Android Internals

Security and Porting



AOSP | Build & Porting | Security | Boot & OTA | Reverse Engineering

Android Internals

Security and Porting

A Systems Approach

Regan Russell

Baseline: AOSP `android17-release` (`CP2A.260605.016`)
Kernel: ACK `android17-6.18`

Android Internals: Security and Porting
Copyright © 2026 Regan Russell. All rights reserved.

First edition, typeset November 19, 2026.

Android is a trademark of Google LLC. The Android robot is reproduced or modified from work created and shared by Google and used according to terms described in the Creative Commons 3.0 Attribution License.

Each chapter is anchored in the pinned AOSP source tree, its git history, and experiments on reference devices. Device observations are tagged `[OBSERVED device]`; source claims are tagged `[SRC: path]` and listed under each chapter's *Source anchors*.

Contents

I Foundations **1**

1 Origins and Architecture **2**
1.1 From a camera company to a release model . . . 2
1.2 A history written by `git log` . . . 3
1.3 The layer cake . . . 4
1.4 The cut across the layers: Treble . . . 4
1.5 Three update channels, three trust anchors . . . 5
1.6 What Android 17 adds, and where this book meets it . . . 6
1.7 The reference platforms, and how to read this book . . . 7
Source anchors . . . 7
Experiments . . . 7

2 Getting, Reading and Building the Source **9**
2.1 The checkout: `repo`, manifests, and the superproject . . . 9
2.2 What "this build" actually contains: release configs and flags . . . 10
2.3 The build system: Soong, the Make remnant, and Kleaf . . . 11
2.4 Reading the tree at scale . . . 12
2.5 Building — and building on a small machine . . . 12
2.6 Worked example: resolving a flag end to end . . . 13
Source anchors . . . 13
Experiments . . . 14

3 The Android Linux Kernel **15**
3.1 Two trees, and a branch name that encodes a contract . . . 15
3.2 GKI: one kernel binary, vendor code as modules . . . 16
3.3 KMI: the stable ABI, and how it shrank in Android 17 . . . 16
3.4 Vendor hooks: sanctioned code inside core paths . . . 17
3.5 The Android-specific kernel . . . 18
3.6 How Android checks the kernel: from config fragments to certification . . . 18
3.7 The reference kernels . . . 19
Source anchors . . . 19
Experiments . . . 19

II Boot and the Native World **21**

4 Booting: Bootloader to Kernel **22**

4.1 The boot image family . . . 22
4.2 libavb: what "verified" actually means . . . 23
4.3 Lock state and the boot colours . . . 24
4.4 Why lying in userspace doesn't help: the root of trust reaches KeyMint . . . 24
4.5 Who may unlock . . . 25
4.6 Seeing it fail: the tamper experiments . . . 25
4.7 A/B, and when a boot counts as "good" . . . 26
Source anchors . . . 26
Experiments . . . 27

5 init **28**
5.1 One binary, five programs . . . 28
5.2 Boot as three `exec`s, and a domain at each . . . 28
5.3 First stage: a minimal world, and the verified partitions . . . 29
5.4 Second stage: properties, namespaces, and the `.rc` triggers . . . 30
5.5 The Treble boundary inside init: `vendor_init` . . . 30
5.6 The property service: three rules . . . 30
5.7 No service runs as `init` . . . 31
5.8 `ueventd`: where DAC gives way to MAC . . . 31
5.9 What changed in Android 17: fail-closed policy loading . . . 32
Source anchors . . . 32
Experiments . . . 32

6 Bionic and the Dynamic Linker **34**
6.1 Bionic: a libc built for the platform . . . 34
6.2 The linker, and configuration generated at boot . . . 35
6.3 The namespace graph: isolation enforced at `dlopen` . . . 35
6.4 Apps and the public-library contract . . . 36
6.5 The targetSdk ratchet in the linker . . . 36
6.6 W^X for app code: what the linker and SELinux do, and what they leave open . . . 37
Source anchors . . . 38
Experiments . . . 38

7 Native Daemons and System Plumbing **39**
7.1 The shape of a privileged daemon . . . 39
7.2 `logd`: everyone writes, few read everything . . . 40
7.3 The crash pipeline: a "ptrace-anything" helper, safely caged . . . 40
7.4 `lmkd`: deciding what to kill, and killing it cleanly . . . 41
7.5 Task profiles and cgroups: policy by name . . . 42
7.6 What Android 17 changes here . . . 42
Source anchors . . . 42
Experiments . . . 43

