask: does a flashlight app really need contacts? Permissions tell you a lot.
3. Remember the five inner parts — you'll recognise them the moment you unzip one.

Challenge 1: List the five parts you expect to find inside any APK, and say in one line what `AndroidManifest.xml` is for.

Developer Options and USB Debugging

Lesson 3 · Core

Unlock the tools built into your phone

Android hides a whole menu of engineering settings called **Developer Options**. It's off by default so ordinary users don't stumble into it. Turning it on lets you enable **USB debugging**, which is what allows your computer to talk to the phone with `adb` (next lesson).

Why it's gated behind a "secret" tap

USB debugging is powerful — with it a connected computer can install apps, read logs and open a shell. That's exactly why Android makes you deliberately enable it, and why you should turn it **off again** when you're done, and never enable it on a phone you don't fully control.

Tools

Just your own Android device and its **Settings** app.

Method

1. Open **Settings → About phone**.
2. Find **Build number** and **tap it 7 times**. You'll see "You are now a developer".
3. Go back to **Settings → System → Developer options**.
4. Turn on **USB debugging**. When you first connect a computer, the phone asks you to **authorise** that computer — accept only ones you trust.

> **Challenge 1:** On a device you own, enable Developer Options and turn on USB debugging. Note where the "authorise this computer" prompt appears and why it matters.

Meet ADB — Your Main Tool

Lesson 4 · Core

The bridge between computer and phone

ADB (Android Debug Bridge) is the single most useful tool you'll learn here. It's a command-line program on your computer that talks to a device (real or emulator) over USB or the network. It comes in the **platform-tools** package — a small download you don't need the whole Android Studio for.

Tools

- **platform-tools** — download from the official Android developer site, unzip it, and you have `adb`.
- A device with USB debugging on (previous lesson), or an emulator (next lesson).

Method — the commands you'll use daily

1. `adb devices` — list connected devices; confirms the bridge works.
2. `adb shell` — drop into a Linux shell *on the device* (yes, it's Linux under there).
3. `adb install app.apk` — install an APK you own.
4. `adb pull /path/on/device local/` and `adb push local /path/on/device` — copy files off and onto the device.
5. `adb logcat` — stream the device's live log (great for seeing what an app prints).

Challenge 1: Connect your own device (or start an emulator), confirm it appears with `adb devices`, open a shell, and run `whoami` and `getprop ro.build.version.release` inside it. What Linux user are you, and what Android version is the device?

Reading Device Logs with logcat

Lesson 5 · Core

The phone is always talking — learn to listen

Android collects log messages from the system and from apps in ring buffers. `logcat` is the tool that reads them. For a tester it answers "what just happened?" after a tap, a crash or an install. For a developer it is also a classic place where secrets leak: tokens, passwords or personal data written to the log by mistake.

- Every line has a **priority** (V verbose, D debug, I info, W warn, E error, F fatal), a **tag** and a process id.
- There are several buffers: `main`, `system`, `crash`, `events`, `radio`.
- Since Android 4.1 a normal app can only read *its own* log lines; through `adb` you see the whole device log of a phone you control.

Tools

- `adb logcat` — live stream; `adb logcat -d` dumps and exits; `adb logcat -c` clears.
- `adb logcat -v time` or `-v threadtime` — adds timestamps.
- `adb logcat *:E` — only errors and above; `adb logcat -b crash` — crash buffer only.
- `adb logcat --pid=$(adb shell pidof -s com.example.app)` — one app only.
- Android Studio's **Logcat** window — the same data with filters and colours.

Method

1. Clear the log, then start the app you want to observe.
2. Filter by the app's process id so system noise disappears.
3. Perform one action at a time (log in, open a screen) and watch what appears.
4. Look for crashes (`FATAL EXCEPTION`), URLs, tokens, e-mail addresses or anything that should not be logged.
5. Save evidence with `adb logcat -d > app_log.txt`.

