Android Architecture Explained (For Hackers, Pentesters & Bug Bounty Hunters)

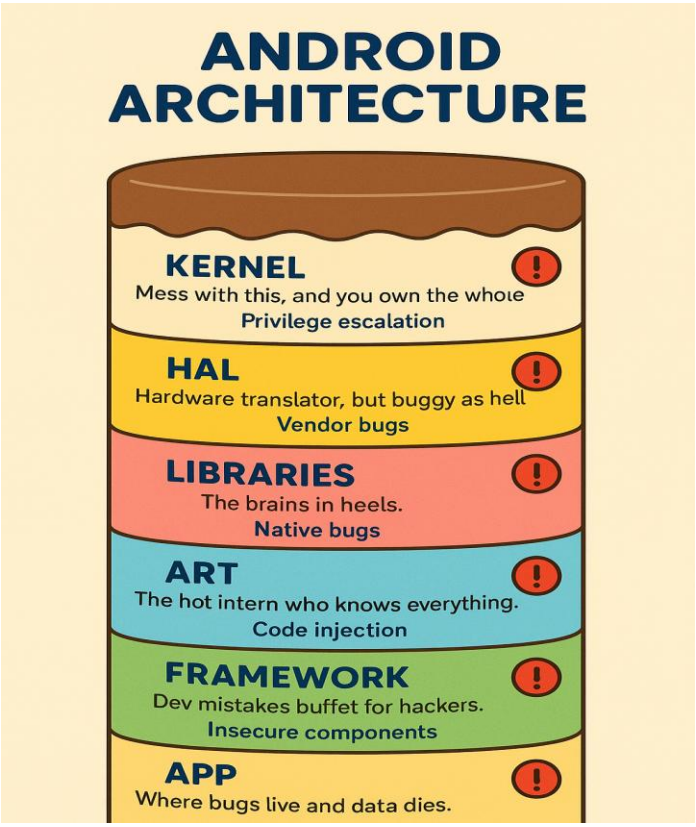
So What Is Android Architecture, Really?

Imagine Android like a cake—layered, messy, and way too tempting. Each layer has its own role, but together? They make your phone deliciously functional.

But here's the twist: for a hacker, **each layer is also an attack surface.**

"Know your target. Own your target."

If you're hunting bugs in Android or testing app security, you need to know exactly where to poke—and what breaks when you do. Android isn't just an app playground. It's a stack of systems, each with attack surfaces, misconfigurations, and juicy opportunities.



Let's go layer by layer—not like a developer, but like a hacker who wants root.

 # Layer 1: Linux Kernel – The Attack Surface Base Layer

"You don't pop apps—you pop the kernel."

Think of this as Android's... uh, underwear. You don’t see it, but everything depends on it being solid. This is the foundation of Android. The Linux kernel is the heart of the Android OS.It’s a modified Linux kernel that:

--Talks to your device hardware (camera, mic, screen, etc.)

--Handles memory, security, and power management

--Does scheduling and multitasking

--Includes the Binder IPC – Android’s way of letting apps and services talk to each other securely

Key Features:

• **Memory Management**: Isolates app processes and allocates resources.

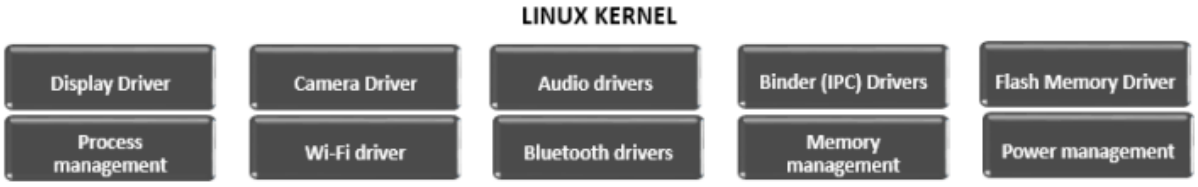
• **Process Scheduling**: Balances CPU time among processes.

• **Device Drivers**: For camera, audio, display, and sensors.

• **Power Management**: Maximizes battery life.

• **Binder IPC**: Fast inter-process communication for Android services.

• **Security**: Implements SELinux, ASLR, stack protection, and more.



Why it matters for security :Kernel bugs = king-level exploits. Privilege escalation, rooting, or bypassing SELinux often start here.

→Pentest interest:

-- The first line of defense.

-- The last layer before full control.

→Bug bounty interest:

-- Look for outdated drivers, unpatched kernel CVEs, or vendor-modified kernel code. Exploits here often lead to full device compromise.