III IPC and the Runtime 44

8 Binder 45
8.1 The driver model . . . 45
8.2 Two drivers, one ABI . . . 46
8.3 The kernel security surface . . . 47
8.4 Identity: who is calling? . . . 47
8.5 `servicemanager`, and the service as the last line of defence . . . 48
8.6 The bindings, AIDL, and the end of HIDL . . . 48
8.7 Android 17: hardening the wire . . . 49
Source anchors . . . 49
Experiments . . . 50

9 Zygote and Process Creation 51
9.1 The zygote idea . . . 51
9.2 From the window manager to `fork()` . . . 51
9.3 The fork-server protocol and its three gates . . . 52
9.4 Two fork paths . . . 52
9.5 `ForkCommon`: the fork, and what the child must reset . . . 53
9.6 `SpecializeCommon`: a clone becomes an app, in order . . . 53
9.7 What the zygote model costs: shared address space . . . 54
9.8 New in Android 17: SafeSetID for app-zygote UID transitions . . . 54
9.9 Back into Java, and home again . . . 55
Source anchors . . . 55
Experiments . . . 55

10 DEX and ART 57
10.1 DEX: the format ART runs, and the checksum that proves nothing . . . 57
10.2 The compilation model, and isolating the compiler . . . 58
10.3 `odsign`: a signing authority that destroys its own key . . . 58
10.4 Running code without RWX . . . 59
10.5 Hidden API enforcement . . . 59
10.6 Dynamic code loading: the read-only rule, and what it leaves open . . . 60
Source anchors . . . 60
Experiments . . . 61

IV The Framework 62

11 system_server 63
11.1 Birth: identity and capability . . . 63
11.2 Android 17: policing UID transitions with SafeSetID . . . 64
11.3 Startup, in phases, ordered for security . . . 64
11.4 When it dies: the Watchdog and the soft reboot . . . 65
11.5 Why `system_server` is the prime target — and what still holds it . . . 65
Source anchors . . . 66

Experiments . . . 66

12 Activities, Processes and Windows **68**
12.1 Process importance: the OOM-adjustment scale . . . 68
12.2 The app freezer: starving the background . . . 69
12.3 Background Activity Launch control . . . 69
12.4 Borrowed identity: intents and PendingIntents in Android 17 . . . 70
12.5 Windows, touches, and the tapjacking boundary . . . 71
12.6 A worked example: audio follows importance . . . 71
Source anchors . . . 71
Experiments . . . 72

13 Packages and Installation **73**
13.1 Signature schemes, newest first . . . 73
13.2 Why "newest first" is not a downgrade hole . . . 73
13.3 Android 17: hybrid post-quantum signing (v3.2) . . . 74
13.4 Rotation, upgrades, and the signature as SELinux input . . . 75
13.5 The install pipeline . . . 75
13.6 App data: DAC and MAC isolation, set together . . . 75
13.7 APEX: anchoring Mainline trust in the factory image . . . 76
Source anchors . . . 76
Experiments . . . 77

V Security **78**

14 The Android Security Model **79**
14.1 The threat model, in one paragraph . . . 79
14.2 Identities: the UID space . . . 79
14.3 Permissions: what the platform actually declares . . . 80
14.4 From a permission to enforcement: three layers, one example each . . . 80
14.5 Install-time gates . . . 81
14.6 `fs_config` and the privileged-binary inventory . . . 82
14.7 Defence in depth, assembled . . . 82
14.8 What the rest of Part V examines . . . 83
Source anchors . . . 83
Experiments . . . 83

15 SELinux on Android **85**
15.1 Why MAC, and the shape of the policy . . . 85
15.2 Compiling and loading the policy at boot . . . 85
15.3 How an app arrives in its domain . . . 86
15.4 What the categories actually enforce . . . 87
15.5 Android 17's tier: the end of ashmem for new apps . . . 87
15.6 Where `neverallow` is really enforced . . . 88
Source anchors . . . 88

Experiments . . . 89

16 Kernel Hardening and Exploit Mitigations 90
16.1 A note on method: resolving the config without a device . . . 90
16.2 What GKI 6.18 builds in — and leaves out . . . 90
16.3 Seccomp: which syscalls an app may attempt . . . 91
16.4 Where the "allowed" dangerous syscalls are stopped . . . 92
16.5 Userspace mitigations . . . 92
16.6 The layered view, and the one question that matters . . . 93
Source anchors . . . 94
Experiments . . . 94

17 Storage and Encryption 95
17.1 Two layers on `/data` . . . 95
17.2 The FBE key hierarchy . . . 95
17.3 Storing a key so it can be destroyed . . . 96
17.4 From PIN to unlocked storage . . . 96
17.5 BFU versus AFU: the state that actually matters . . . 97
17.6 Secure Lock Device: hardening re-auth, but not the key . . . 97
17.7 Shared storage: a different isolation model . . . 98
Source anchors . . . 98
Experiments . . . 98