Defender's view
OWASP MASVS-STORAGE treats sensitive data in logs as a finding. Release builds should strip debug logging (for example with R8 rules or a logging wrapper that is disabled in release).

Challenge 1: On your emulator, show only error-level messages from the Settings app while you open and close it. Which command do you use?

Emulator Setup

Lesson 6 · Core

A safe phone that lives on your computer

The **Android Emulator** gives you a full virtual device you can wipe and recreate at will. It's the ideal place to learn: you can't brick it, it's free, and nothing you do touches a real phone. Android Studio ships an **AVD Manager** (Android Virtual Device Manager) that creates and runs these virtual phones.

Tools

- **Android Studio** — the official IDE; it bundles the emulator and AVD Manager.
- The **AVD Manager** inside it, plus the platform-tools you already have for `adb`.

Method

1. Install Android Studio and open it.
2. Open **Device Manager / AVD Manager** (the phone icon in the toolbar).
3. Click **Create device**, pick a phone (e.g. Pixel), and choose a system image (a Google-APIs image for a normal setup).
4. Launch the AVD. Then run `adb devices` — the emulator appears just like a real phone (something like `emulator-5554`).

> **Challenge 1:** Create and boot an emulator, confirm it shows up in `adb devices`, then open a shell on it with `adb shell` and confirm it's Linux by running `uname -a`.

Rooting — What It Is and the Risks

Lesson 7 · Core

Becoming root on your phone

On the Linux systems from earlier, `root` (UID 0) is the all-powerful account. On a normal Android phone you are **never** root — the shell user is unprivileged, and that's a security feature. **Rooting** means modifying the device so you can gain root access, letting you read any file and change the system.

This lesson **explains** rooting so you understand it. It is **not** a how-to guide, and you should not root your daily phone. When you later need root for real testing, use an **emulator** or a dedicated spare device — never your main phone.

Why it matters for security work

Root lets a tester read an app's private data directory, inspect databases, and hook into running apps. That's why the advanced courses use rooted *lab* devices — the power that helps a tester is the same power that malware wants.

The real risks (read these carefully)

- **Security** — root removes the sandbox guarantees; one bad app can now touch everything.
- **Warranty** — rooting typically **voids the manufacturer warranty**.
- **OTA updates** — official over-the-air system updates often **stop working** or fail on a rooted/modified device, leaving you unpatched.
- **Bricking** — a mistake in the process can leave the device unbootable.

Tools

None to run here — this is a concepts lesson. For hands-on root later, prefer a rooted **emulator image** or a throwaway test device.

Method — how to think about it

1. Assume any phone you rely on daily should stay **un-rooted**.
2. Keep a separate device or emulator for anything that needs root.
3. Weigh security, warranty and update loss before ever considering it.

> **Challenge 1:** Without rooting anything, write down three concrete risks of rooting a phone you depend on, and explain why a rooted emulator is the safer place to learn.

File System and App Data Layout

Lesson 8 · Core

Where everything lives on the device

Because Android is Linux, it has one filesystem tree starting at `/`. The parts you'll care about most are where **apps store their private data** and where **shared storage** lives. Understanding this map is the foundation for all later app-security work.

The paths to remember

- `/data/data/<package>/` — an app's **private** data directory. Each app owns its own, protected by its Linux UID. This holds `shared_prefs/`, `databases/` and files. On a normal phone you can't read another app's dir — the sandbox in action.
- `/sdcard` (also `/storage/emulated/0`) — **shared** storage: photos, downloads, anything meant to be visible across apps.
- `/system` — the read-only OS partition.

Tools

`adb shell`, and inside it the usual Linux commands (`ls`, `cd`, `cat`) plus Android's package manager `pm`.

Method

1. List installed packages with `adb shell pm list packages` — that's every app's package name.
2. Filter it: `adb shell pm list packages | grep <name>`.
3. Explore shared storage: `adb shell ls /sdcard` (this you can read).
4. Note that `/data/data/<pkg>` is only readable for that app (or via `run-as` for a debuggable app you own, or root on a lab device).

