SYNTAX COLLEGE

5 DAYS · 17 HANDS-ON LABS · 75% HANDS-ON

Android Internals Deep Dive

From silicon to APK: hardware and boot, ROM and Verified Boot, Zygote and System Server, Binder from framework to kernel, JNI and native device access, SELinux – and a capstone malware hunt with JADX and Frida.

DURATION	HANDS-ON	PROJECTS	LEVEL
5 days · ~45 hours	~75% · 17 labs + bonus drills	5 starter + 5 finished	Advanced / technical

01 About the course

Each day answers one question about the system – **what's inside**, **how it runs**, **how it communicates**, **what happens below the framework**, and **how it's protected** – and builds directly on the previous one. Theory is kept to about a quarter of the time; the rest is investigation on emulators and real devices, AOSP code tracing, and coding in Kotlin, C++ and AIDL.

By Day 5 participants can explore any layer of the device, trace a request from an app through Binder into the kernel and down to the hardware, write native code that talks to device nodes directly, diagnose security denials, and reverse-engineer and instrument a suspicious app.

Who should attend

- Platform and BSP engineers at device, automotive (Android Automotive) and semiconductor companies
- Security researchers and reverse engineers at cyber and digital-forensics companies
- Senior Android developers who need to work below the SDK
- ROM developers and SoC integrators

What participants will be able to do

- Explore hardware, init and a factory image from the shell
- Verify partitions and the boot chain (AVB, dm-verity)
- Trace Zygote, System Server and ActivityManager in AOSP
- Cross boundaries: hidden APIs, service call, JNI, your own AIDL service
- Follow Binder into the kernel and read /dev/input from C++
- Diagnose DAC / SELinux / capability denials; hunt malware with JADX and Frida

02 Day-by-day syllabus

DAY 1 What's inside – hardware, boot and firmware

Participants leave knowing how the kernel sees hardware, how init assembles the system, and what a factory image contains.

- AOSP vs GMS and how responsibility is split among device makers
- The Android stack: hardware, kernel, HAL, native, framework, apps
- Boot overview with emphasis on Zygote
- ROM layout: `boot.img`, `system.img`, `vendor.img`, `vbmeta.img`

HANDS-ON LABS

- **Hardware Investigation** (~1h · investigation)
  Exploring how the kernel exposes hardware in userspace through the Device Tree and /sys.
- **Init Process Investigation** (~1.5–2h · investigation)
  Tracing init.rc triggers and actions and identifying the core services init starts.
- **ROM Extraction & Forensic Analysis** (~2h · investigation)
  Downloading a factory image, extracting boot/system/vendor/vbmeta and inspecting key files and red flags.

> **Bonus drill:** Search the vendor partition for suspicious patterns – hard-coded IPs, root services, open permissions.

DAY 2 How it runs – storage, boot security, Zygote and System Server

Participants leave knowing how partitions are organized and verified, how Zygote creates every process, and how ActivityManager orchestrates the app lifecycle.

- AOSP walk-through: Bionic, system, framework, Soong
- Zygote: fork(), ART loading and JNI registration
- System Server: bootstrap, core and optional services; Activity Manager
- Bootloader, kernel and init; dm-verity and Verified Boot
- Project Treble, Mainline and OTA

HANDS-ON LABS

- **Storage & Filesystem Investigation** (~45m · investigation)
  Mapping partitions with ADB: /dev/block, dm-* devices, fstab and mount points.
- **Verified Boot & dm-verity** (~1h · investigation)
  Inspecting AVB and vbmeta with avbtool and identifying the dm-verity configuration.
- **Zygote Code Tracing** (~1–1.5h · AOSP code tracing)
  Following fork() and specialization through AOSP to the birth of an app process.
- **ActivityManager AOSP Digging** (~1–1.5h · AOSP code tracing)
  Reading AMS internals: process records, lifecycle orchestration and process management.

> **Bonus drill:** Trace the FUSE layer – the same file through /storage/emulated/0 and /data/media/0, comparing owners and permissions.

DAY 3 How it communicates – Reflection, Binder, JNI and AIDL

Participants see Binder from every angle – from Java, from the terminal, from native C++, and as service authors.

- Binder framework: Proxy, Stub, Parcel, ServiceManager
- JNI and NDK: JNIEnv, memory between ART and native, building .so libraries
- Reflection and hidden APIs
- AIDL interfaces and generated Stub/Proxy
- ART: AOT/JIT, memory and GC

HANDS-ON LABS

- **Reflection & Hidden API** (~1–1.5h · coding · Kotlin)
  Using reflection to reach hidden fields and ActivityManager methods – and seeing what the system blocks.
- **Service Call – bypassing the SDK** (~1–1.5h · investigation)
  Decoding AIDL from AOSP and calling system services directly from the terminal with `service call`.
- **JNI Bridge & Native Access (BinderProbe)** (~1.5–2h · coding · C++)
  A native NDK component that calls internal system APIs – local calls vs cross-process Binder calls.
- **AIDL Client–Server** (~1.5–2h · coding · Kotlin/AIDL)
  Building your own Binder service between two apps and passing complex data over IPC.

> **Bonus drill:** Script a scan of all services with transaction code 1; measure JNI marshalling overhead vs Binder overhead.

DAY 4 Going deeper – kernel Binder, a full hardware stack, native devices

Participants leave knowing how Binder works in the kernel, how a real hardware stack flows from app to chip, and how to read kernel device nodes directly.