18 Keys, Credentials and Trusted Execution 100
18.1 From a PIN to the keys that protect everything . . . 100
18.2 Throttling in the TEE: twenty guesses in thirteen years . . . 101
18.3 Android 17's software rate limiter . . . 101
18.4 Keystore super-keys: the keys that vanish when you lock . . . 101
18.5 Enforced twice: Keystore, then KeyMint . . . 102
18.6 Post-quantum keys . . . 102
18.7 Attestation: proving what a device is . . . 103
18.8 Trusty: where the trusted code lives . . . 103
18.9 The newest boundary: protected VMs and AiSeal . . . 103
Source anchors . . . 104
Experiments . . . 104

19 Malware Analysis and Reverse Engineering 106
19.1 The platform as malware sees it . . . 106
19.2 The banking-trojan playbook, and Enhanced Confirmation Mode . . . 107
19.3 Developer verification . . . 108
19.4 Overlays, tapjacking, and the trusted-UI line . . . 108
19.5 Accessibility abuse — and a tested finding . . . 108
19.6 SMS codes, cellular, and the target-SDK gap meeting malware . . . 109
19.7 The analyst's boundaries . . . 109
19.8 Tools, and a defender's checklist . . . 110
Source anchors . . . 111

Experiments . . . 111

VI Porting 112

20 Treble, VINTF and the Vendor Interface 113
20.1 Treble in one sentence, and six layers . . . 113
20.2 Framework Compatibility Matrices . . . 114
20.3 The kernel contract became certification . . . 115
20.4 The device side: manifests and fragments . . . 115
20.5 Where compatibility is actually checked — and where it only warns . . . 116
20.6 Stable AIDL: freezing, and why released builds behave as "last frozen" . . . 116
20.7 The porter's view . . . 117
Source anchors . . . 117
Experiments . . . 117

21 Writing HALs 119
21.1 The interface . . . 119
21.2 Versioning and freezing . . . 119
21.3 The reference implementation is Rust . . . 120
21.4 Registration needs the VINTF declaration . . . 120
21.5 The four pieces of SELinux a HAL needs . . . 121
21.6 A defect found by reading to the end: the version mismatch . . . 121
21.7 A production variant, and the checklist . . . 122
Source anchors . . . 123
Experiments . . . 123

22 Anatomy of a Device: Cuttlefish 124
22.1 Products and lunch, organised by partition . . . 124
22.2 Board configuration: partitions, boot, and A/B . . . 124
22.3 The test-key problem . . . 125
22.4 The device-side configuration files . . . 125
22.5 The device SELinux policy . . . 126
22.6 The crucial caveat: Cuttlefish's "TEE" is a host process . . . 126
22.7 The porting checklist . . . 127
Source anchors . . . 128
Experiments . . . 128

23 Bring-up on Real Hardware 129
23.1 Choosing a target . . . 129
23.2 Two kinds of bring-up . . . 130
23.3 Read the target before you touch it . . . 130
23.4 The unlock gate . . . 131
23.5 The kernel: a certified GKI plus vendor modules . . . 131
23.6 First boot, the serial console, and debugging init . . . 131
23.7 To the launcher — and the security must-dos . . . 132

23.8 What was done, and what remains . . . 132
Source anchors . . . 133
Experiments . . . 133

24 Updates and Recovery **134**
24.1 Who may sign an update . . . 134
24.2 Anti-rollback: three layers and a final guard . . . 134
24.3 Virtual A/B: update now, decide later . . . 135
24.4 “Did the update work?” — three independent commit points . . . 135
24.5 When there is no update: RescueParty . . . 136
24.6 Mainline modules, and the shared safety net . . . 137
24.7 The update trust chain, assembled . . . 137
Source anchors . . . 137
Experiments . . . 138

Index **139**

PART V

Security

CHAPTER 14

The Android Security Model

Part V is about security as a model rather than a collection of features, and this chapter is its foundation. It makes one claim and spends the chapter defending it: **Android's security is Linux discretionary access control, plus SELinux mandatory access control, plus framework permission checks — and every permission is enforced in exactly one of those three places.** The single most useful habit this book can give a security engineer is to ask, of any permission or restriction, *which layer enforces it* — because the answer determines how it behaves, how it fails, and whether it can be bypassed. A permission enforced by a Linux group behaves nothing like one enforced by a Java method in `system_server`, and confusing the two is the root of most mistaken reasoning about Android security. With the identities (Chapter 9), Binder identity (Chapter 8), and signatures (Chapter 13) already in hand, this chapter assembles them into the model, and Part V's remaining chapters read specific subsystems in its light.