> **Challenge 1:** On your device/emulator, list all installed packages, filter for one app you recognise to get its exact package name, then list the contents of `/sdcard`. Why can you read `/sdcard` but not another app's `/data/data` folder?

Reading an APK — a First Look

Lesson 9 · Core

Open the box you learned about

Back in lesson 2 you learned an APK is a ZIP with a known structure. Now you'll actually open one — **an app you own** — and read its manifest. This is your first taste of static analysis, kept deliberately gentle. You're inspecting structure, not cracking anything.

Tools

- `unzip` — to peek at the raw contents.
- **aapt** (Android Asset Packaging Tool, in the Android SDK build-tools) — dumps a readable summary including the manifest.
- **apktool** — decodes the binary `AndroidManifest.xml` back into readable XML.

Method

1. Get an APK you own — e.g. pull one off your device: `adb shell pm path <pkg>` gives the path, then `adb pull <that path> app.apk`.
2. Peek inside: `unzip -l app.apk` — you'll see `AndroidManifest.xml`, `classes.dex`, `res/`, `lib/`, `resources.arsc`. Exactly as promised.
3. Read a summary: `aapt dump badging app.apk` — prints the package name, version, and the permissions the app requests.
4. Want the full readable manifest? `apktool d app.apk -o out`, then open `out/AndroidManifest.xml`.

> **Challenge 1:** Pull an APK you own from your device, list its contents with `unzip -l` to confirm the five parts, then use `aapt dump badging` to read its package name and the permissions it declares.

App Components & Intents

Lesson 10 · Core

The four building blocks of every app

The manifest you read in the last lesson lists an app's **components**. Android starts and connects them through messages called **intents**. Every component that other apps can reach is part of the app's attack surface, so testers always map them first.

Component	Purpose	Example
Activity	one screen with a user interface	login screen
Service	background work without UI	music playback, sync
Broadcast receiver	reacts to system or app events	boot completed, battery low
Content provider	shares structured data with other apps	contacts, files

Intents and the “exported” flag

- **Explicit intent** — names the exact target component. **Implicit intent** — describes an action (e.g. “share text”) and lets the system choose.

- **Intent filters** declare which implicit intents a component accepts.
- `android:exported="true"` means other apps may start the component. Apps targeting Android 12 (API 31) or higher must set this attribute explicitly for components with intent filters.

Tools

- `apktool d app.apk` then read `AndroidManifest.xml` for `<activity>`, `<service>`, `<receiver>`, `<provider>`.
- `adb shell dumpsys package com.example.app` — the installed app's components and permissions.
- `adb shell am start -n com.example.app/.MainActivity` — start an activity explicitly.

Method

1. List every component of *your own* app and mark which are exported.
2. For each exported one, ask: does it need to be reachable from other apps? Is it protected by a permission?
3. Write the result as a small table — this becomes the input for the IPC lessons in the advanced course.

> **Challenge 1:** Using `dumpsys` on your emulator, find the activity that is launched when you tap the app icon of an app you installed yourself.

App Signing, Updates & Installing Safely

Lesson 11 · Core

Who really made this app?

Android refuses to install an APK that is not **signed**. The signature does not prove that an app is trustworthy — anyone can create a key — but it proves that every *update* comes from the holder of the same key. That is what stops a fake “update” from replacing your banking app.

- **Signature schemes**: v1 (JAR signing), v2 (Android 7.0, whole-file), v3 (Android 9, adds key rotation), v4 (Android 11, for incremental installs).
- **Same key rule**: an update with a different signing certificate is rejected; the user would have to uninstall first and lose the app's data.
- **Play App Signing**: on Google Play the developer signs with an *upload key*, and Google re-signs the delivered APKs with the app signing key it protects. New Play apps are published as Android App Bundles (AAB).
- **Sideloading**: since Android 8.0, “Install unknown apps” is granted per source app (browser, file manager), not globally.