-- Privilege escalation via kernel exploits (e.g., dirty cow, binder overflow)

-- Custom driver vulnerabilities (e.g., OEMs write shitty code)

-- Kernel bugs = root access. Vendors often patch late, so driver CVEs are your backdoor

--SELinux misconfigurations (check permissive modes, audit logs)

Layer 2: Hardware Abstraction Layer (HAL)– The Freaky Middleman

“If it touches hardware, you can fuzz it.”

HAL is the flirty middle-person who speaks both human and robot.

It sits between the kernel and higher levels. HAL lets Android OS talk to hardware without caring what phone you’re on. The Hardware Abstraction Layer (HAL) connects the Android OS to hardware, acts as a translator between the Android OS and the device hardware. Often written in native C/C++, and not always audited well. HAL is a collection of C/C++ modules that talk to the kernel drivers (camera, GPS, Wi-Fi, sensors) and expose them to higher layers.

HAL Types:

• **Binderized HAL**: Preferred, uses Binder IPC and runs in its own process.

• **Passthrough HA**L: Directly loaded into the system server; less secure.

• **Manifest HAL**: Declared in device manifests; ensures version compatibility.

Key HALs:

• Audio HAL

• Camera HAL

• Sensor HAL

• Graphics HAL

• Radio, Wi-Fi, Bluetooth HALs

HALs are defined using HIDL (deprecated) or AIDL (modern, Kotlin/Java compatible).

Why it matters for security:

HAL runs with system-level privileges if you exploit a buggy HAL module, you can escalate privileges.

HAL is why app developers don’t need to code differently for every device on the planet.

→**Pentest interest:** Fuzz the binder interfaces, analyze HIDL/AIDL files, and check vendor partitions.

OEMs love writing sloppy HAL code → prime hunting ground for privilege escalation.

→ **Bug bounty interest:**

-- Stack overflows in poorly written vendor HALs

-- Direct attack on binderized HAL interfaces via fuzzing

-- Legacy passthrough HALs with no IPC boundaries

-- Many OEMs add custom HALs (for fingerprint, sensors, etc.) → custom code = custom bugs. Look for unsafe memory handling, buffer overflows in vendor HALs.

Layer 3: Native Libraries Layer - The Brains in Heels

"The moment you see C code, smell the buffer overflows."

This layer includes: libc, libmedia, OpenGL, WebKit, SQLite, etc.

This layer offers optimized C/C++ libraries for performance-critical features.

Core Libraries: These libraries do all the heavy lifting:

• **libc:** Standard C library

• **Surface Manager:** Display composition

• **OpenGL ES:** 2D/3D graphics rendering

• **Media Framework**: Audio/video codecs

• **SQLite:** Embedded database engine

Security Libraries:

• **Keystore:** Secure hardware-backed key storage

• **DRM:** Digital rights management

• **Biometrics:** Fingerprint, face unlock

Why it matters for security:

These libs are often targets for memory corruption bugs (use-after-free, buffer overflow).

These are the libraries that make apps fast AF. No waiting for your filters or videos to load thus contain bugs.

→Bug bounty interest:

--Exploitable bugs in media stack (e.g., libstagefright RCEs)

--SQLite injection from untrusted data (e.g., malicious clipboard, NFC) , SQLite injection via apps using unpatched SQLite.

--WebView abuses (file:// access, XSS from local HTML)

--Vulnerabilities in media codecs (libstagefright CVEs let attackers exploit video/audio parsing).

4. Layer 4: Android Runtime (ART) - The Hot Intern Who Knows Everything

"It's not just Java. It's compiled DEX → native → your payload."

ART replaced the Dalvik VM and significantly improved performance and battery life.

Feature	**Dalvik**	**ART**
Compilation	JIT (runtime)	AOT (install time)
App Performance	Slower	Faster
Boot Time	Faster	Slower initially
Battery Consumption	Higher	Lower
Garbage Collection	Basic	Concurrent optimized

ART compiles apps using AOT and sometimes JIT. It's also where garbage collection and app isolation live. ART is what actually runs your apps, it runs them fast. Originally we had Dalvik (RIP), but now it's all about ART:

--Compiles apps ahead of time (AOT) into native code

--Reduces startup time and boosts battery life

Features badass garbage collection to clean up memory

ART Workflow:

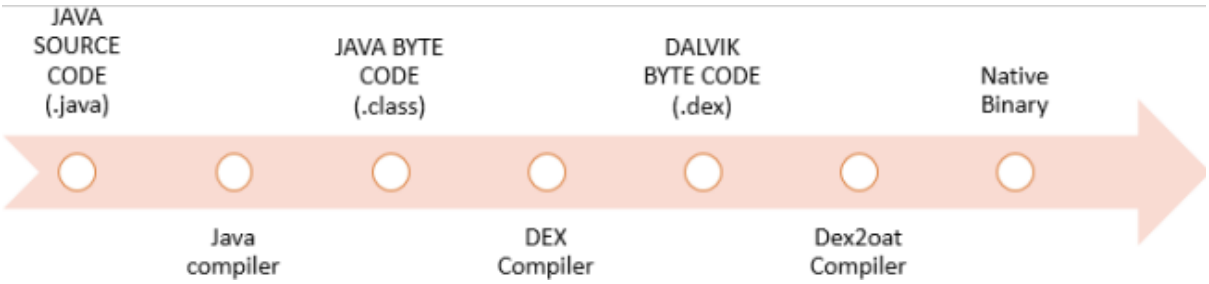
1. Java source code is compiled by java compiler to java byte code.