14.1 The threat model, in one paragraph

Android assumes three things, and designs to them. **Apps are mutually untrusted**: any app may be malicious, so each is isolated from the others by default. **The user must be protected from apps**: an app gets only what it is granted, and the sensitive grants require the user's explicit consent. And **the platform must be protected from all apps**: no app, however it is signed or whatever it holds, may subvert the system itself. These three give the model its shape — an app is a sandbox, a permission is a hole deliberately opened in that sandbox, and the system sits behind boundaries that no permission can breach. The rest is mechanism.

14.2 Identities: the UID space

The oldest and sturdiest boundary is the Unix UID, and Android lays a rich structure over it. The `AID_` constants (159 of them) partition the UID space: root at 0; fixed system identities from 1000 (`AID_SYSTEM`) up; a band of **group** IDs (3003 `inet`, 3005 `net_admin`, 3009 `readproc`, …) that are capabilities expressed as supplementary groups; partition-reserved ranges; **apps** from 10000; SDK-sandbox and isolated-process ranges above that. [SRC: `android_filesystem_config.h`] The UID and GID namespaces deliberately reuse numbers — 20000 is both an SDK-sandbox UID and a cache GID — because they are read in different contexts.

Multi-user multiplies the scheme with one formula: **`uid = userId` × `100000 + appId`**. [SRC: `UserHandle.java`] A single installed APK therefore runs as *different Linux identities* for different users — the test app on the Pixel 11 is UID **10346** for the primary user (appId 10346, user 0), and would be 1010346 for a second user: two separate kernel identities, isolated from each other

by the same UID mechanism that isolates unrelated apps. [OBSERVED Pixel 11, `run-as … id`] SELinux encodes the same split in MLS categories (Chapter 15), so the per-user, per-app isolation is enforced twice.

14.3 Permissions: what the platform actually declares

The user-facing "permission model" — the runtime prompts for location, camera, contacts — is a small corner of a much larger system. The platform declares **1,160 permissions**, and sorting them by protection level is clarifying:

Base level	Count	Who gets it
`signature`	864	only apps signed with the declaring (usually platform) key
`internal`	133	never by signature alone — only via a flag (role, installer, …)
`normal`	108	automatically at install
`dangerous`	51	the user, at a runtime prompt
other	4	`role`, `system`, `signatureOrSystem`, `module`

[SRC: `frameworks/base/core/res/AndroidManifest.xml`] The headline number is the one to remember: **more than four-fifths of the platform's permissions are not for third-party apps at all** — they are `signature` or `internal`, reachable only by platform-signed code, privileged apps, role holders, or installers. The entire visible permission model — the prompts a user ever sees — is the **51 `dangerous`** permissions. When someone says "Android's permission model," they usually mean those 51; this chapter is about the other 1,109 as well, because that is where most of the system's own security decisions live. (`INTERNET`, famously, is `normal` — granted silently at install — which brings us to how a permission actually becomes enforcement.)

14.4 From a permission to enforcement: three layers, one example each

A permission is just a string until something checks it, and Android checks different permissions in different layers. There are three patterns, and `INTERNET` happens to illustrate the most interesting one.

Layer 1 — permission becomes a Linux group (DAC). Some permissions map, via `platform.xml`, to supplementary GIDs: `INTERNET` → `inet` (3003), `READ_LOGS` → `log`, `NET_ADMIN` → `net_admin`. [SRC: `platform.xml`] `system_server` turns an app's granted permissions into GIDs, and the zygote `setgroups` them at fork (Chapters 9, 12). After that the **kernel** enforces access through

ordinary ownership and modes. On the Pixel, the test app — which declares `INTERNET` — carries GID **3003 (inet)** in its group set, alongside the per-app categories in its SELinux context (`u:r:runas_app:s0:c90,c257,c512,c768` under `run-as`). [OBSERVED Pixel 11]

Layer 2 — permission becomes an eBPF map entry (kernel, via a Mainline module). This is the `INTERNET` subtlety: the real gate moved. You might expect the `inet` group to *be* the enforcement of internet access. It is not, any more. Upstream GKI carries no "paranoid network" patch — there is no `AID_INET` check in the kernel's networking code. Instead, `INTERNET` is enforced in two places: a **libc socket hook** (`netdClientSocket`) returns `EPERM` for `AF_INET` sockets when networking is not allowed for the process — but this is a *library* hook, advisory, bypassable by code that issues the raw `socket` syscall — and, authoritatively, an **eBPF `cgroup/sock` program** in the Connectivity Mainline module checks the caller's appId against a `uid_permission_map` and blocks socket creation for a UID lacking the internet bit. [SRC: `NetdClient.cpp`, `packages/modules/Connectivity/bpf/progs/netd.c`] So the `inet` GID is now largely vestigial as *enforcement*; the true gate is a permission map in a Mainline module, applied by eBPF in the kernel. This is a perfect instance of the chapter's thesis and of Chapter 1's "the build is not the whole story": to know how `INTERNET` is enforced you must look past the GID the documentation implies, to the module and the eBPF program that actually decide.