Tools

- `apksigner verify --verbose --print-certs app.apk` (Android SDK build-tools) — schemes used and certificate digest.
- `keytool -printcert -jarfile app.apk` — alternative view of the certificate.
- `adb shell pm path com.example.app` and `adb pull` — get the installed APK from your own device.

Method — check an APK before installing it

1. Prefer the official store. If you must sideload, get the APK from the developer's own channel.
2. Run `apksigner verify --print-certs` and note the SHA-256 certificate digest.

3. Compare it with the digest of the version already installed from the store (pull it with `adb`).
4. Different digest = different signer: do not install.

> **Never share your signing key**
> If you publish apps, keep the upload key in a password manager or hardware-backed store with a backup. A leaked key lets someone sign convincing updates in your name.

> **Challenge 1:** Pull an app you installed from the store on your emulator and print its signing certificate. Which signature schemes were verified?

Ethics and Legal Boundaries

Lesson 12 · Ethics

The rule that keeps you a professional

Everything in this course was framed around **your own** devices and apps — and that's not a stylistic choice. Touching a device, account or app you don't own or aren't authorised to test is, in most countries, a **crime**. The skills you're building are legal precisely because of *where* you point them.

The boundaries to internalise

- **Only your own devices and apps.** Your phone, your emulator, apps you own or are explicitly authorised to test.
- **Unauthorised access is illegal.** Anti-hacking laws (the CFAA in the US, the Computer Misuse Act in the UK, and equivalents elsewhere) make accessing a system without authorisation an offence — even "just looking".
- **Authorisation must be explicit.** "I assumed it was fine" is not a defence. Get permission in writing before testing anything you don't own.
- **Respect data.** Even on systems you're allowed to test, don't exfiltrate, expose or keep other people's personal data.

Responsible disclosure

If you ever find a real vulnerability — in an app you're authorised to test, or through a bug-bounty program — the professional path is **responsible disclosure**: report it privately to the vendor, give them reasonable time to fix it, and don't publish or exploit it in the meantime. That's what separates a security professional from a criminal.

Tools

None here — this is the most important lesson in the course, and it's about judgement, not commands.

Method — the checklist before any test

1. Do I own this device/app, or have explicit written authorisation?
2. Is my testing confined to that scope?
3. If I find something serious, do I have a responsible-disclosure path?

Challenge 1: A friend asks you to "test the security" of an app on their phone. Write down the three things you'd confirm before touching it, and describe what you'd do if you found a real flaw.

The Android Permission Model

Lesson 13 · Core

Why permissions exist at all

You already know every app runs as its own **Linux UID** inside the sandbox. That isolation is the default — an app starts with almost no reach outside its own container. **Permissions** are the controlled exceptions: the only way an app gets to your camera, location, contacts or files is to *declare* the permission in its manifest and, for the sensitive ones, have you approve it at runtime.

The two kinds you must tell apart

- **Install-time (normal) permissions** — low-risk things like `INTERNET` or `VIBRATE`. Granted automatically when the app is installed; the user isn't prompted.
- **Runtime (dangerous) permissions** — anything that touches your private data or hardware: `CAMERA`, `ACCESS_FINE_LOCATION`, `READ_CONTACTS`, microphone. The app must ask *while running*, and you see the system dialog (Allow / Don't allow / only this time).
- **Special permissions** — a small set of high-power toggles (e.g. "Display over other apps", "All files access", accessibility) that live behind a dedicated Settings screen, not the normal dialog.

How modern Android tightened the model

- **Runtime prompts** (since Android 6) — dangerous permissions are no longer all-or-nothing at install.
- **Scoped storage** (Android 10+) — apps get their own media sandbox instead of free rein over `/sdcard`; broad file access is now a special permission.
- **One-time & "while using" grants** (Android 10/11+) — location and camera can be granted just for the current session, and unused permissions are auto-revoked.