- Binder driver: /dev/binder, ioctl BINDER_WRITE_READ, BC_/BR_TRANSACTION, nodes, refs, thread pools
- Parcels and Binder objects across processes
- The GPS full-stack flow: driver → AIDL HAL → JNI → LocationManager
- Android kernel vs Linux kernel: Binder, Ashmem, power management, /sys and /dev

HANDS-ON LABS

- **Binder Deep Investigation** (~2h · investigation)
  Reading Binder state in the kernel: nodes, references, the transaction stack and thread pools.
- **GPS Full-Stack Trace** (~1.5–2h · AOSP code tracing)
  Following a location request from the app through the framework, JNI and the AIDL HAL down to the driver.
- **Native Input Device Probe** (~2h · coding · C++)
  NDK code that reads raw events directly from /dev/input – bypassing the entire Android framework.

Bonus drill: Live-trace GPS Binder transactions; implement gesture recognition from raw touch events.

DAY 5 How it's protected – permissions, SELinux and reverse engineering

Participants leave able to diagnose which security layer blocks an access and to reverse-engineer and dynamically instrument a suspicious app.

- The app sandbox vs the classic Linux user model
- DAC (UID/GID) vs MAC – SELinux enforcing/permissive, policies and contexts
- Linux capabilities and privilege-escalation prevention
- Runtime and signature permissions; Package Manager Service

HANDS-ON LABS

- **The Permissions Detective** (~1–1.5h · investigation)
  Diagnosing whether a denial comes from DAC, an SELinux AVC, or a missing capability.
- **Process Security Inspection** (~1–1.5h · investigation)
  Mapping security contexts, UIDs and capabilities of system processes vs third-party apps (ps -AZ, id).
- **The Ultimate Malware Hunt – capstone** (~2–2.5h · JADX + Frida)
  Static and dynamic analysis of a suspicious lab app: exposed components, reflection and JNI, root-detection bypass, tracing data to the exfiltration point.

Bonus drill: Categorize AVC denials by resource type; trace netd's capability lifecycle from init.rc to runtime.

Optional bonus modules – can extend the course or replace parts of a day

- **Android application from the inside** – components, intents, manifest, permissions, location, storage and FileProvider, with do-it-yourself ADB labs (aapt, granting/revoking permissions, pulling internal storage).
- **ADB in depth** – screen recording and screenshots, file transfer, driving the camera, sending SMS and navigation from the shell, heap and per-process memory inspection.
- **NDK without Android Studio** – a native calculator app, building and running a native CLI tool for the emulator's CPU architecture, and the NDK stable APIs.
- **Kernel boot & system diagnostics** – dmesg boot analysis, strace on Zygote's fork(), system properties, and post-mortem crash and stack-trace analysis.
- **Android kernel additions** – Ashmem, ION, wakelocks, Low Memory Killer, logger, alarms – and Binder security tokens with examples from the framework.

- **The camera case & System UI** – tracing a camera capture from the tap to the driver, and how the status bar, navigation bar and keyguard work.
- **Vertical integration** – writing a small kernel driver, connecting it to a C++ HAL, exposing it through JNI and calling it from the framework.
- **Dynamic instrumentation with Frida** – extended hooking labs beyond the capstone.

17	9	3	4	1	~25h
hands-on labs	investigation labs	AOSP code-tracing labs	coding labs – Kotlin, C++, AIDL	capstone – JADX + Frida	hands-on time

What participants receive

- Step-by-step lab guides with hints and full solutions
- 5 starter + 5 finished Android Studio projects (incl. C++/NDK)
- Presentations and per-day notes
- Glossary and an ADB / shell command reference
- Purpose-built lab apps (incl. a suspicious app for the capstone)
- Runs on x86 emulators; real-device demo (Pixel) included

03 Format & logistics

Sessions	5 full days
Hours	09:00–16:30 incl. breaks
Method	25% theory · 75% labs
Dates	Coordinated with your team

Requirements

- A training room with a projector or large screen
- A laptop per participant with Android Studio installed (setup guide sent in advance)
- Android devices or emulators; USB debugging enabled
- Up to 15 participants per group
- Recommended: a rootable test device or userdebug emulator image

Topics can be weighted toward your stack – e.g. automotive (AAOS/VHAL), OTA and Verified Boot, or security research.

04 Your instructor

All Syntax College courses are designed and taught personally by **Eran Katsav**, founder of Syntax College. Eran holds a B.Sc. in Computer Science from the Hebrew University of Jerusalem and an M.Sc. with honors from Tel Aviv University, and has more than 20 years of experience training engineers in industry, the defense sector and academia.

He lectures on native Android development in Kotlin and Java at Reichman University and at the Holon Institute of Technology (HIT), where he also heads the mobile lab. As an independent developer he created **Bip it**, the first Hebrew voice assistant, which reached about one million downloads in Israel; white-label versions of its engine ship in all Suzuki vehicles sold in Israel.

Selected clients: **Samsung, General Motors, Cisco, ironSource, CWT, Ophir Optronics**. Authorized supplier of the Israeli Ministry of Defense.

Bring this course to your team
Tailored syllabus, on-site at your offices, in Hebrew or English.

Eran Katsav
+972-50-6944922
info@syntax.org.il
www.syntax.org.il