Layer 3 — framework-only checks. The 864 signature and 133 internal permissions that guard system services are enforced by neither DAC nor a dedicated LSM but by the service itself, calling `checkCallingPermission` on its Binder caller (Chapter 8). SELinux decides who may *find* and *call* the service; the permission check inside it decides what the call may *do*. This is the "service is the last line of defence" pattern from Chapter 8, and it accounts for the great majority of the platform's permissions.

Three layers, then — a GID the kernel enforces, an eBPF map a module enforces, a Java check a service enforces — and the only way to reason correctly about any given permission is to know which one it uses. The first two are the kernel half of the chapter's thesis; SELinux frames all three, deciding which processes, files and services a caller can reach before any permission is checked.

14.5 Install-time gates

Some policy is enforced not per-call but once, at install. The most consequential is the **minimum target-SDK floor**: an app targeting below API 24 cannot be installed. The flag that sets this has no value in `cp2a` itself but inherits — through the release chain of Chapter 2 — from `bp1a`, where it is **enabled**; the resolver confirms it. [SRC×2: `min_target_sdk_24` resolves ENABLED] The bypass (`--bypass-low-target-sdk-block`) is stripped from any caller that is not system, root, shell, or a debuggable build, so it is not a third-party escape. [SRC: `PackageInstallerService.java`] This single gate is why, throughout this book, "Android allows X for apps targeting below N" describes a shrinking population of *preinstalled legacy* code and never anything a user can newly install — the W+X segments of Chapter 6, the v1-only signatures of Chapter 13, the private-library exemptions, all of them. The floor rises over time, dragging the whole ecosystem's minimum behaviour up with it, which is the entire point of the target-SDK ratchet.

14.6 `fs_config` and the privileged-binary inventory

The last piece of the static model is the image itself. `fs_config` gives every file in the system images its owner, mode and **file capabilities**. [SRC: `fs_config.cpp`] The security-relevant reading is how *few* privileged binaries ship: on a `user` build, the set-uid-root helpers like `su` and `procmem` are explicitly "NOT included," and only **three** executables carry file capabilities — `inputflinger` (`CAP_BLOCK_SUSPEND`), `run-as` (`CAP_SETUID|CAP_SETGID`), and `simpleperf_app_runner` (`CAP_SETUID|CAP_SETGID`). [SRC] There are, in effect, no setuid binaries on a shipping user build; privilege is held by a tiny, auditable set of capability- bearing helpers.

`run-as` deserves a note because this chapter has leaned on it: it is the mechanism (`CAP_SETUID|CAP_SETGID`) that lets the shell become a **debuggable** app's UID, which is exactly how the device anchors in this and the previous chapter read an app's own data directory and identity. It is a deliberate, narrow boundary — it refuses non-debuggable apps — and a classic target for scrutiny when porting, because a flaw in it would hand the shell any debuggable app's identity.

14.7 Defence in depth, assembled

Collecting the whole book so far, a single app process is confined by a stack of independent mechanisms, each set in one place and enforced in another:

Boundary	Set at	Enforced by
UID and GIDs	zygote `setresuid` / `setgroups`	kernel DAC
capabilities (none for apps)	zygote	kernel
seccomp filter	zygote, before `setuid`	kernel
SELinux domain + MLS	zygote `selinux_android_setcontext`	kernel LSM
internet / local-network	Connectivity module's permission map	eBPF cgroup program
service access	servicemanager + `binder: call`	userspace SELinux + kernel LSM
API permissions	each system service	`system_server` Java
install policy	PackageManager	`system_server`

No single mechanism is the sandbox; the sandbox is their conjunction. An attacker does not defeat "the Android sandbox," they must defeat a specific layer for a specific goal, and the layers were chosen so that the ones most exposed to untrusted input (seccomp, SELinux, the eBPF network gate) are also the ones hardest to subvert from userspace. This table is the mental model Part V builds on.