For a tester this matters because an app's *declared* permissions (in the manifest) tell you its intended reach, while its *granted* permissions tell you what it can actually do right now. The two can differ.

Tools

- **Settings > Apps > [app] > Permissions** on the device.
- `adb shell` with the package manager `pm` and `dumpsys`.

Method

1. See the catalogue of dangerous permissions grouped: `adb shell pm list permissions -d -g`.
2. Inspect one app you own: `adb shell dumpsys package <pkg> | grep -i permission` — you'll see `requested permissions` vs `granted=true/false`.
3. Compare that with what the manifest declared (from the "Reading an APK" lesson: `aapt dump badging app.apk` lists the `uses-permission` lines).

Challenge 1: Pick one app you own and, without reading ahead, decide which of its permissions are install-time (normal) and which are runtime (dangerous). Then verify with `adb shell dumpsys package <pkg>`: which dangerous permissions are actually granted, and which were declared but denied?

Platform Defences — SELinux, Verified Boot & Play Protect

Lesson 14 · Foundations

The sandbox is only the first wall

You know that each app runs as its own Linux user. Modern Android stacks several more layers on top, so a single bug rarely gives full control of the device. Knowing these layers explains why rooting weakens a phone and why testers use emulators.

Layer	What it does	How to see it
SELinux	mandatory access control: even root processes are limited by policy (enforcing since Android 5.0)	`adb shell getenforce`
Verified Boot	checks that boot and system partitions are unmodified; the bootloader lock is part of it	`adb shell getprop ro.boot.verifiedbootstate`
Encryption	file-based encryption ties user data to the lock screen credential	`adb shell getprop ro.crypto.type`
Hardware keystore	keys live in a TEE or StrongBox chip and cannot be exported	app-level API (KeyStore)
seccomp	limits which kernel system calls app processes may use	enabled by the system
Play Protect	scans installed and sideloaded apps for known harmful behaviour	Play Store › Play Protect

Verified boot states are reported as colours: **green** (locked, official software), **yellow** (locked with a custom key), **orange** (unlocked bootloader), **red** (verification failed).

Method

1. On your emulator and — read-only — on your own phone, query SELinux mode, verified boot state and encryption type.
2. Compare the results and explain each difference.
3. Note them in your lab journal as “device baseline”.

Why this matters for app testing
Apps can ask the Play Integrity API whether the device looks genuine. That is why some apps behave differently on rooted devices or emulators — a defence you will meet again in the advanced courses.

Challenge 1: Run the three `getprop`/`getenforce` commands on your emulator. Which verified boot colour do you get, and why is it different from a store-bought phone?

Android Versions & Security Patch Levels

Lesson 15 · Foundations

Old software is the easiest target

Every month Google publishes the **Android Security Bulletin** listing fixed vulnerabilities with CVE numbers and severity. A device's **security patch level** (a date) tells you which of those fixes it contains. For testers and defenders, the patch level is the first number to check.

- **Android version** (e.g. 15, 16) and **API level** (SDK int) describe the platform; apps declare `minSdkVersion` and `targetSdkVersion`.
- **Patch level** `YYYY-MM-01` covers framework and system fixes of that bulletin; `YYYY-MM-05` additionally includes vendor and kernel components.
- **Project Mainline** (since Android 10) delivers some system components as “Google Play system updates”, independent of the vendor's firmware.
- Manufacturers publish their own support periods; once a device stops receiving updates, new public vulnerabilities stay open.

Tools

```
adb shell getprop ro.build.version.release        # Android version
adb shell getprop ro.build.version.sdk            # API level
adb shell getprop ro.build.version.security_patch # e.g. 2026-10-05
adb shell getprop ro.product.model
```

On the phone itself: Settings › About phone › Android version, and Settings › Security › updates.

Method — a patch audit of your own devices

1. Record model, Android version and patch level for each device you own.
2. Compare the patch date with today: more than a few months old means missing fixes.
3. Check for pending system and Google Play system updates and install them.
4. Devices that no longer get updates should not hold sensitive accounts — plan a replacement.