2.App code is compiled into DEX bytecode.

3. dex2oat converts it to native code.

4. ELF binaries are created for execution.

5. Profile-Guided Optimization (PGO) tunes performance over time.



Why it matters for security:

ART enforces sandboxing — apps can't directly touch each other. But if you break out (e.g., via Binder or deserialization bugs), you're in business.

ART enforces sandboxing, but if you break out—game over.

→Bug bounty interest:

--Misuse of reflection and dynamic code loading

--Tampering with DEX files pre-compilation (supply chain)

--dex2oat parsing bugs

--Look for unsafe serialization/deserialization (classic attack surface).

--Tampering with compiled code (dex2oat, OAT files).

--Detecting or bypassing obfuscation/protection (ProGuard/R8).

Pro tip: Analyze the app bytecode with Jadx, APKTool, and reverse-engineer code flow. Look for unsafe class loading, obfuscation holes, or debug flags left on.

Fun fact: ART errors leave juicy stack traces in logcat — great intel for bug hunters.

Layer 5: Application Framework Layer - Where the API Gold Lies

"This is where app logic lives. And devs love trusting user input."

This is where Android provides Java/Kotlin APIs for developers.

This is the layer developers use to build apps—clean, powerful APIs.

Core System Services:

•**Activity Manager:** App lifecycle and task stack management

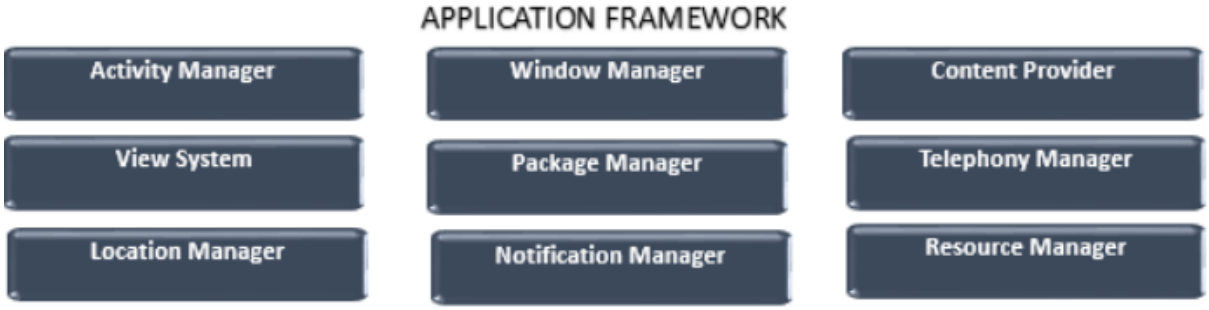
•**Window Manager**: UI rendering, screen orientation

•**Content Providers:** Secure inter-app data sharing

•**Package Manager:** Install/remove apps, manage permissions

•**Notification Manager:** Push and local notifications

•**Location Manager:** GPS and network-based positioning



Why it matters for security:

--Attack Surface:

--Exported Activities/Services → privilege escalation.

--Content Providers → SQL injection, path traversal, unintended data leaks.

--Intents → Intent spoofing / privilege confusion.

→Bug bounty interest:

--Content Provider injection or path traversal

--Insecure exported components (android:exported="true" with no permissions)

--Intent hijacking / intent sniffing

--Leaking private data via improperly protected APIs

-- Check AndroidManifest.xml for exported components, weak permission checks, and misused IPC.

Tools: adb shell, drozer, Objection, Frida, custom broadcast scripts

Layer 6: Application Layer – The Battlefield

“Where bugs live and data dies.”

This is where you’ll: Reverse apps**,** Look for hardcoded secrets**,** Exploit insecure storage, Abuse WebView

At the top are user-facing applications, including both system and third-party apps. All user/system apps (Instagram, WhatsApp, Google Maps, etc.) that sit at the top. These are the apps you open, tap, swipe, and delete when you’re mad. Installable files in Android are called Android application package (APK) files. APK files are nothing but ZIP files that are based on the JAR file format.