14.8 What the rest of Part V examines

The model in this chapter is stable — most of it predates Android 17 — but the platform keeps extending it, and Android 17's additions are where the book's device testing paid off. Several are features that layer *onto* this model and that later chapters take one at a time: **Advanced Protection Mode**, which flips a coordinated set of these settings to their strict positions; the **TLS defaults** (Certificate Transparency, Encrypted Client Hello) that ride the target-SDK ratchet; the **local-network permission** that splits a new hole out of `INTERNET`; the **assistant and Computer Control** boundaries around on-screen content; and the **location button**, a new trusted-UI path for one-time location. Each extends the three-layer model rather than replacing it, and each carries the Android 17 lesson this book keeps returning to: **what the source ships is the baseline, not the verdict.** A protection can be in the tree but gated off by a flag, and — as the Pixel 11 testing in these chapters shows — a flag the source leaves off can be switched *on* by Google's server configuration after the fact. The model is in the source; its live state is on the device; and a security engineer needs to read both.

Source anchors

- Identities: `system/core/libcutils/include/private/android_filesystem_config.h` (the `AID_` map); `frameworks/base/core/java/android/os/UserHandle.java` (the UID formula).
- Permissions: `frameworks/base/core/res/AndroidManifest.xml` (the 1,160 `<permission>` declarations); `frameworks/base/data/etc/platform.xml` (permission→GID).
- INTERNET enforcement: `system/netd/client/NetdClient.cpp` (the libc hook); `packages/modules/Connectivity/bpf/progs/netd.c` and `.../connectivity/PermissionMonitor.java` (the eBPF gate).
- Install gates: `frameworks/base/services/core/java/com/android/server/pm/{PackageManagerService,InstallPackageHelper, PackageInstallerService}.java` (`MIN_INSTALLABLE_TARGET_SDK`, the bypass).
- fs_config: `system/core/libcutils/fs_config.cpp` (modes, file capabilities, the user-build inventory).

Experiments

These run on any device over `adb`; `run-as` needs a debuggable app.

1. **Read an app's identity (run on device).** For a debuggable app, `adb shell run-as <pkg> id`: the UID is `userId × 100000 + appId`, and the group set includes the GIDs its permissions map to — `3003 (inet)` for `INTERNET`. (Observed UID 10346 with `inet` on the Pixel 11.)
2. **Distinguish the two INTERNET gates.** From an app without `INTERNET`, open an `AF_INET` socket through libc (`EPERM` from the NetdClient hook) and then via the raw `socket`

syscall (`EPERM` from eBPF): the §14.4 point that the real enforcement is the eBPF map, not the library hook.

3. **Hit the install floor.** `adb shell pm install` a target-SDK-23 APK → `INSTALL_FAILED_DEPRECATED_SDK_VERSION`; retry with `--bypass-low-target-sdk-block` from the shell (allowed) versus from an ordinary installer (stripped).
4. **Inventory privilege.** On an unpacked system image, `find <unpacked-dir> -perm -4000` (no setuid binaries on a user build) and `getcap -r` (only `inputflinger`, `run-as`, `simpleperf_app_runner`) — the §14.6 claim, checkable statically.
5. **Map permissions to groups.** `adb shell id` (shell's own groups) and, via `run-as`, an app's, cross-referenced against `platform.xml` — the §14.4 Layer-1 mechanism.

Index

A

A/B updates, 5, 24, 26, 125, 127, 130, 134, 135, 136, 137, 138
  Virtual A/B, 5, 26, 125, 130, 134, 135, 137, 138
accessibility services, 106, 107, 108, 110, 111
aconfig, 3, 10, 46
activities, 42, 43, 45, 48, 51, 55, 63, 68, 69, 70, 71, 72
`adb`, 7, 13, 14, 19, 22, 27, 31, 32, 38, 43, 50, 55, 61, 66, 72, 77, 83, 89, 94, 99, 105, 108, 117, 123, 128, 131, 133, 138
Advanced Protection Mode, 71, 83, 108
AIDL, 5, 11, 48, 102, 103, 113, 114, 116, 117, 119, 120, 122
  stable AIDL, 113, 114, 116, 117, 119, 120, 122
AiSeal, 3, 6, 11, 104, 124
Android Common Kernel, 6, 8, 10, 11, 15, 18, 20
Android Runtime (ART), 4, 37, 51, 57, 58, 59, 60, 66, 110
Android Verified Boot (AVB), 3, 5, 6, 16, 22, 23, 24, 25, 26, 27, 29, 76, 102, 103, 125, 127, 129, 130, 131, 132, 133, 134, 135, 137
  avbtool, 12, 13, 22, 23, 25, 27
  boot states (green, yellow, orange), 24, 27, 29, 33, 125, 127, 130, 131, 135
  libavb, 22, 23, 24, 26, 115
  rollback protection, 23, 26, 29, 134, 135, 136, 137
  vbmeta, 23, 25, 26, 27, 29, 76, 125, 127
Android Virtualization Framework (AVF), 3
Android.bp, 5, 11, 14, 121, 123
Android.mk, 11, 14
AOSP, 2, 6, 13, 25, 27, 85, 107, 108, 109, 115, 125, 127, 129
APEX, 3, 4, 5, 30, 34, 35, 36, 38, 40, 42, 58, 59, 73, 76, 77, 104, 114, 122, 137
APK, 5, 6, 11, 23, 36, 58, 73, 74, 77, 79, 84, 86, 102, 110, 125
APK signing, 5, 6, 11, 13, 23, 57, 58, 73, 74, 75, 77, 86, 96, 102, 110, 115, 125, 128, 132, 134
  v1/v2/v3/v4 schemes, 18, 22, 42, 43, 73, 75, 77, 95, 102, 119, 120, 121, 125, 127, 135
application sandbox, 3, 18, 29, 37, 51, 53, 60, 75, 77, 79, 82, 85, 86, 87, 90, 91, 106, 109, 110
ashmem, 18, 87
ASLR, 54, 90, 93