> **For app developers**
> A higher `targetSdkVersion` opts your app into newer security behaviour (for example cleartext traffic off by default, stricter exported rules). Google Play requires a recent target API level for new apps and updates.

> **Challenge 1:** Read the API level and patch level of your emulator image. If the patch level is `2025-06-05`, what does the “-05” tell you?

How Apps Talk to the Network — HTTPS & Network Security Config

Lesson 16 · Core

Every interesting app has a backend

Most apps are thin clients: the important data lives on a server reached over HTTPS. Android gives developers a declarative file, the **Network Security Configuration**, that controls which connections an app trusts. Reading it tells you a lot about an app before you ever intercept traffic.

- **Cleartext HTTP** is blocked by default for apps targeting Android 9 (API 28) or higher.

- **User-installed CA certificates** are not trusted by default for apps targeting Android 7.0 (API 24) or higher — only system CAs are.
- The config file lives in `res/xml/` and is referenced in the manifest with `android:networkSecurityConfig`.
- `<debug-overrides>` lets a developer trust extra CAs only in debuggable builds — the clean way to test your own app through a proxy.
- **Certificate pinning** (`<pin-set>`) restricts the app to specific server keys.

```
<network-security-config>
  <base-config cleartextTrafficPermitted="false" />
  <debug-overrides>
    <trust-anchors><certificates src="user" /></trust-anchors>
  </debug-overrides>
</network-security-config>
```

Tools

- `apktool d` — decode the manifest and `res/xml/network_security_config.xml`.
- An intercepting proxy (Burp Suite, mitmproxy, OWASP ZAP) — used in the advanced courses on your own debug builds.

Method — read the config of your own app

1. Decode the APK and find `android:networkSecurityConfig` in the manifest.
2. Check for `cleartextTrafficPermitted="true"` and which domains it applies to.
3. Check trust anchors: is `src="user"` present outside `<debug-overrides>`? That weakens release builds.
4. Note whether pinning is configured and when the pins expire.

> **Challenge 1:** A release build contains `<base-config cleartextTrafficPermitted="true">` with `<certificates src="user"/>`. Name both risks and the fix.

Harden Your Own Phone — a Settings Checklist

Lesson 17 · Defense

Apply what you learned to the device in your pocket

Security knowledge starts at home. The same settings that make a test device convenient — USB debugging, unknown sources, root — make a daily phone weaker. This checklist turns the course so far into concrete protection for your own Android phone.

Method — the checklist

1. **Screen lock**: a PIN of 6+ digits or a password. The lock credential also protects the encryption of your data.
2. **Updates**: install system updates and Google Play system updates promptly; check the security patch level.
3. **Developer Options off**: disable USB debugging and Developer Options when you are not developing; revoke USB debugging authorisations.
4. **App sources**: install from the Play Store or another trusted store; leave “Install unknown apps” disabled for browsers and messengers.

5. **Play Protect on**: keep app scanning enabled.
6. **Permissions review**: open the Permission manager and the Privacy dashboard (Android 12+); remove location, microphone, camera and SMS access that apps do not need. Keep “Pause app activity if unused” on so unused apps lose permissions.
7. **Accounts**: protect the Google account with a passkey or two-step verification and review its recovery options.
8. **Find my device and backup**: enable remote locate/erase and an encrypted backup.
9. **No root, locked bootloader** on the phone that holds your banking, mail and authenticator apps.

Tools

- `adb shell settings get global development_settings_enabled` — 1 means Developer Options are on.
- `adb shell settings get global adb_enabled` — 1 means USB debugging is on.
- `adb shell pm list packages -3` — every third-party app you installed; review the list for apps you no longer use.

Stronger options
Recent Android versions offer extra protection modes for high-risk users (for example Google's Advanced Protection), which enforce several of these settings together. Turn them on if your threat model needs it.