App Types:

•**System Apps:** Phone, Contacts, Settings

•**User Apps:** Installed via Google Play or sideloaded

•**Widgets:** Home screen components and shortcuts

Android App Components:

•**Activities:** UI screens (e.g., LoginActivity)

•**Services:** Background tasks (e.g., media playback)

•**Broadcast Receivers**: System event handlers (e.g., battery low)

•**Content Providers:** Structured app data interfaces

Why it matters for security:

This is the part people judge Android by. If the app layer sucks, the whole experience sucks—even if the backend is pure gold.

→**for pentesters:**

Common issues:

--Insecure storage (sensitive data in SharedPreferences or SQLite without encryption).

--Hardcoded API keys.

--Insecure WebViews (XSS, JavaScript bridges).

--Weak crypto (bad use of AES/ECB or no TLS pinning).

→**Bug bounty interest:**

--Debuggable apps (android:debuggable="true")

--WebView with JS enabled + no domain whitelist

--Insecure local storage (SharedPrefs, SQLite, internal files)

--Misconfigured permissions, clipboard leakage

-- Finding data leaks, insecure exports, or bypassing authentication logic.

Bonus: Watch for apps using root, unsafe libraries, or ignoring SSL pinning.

Final Word

Android's architecture isn't just a bunch of boring layers—it's a sexy stack of powerful systems working together to give you fast, secure, battery-efficient experiences. From the tight Linux base to the flashy app layer, everything is optimized for smooth performance even if it's rocking low-end hardware.

If you're bug hunting:

- **Kernel = full control**
- **HAL = privilege escalation**
- **Libraries = memory corruption playground**
- **ART = code execution games**
- **Framework = dev mistakes buffet**
- **Apps = the jackpot of secrets**

Why This Matters for Pentesters & Bug Hunters

- **Performance vs. Exploitation:** Understanding ART, HAL, and kernel surfaces tells you *where exploits live*.
- **Secure Design vs. Weak Configs:** Knowing about Intents, Content Providers, and permissions shows you *where devs screw up*.
- **Debugging:** Logs, stack traces, and decompiled artifacts all trace back to architecture knowledge.
- **Modern App Testing:** Jetpack components (ViewModel, LiveData, Room) bring new attack vectors if misused.

 # Bug Hunter's Glossary (Android Edition)

Here's your **20-term survival kit** — explained for pentesters:

1. **AOT (Ahead-of-Time):** Compiles DEX bytecode to machine code before runtime. Faster execution, but artifacts can leak sensitive code in decompiled OAT files.
2. **JIT (Just-in-Time):** Compiles methods on the fly. Useful for profiling attacks or detecting anti-debugging tricks.
3. **dex2oat:** ART's compiler, converts DEX → OAT/ELF files. Sometimes leaves traces of obfuscation bypassed code.
4. **Binder:** Android's IPC backbone. Target for privilege escalation via malformed transactions.
5. **AIDL:** Defines Binder APIs. Poorly designed AIDL = attack vector (e.g., missing permission checks).
6. **SELinux:** Mandatory access control. Exploit developers look for SELinux policy misconfigurations to escape sandboxes.

7. **GKI (Generic Kernel Image):** A standardized kernel image. Means kernel bugs are more uniform = bigger bounty if you find one.
8. **System Server:** Hosts critical services (ActivityManager, PackageManager). Exploiting it = system compromise.
9. **NDK (Native Dev Kit):** Lets apps use C/C++. Native bugs here are juicy (memory corruption, RCE).
10. **JNI (Java Native Interface):** Bridge between Java ↔ C/C++. Memory unsafety in JNI code is a big bug class.
11. **ViewModel:** Holds UI state. Not directly security-critical but mishandling data here can leak sensitive info.
12. **LiveData:** Observables for UI. Sometimes abused for data exfiltration if not scoped.
13. **Room:** SQLite wrapper. SQL injection vulnerabilities lurk if developers use raw queries unsafely.
14. **WorkManager:** Background job scheduler. Misconfigurations could leak background tasks or expose logic flaws.
15. **ProGuard/R8:** Code obfuscators. As a hunter, you'll often deobfuscate to reverse-engineer apps.
16. **WebView:** Embedded browser component. A hotspot for XSS, JavaScript injection, file scheme abuse.
17. **Keystore:** Secure hardware-backed key storage. Weak use (exported keys) = bounty.
18. **Intents:** Messaging objects between components. Intent spoofing/broadcast hijacking = classic vuln.
19. **Content Providers:** Data-sharing components. SQLi, path traversal, data leakage = top targets.
20. **APK Signing (v2/v3 schemes):** Ensures APK integrity. Misuse or downgrade attacks can be exploited.

BY - BITHOWL