B

background activity launch (BAL), 69, 70, 72
banking trojans, 106, 107, 108
BFU and AFU, 95, 97, 99, 102, 107, 110
Binder, 3, 4, 6, 7, 15, 16, 17, 18, 20, 31, 39, 45, 46, 47, 48, 49, 50, 55, 66, 69, 72, 79, 81, 82, 87, 89, 113, 116, 119, 120, 127, 128
  oneway calls, 47
  Parcel, 48, 49
biometrics, 97, 102
Bionic, 2, 3, 6, 10, 34, 53, 91
BoardConfig.mk, 11, 124, 125
bootconfig, 22, 31, 33
`boot.img`, 5, 22, 26, 27, 51, 54, 125
bootloader, 2, 22, 23, 24, 25, 27, 28, 29, 30, 31, 46, 100, 103, 127, 129, 131, 132, 133, 135, 136, 137, 138
bootloader unlocking, 25, 27, 29, 33, 125, 130, 133, 138
Branch Target Identification (BTI), 93
bring-up, 117, 120, 122, 129, 130, 131, 133
build ID, 2, 9
build variants, 10, 11, 14, 38, 40, 82, 84, 127
  userdebug, 10, 11, 14, 38, 40

C

cgroups, 18, 42, 43, 53, 65, 69, 82
checkpointing (userdata), 125, 136, 137, 138
`checkvintf`, 18, 116, 117, 133
Compatibility Test Suite (CTS), 88
content providers, 69
Control-Flow Integrity (CFI), 90, 94
`crash_dump`, 40, 42
`crash_dump`, 43

Cuttlefish, 3, 5, 7, 11, 12, 14, 16, 19, 22, 24, 25, 27, 32, 43, 46, 50, 55, 66, 104, 108, 115, 117, 122, 123, 124, 125, 126, 127, 128, 129, 132, 135

D

device tree, 22, 31, 131
DEX, 12, 57, 58, 60, 61, 106, 110
`dex2oat`, 58
`dexdump`, 13, 57, 61
DICE, 103, 127
discretionary access control (DAC), 30, 31, 32, 39, 40, 52, 75, 80, 82, 85, 87, 98, 120
`dlopen`, 38, 60, 106
`dlopen`, 35, 37
dm-verity, 18, 29, 76, 115, 136, 137
`dumpsys`, 43, 69, 72, 132
dynamic linker, 5, 12, 34, 35, 36, 37, 38, 110, 113
dynamic partitions, 126

E

eBPF, 18, 42, 81, 82, 84, 92, 93, 94, 115
emulator, 13, 87, 104, 119, 122, 123
Enhanced Confirmation Mode (ECM), 70, 107, 108, 110
`execve`, 37, 38, 57, 106
`execve`, 60

F

fastboot, 25, 133
file-based encryption (FBE), 18, 95, 96, 125, 127, 132
  CE and DE storage, 96, 97, 99, 102, 107, 110
  metadata encryption, 18, 125, 127, 132
first-stage init, 11, 16, 28, 29, 33, 131, 133
`fork`, 4, 12, 34, 35, 51, 52, 53, 54, 80
`fork`, 55
Frida, 110
`fstab`, 132
`fstab`, 127

G

Gatekeeper, 100, 101, 103, 104, 126
Generic Kernel Image (GKI), 5, 8, 10, 15, 16, 17, 18, 19, 20, 22, 46, 54, 81, 88, 90, 113, 114, 115, 116, 117, 118, 127, 129, 130, 131, 132, 133, 135
Generic System Image (GSI), 124, 125, 127, 130, 133
GID, 31, 52, 53, 63, 79, 80, 82, 83, 87
git history, 2, 3, 8, 10, 13, 87, 115, 121, 132
GWP-ASan, 34, 52, 54, 93