Separate lab and life
Do your testing on an emulator or a spare device. Mixing test tools and personal accounts on one phone is the most common beginner mistake.

Challenge 1: Run the checklist on your own phone. Which three settings did you change, and how many third-party apps did you uninstall?

OWASP MASVS & MASTG — the Testing Map

Lesson 18 · Foundations

The standard everyone in mobile security points to

When the advanced courses say a report is "MASVS-mapped", this is what they mean. **OWASP MAS** (Mobile Application Security) is the open, vendor-neutral project that defines *what* a secure mobile app should do and *how* to test it. Learning its vocabulary now means the hands-on courses — and real client reports — won't feel foreign.

The two documents, and how they fit together

- **MASVS** (Mobile Application Security *Verification Standard*) — the "what". A concise list of security requirements, grouped into categories such as **STORAGE** (how data is stored), **CRYPTO**, **AUTH** (authentication/session), **NETWORK** (transport security), **PLATFORM** (how the app uses Android platform features) and **CODE** & **RESILIENCE** (anti-tampering, hardening).
- **MASTG** (Mobile Application Security *Testing Guide*) — the "how". A large manual of concrete test cases and techniques, with discrete **MASTG tests** you can cite by id, covering both static and dynamic analysis on

Android.
- **MAS Checklist** — a spreadsheet that ties each MASVS requirement to the MASTG tests that verify it, so a report can show exactly what was checked.

Verification levels (the one thing people confuse)

- **L1** — baseline security every app should meet.
- **L2** — defence-in-depth for apps handling sensitive data (banking, health).
- **R** — resilience controls against reverse engineering and tampering, layered *on top* of L1/L2 (not a substitute for them).

You're not expected to memorise the standard — just to know it exists, what the category names mean, and that professional findings are written against it rather than invented ad hoc.

Tools

- The public OWASP MAS project pages (MASVS, MASTG, MAS Checklist) — free to read.
- Nothing to install; this is an orientation lesson.

Method

1. Skim the MASVS category list and say, in your own words, what each one protects.
2. Take a concern you've already met in this course and file it under the right category.
3. Note that every finding in the advanced courses will cite a MASVS category and, often, a MASTG test id.

> **Challenge 1:** Map these three concerns to a MASVS category each: (a) an app that writes a login token to a world-readable file in shared storage; (b) an app that requests the CONTACTS permission it never actually uses; (c) an app that doesn't validate the server's TLS certificate. One sentence of reasoning each.

Practice — a First Mini-Audit of Your Own App

Lesson 19 · Practice

Put the whole course together

You now know enough to perform a small, structured review of an app. Use an app **you built yourself** (a sample project from Android Studio is perfect) or an intentionally vulnerable training app, run it on your emulator, and produce a one-page result mapped to MASVS categories.

Tools

- `adb`, `aapt` or `apkanalyzer`, `apktool`, `apksigner`, `logcat`.
- `run-as com.example.app` — opens a shell as your app's user if the build is debuggable, to view its private files without rooting.

Method

1. **Identify**: package name, version, `minSdk`/`targetSdk`, signer digest (`apksigner verify --print-certs`).
2. **Manifest**: list permissions; check `android:debuggable`, `android:allowBackup` and the network security config.
3. **Components**: list exported activities, services, receivers and providers.

4. **Storage**: use the app, then look in `shared_prefs/`, `databases/` and `files/` for plain-text secrets.
5. **Logs**: watch `logcat` during login or data entry.
6. **Report**: one line per finding — title, evidence, MASVS category, risk (low/medium/high), recommendation.

```
adb shell run-as com.example.app ls -R /data/data/com.example.app
adb shell run-as com.example.app cat shared_prefs/settings.xml
aapt dump badging app.apk | head
```

Finding	MASVS	Fix
API token in `shared_prefs` in plain text	STORAGE	store in Android Keystore-backed encryption
Password written to logcat	STORAGE	remove log line, strip logs in release
Cleartext traffic allowed	NETWORK	disable in network security config
Unneeded exported activity	PLATFORM	set `exported="false"`

Challenge 1: Run steps 1–6 on a sample app you created. Produce at least four lines in your findings table, even if some say “no issue found”.

Where to Go Next

Lesson 20 · Next Step

You now have the foundation

You understand how Android is built — Linux kernel, ART runtime, the per-app sandbox — and you can set up a safe test environment: Developer Options, ADB, an emulator, and a first look inside an APK. That's exactly the groundwork the advanced Android courses assume. From here, the path goes from *understanding* the platform to *testing* it.

Your roadmap

- **Android Application Security** — the direct next step. Decompile real APKs with jadx/apktool, read insecure storage, intercept traffic, bypass SSL pinning, and hook running apps with Frida. Everything you set up here (ADB, package names, `/data/data`, the emulator) is the starting point of that course.
- **Mobile Penetration Testing** — the broader discipline: full assessments of mobile apps and their backends, mapped to OWASP MASVS/MASTG, with proper reporting.
- **Android Fuzzing** — going deeper into finding bugs automatically by feeding apps and components malformed input at scale.

Before you move on

Make sure you're comfortable running `adb devices`, opening an `adb shell`, listing packages, and unzipping an APK you own. If those feel natural, you're ready for Android Application Security. Keep the ethics lesson close — the advanced courses put real tools in your hands, and the boundary never changes: **only your own devices, apps and authorised targets**.

Final exam

Pass mark 70 %. Answers are in the Appendix: Solutions.

1. How many times do you tap Build number to unlock Developer Options?

A. 3
B. 10
C. 7
D. 5

2. Which Android component runs each app's compiled bytecode?

A. The Linux kernel
B. SELinux
C. The bootloader
D. ART (Android Runtime)

3. After 'adb shell', what Linux user are you usually running as?

A. shell (an unprivileged user)
B. adb
C. system
D. root

4. Which set correctly lists the five parts inside an APK?

A. Info.plist, code.bin, images/, libs/, data.db
B. AndroidManifest.xml, classes.dex, res/, lib/, resources.arsc
C. manifest.json, main.dex, assets/, bin/, strings.xml
D. package.xml, app.dex, drawable/, native/, table.arsc

5. Which download gives you the adb tool without the full Android Studio?

A. AVD Manager
B. Gradle
C. build-tools
D. platform-tools

6. On Linux and Android, which UID belongs to the all-powerful root account?

A. 500
B. 1
C. 1000
D. 0

7. What is /sdcard (also /storage/emulated/0)?

A. An app's private data directory
B. The read-only OS partition
C. Shared storage readable across apps
D. The Linux kernel image

8. Which is a real risk of rooting a phone you depend on?

A. It extends the manufacturer warranty
B. It strengthens the app sandbox
C. It speeds up OTA updates
D. It commonly breaks OTA updates and can leave you unpatched

9. Which logcat option dumps the current log and exits?

A. -b crash
B. -v time
C. -d
D. -c

10. Apps targeting Android 12+ must explicitly declare which attribute on components with intent filters?

A. android:exported
B. android:allowBackup
C. android:debuggable
D. android:largeHeap

11. An APK update is signed with a different certificate than the installed app. What happens?

A. It installs and keeps the data
B. Play Protect re-signs it
C. It installs as a second copy automatically
D. Android rejects the update

12. What does the '-05' patch level suffix include beyond '-01'?

A. Only Play Store fixes
B. Vendor and kernel component fixes
C. Nothing extra
D. Emulator fixes

13. Which statement about user-installed CA certificates is correct for apps targeting API 24+?

A. They are not trusted by default unless the app's network security config allows them
B. They disable pinning
C. They replace system CAs
D. They are always trusted

14. Which command reports whether SELinux is enforcing?

A. apksigner verify
B. adb logcat *:E
C. adb shell getenforce
D. adb shell pm path