H

Hardware Abstraction Layer (HAL), 3, 5, 7, 26, 34, 36, 45, 48, 85, 100, 103, 109, 113, 114, 115, 116, 117, 119, 120, 121, 122, 123, 124, 126, 127, 128, 129, 130, 132, 133, 136
HardwareAuthToken, 102
hidepid, 29, 33, 107
HIDL, 48, 114, 117, 130
hypervisor, 7, 18, 103, 124

I

init, 5, 16, 23, 28, 29, 30, 31, 32, 35, 39, 40, 46, 51, 58, 65, 67, 85, 92, 93, 117, 120, 121, 122, 125, 129, 131, 133, 136
  property service, 27, 28, 30, 32, 39, 46, 56, 61, 99, 105, 117, 128, 138
  .rc language, 30, 40, 51, 52, 58, 65, 92, 113, 120, 122
  `vendor_init`, 28, 30, 32, 113, 125
`init_boot`, 125
InputDispatcher, 71, 108
`installd`, 73, 75
Intent, 70
`INTERNET permission`, 52, 63, 79, 80, 83

J

JADX, 110
JIT compilation, 37, 59, 61, 110, 115
JNI, 12, 60

K

kcmdlinemodprobe, 3, 16, 46
kernel command line, 16, 23, 31, 33, 46, 91
kernel configuration, 12, 13, 18, 46, 54, 90, 103, 115, 131
Kernel Module Interface (KMI), 6, 15, 16, 17, 19, 131
key attestation, 22, 25, 27, 29, 45, 103, 110, 125, 126, 128, 129, 131, 132
  RootOfTrust, 25, 27, 100, 103, 105, 110, 125, 126, 127, 128
KeyMint, 6, 24, 25, 96, 102, 103, 105, 114, 126
Keystore, 3, 6, 39, 40, 48, 58, 63, 96, 97, 100, 101, 102, 103
Kleaf, 11
KVM, 12, 128

L

linker namespaces, 12, 30, 35, 36, 38, 53, 79, 91, 93, 98, 113
Linux capabilities, 4, 40, 51, 53, 54, 63, 64, 65, 66, 74, 75, 79, 82, 88, 93, 130
Linux Security Module (LSM), 18, 31, 47, 54, 64, 81, 82, 91
`lmkd`, 39, 41, 42, 68
`lmkd`, 43
location button, 106, 108, 111
lock screen, 10, 13, 96, 97
    rate limiting, 10, 13, 101
`logd`, 38, 39, 40, 43, 50, 55, 67, 72, 123, 132
lunch, 11, 12, 14

M

Mainline, 4, 5, 7, 34, 35, 73, 76, 81, 85, 98, 103, 114, 134, 136, 137
malware, 34, 37, 69, 71, 104, 106, 108, 109
mandatory access control (MAC), 30, 31, 32, 39, 76, 85, 92, 102, 120
MediaProvider, 98
memfd, 18, 59, 61, 87
memory safety, 4, 46, 93, 127
Memory Tagging Extension (MTE), 34, 52, 54, 90, 93
ML-DSA, 6, 23, 26, 27, 74, 77, 102, 104

N

`netd`, 65

O

`odrefresh`, 59
`odrefresh`, 58
`odsign`, 57, 58, 96
`odsign`, 61
OOM killer, 41, 68, 72
over-the-air updates (OTA), 5, 34, 76, 86, 116, 122, 134, 135, 136, 137, 138

P

PackageManagerService, 73, 75
PendingIntent, 70
permissions, 4, 5, 18, 31, 40, 48, 52, 59, 63, 64, 65, 66, 70, 71, 73, 75, 76, 79, 80, 82, 83, 95, 104, 107, 108, 109
Pixel 11, 4, 6, 7, 8, 15, 19, 20, 25, 29, 30, 31, 32, 34, 35, 38, 40, 42, 43, 46, 49, 50, 51, 52, 55, 59, 61, 63, 66, 68, 72, 76, 77, 79, 81, 83, 86, 87, 89, 91, 94, 95, 99, 103, 105, 106, 108, 114, 115, 117, 125, 126, 129, 130, 132, 133, 135, 138
pKVM, 18, 103, 105, 127, 129
Pointer Authentication (PAC), 93
post-quantum cryptography, 6, 23, 27, 73, 74, 102
`/proc`, 7, 20, 29, 33, 41, 55, 61, 63, 66, 72, 91, 92, 94, 109, 128
ProcessList, 51
Project Treble, 3, 4, 7, 12, 15, 16, 18, 28, 30, 31, 34, 36, 41, 48, 85, 113, 114, 115, 117, 122, 124, 128, 130, 131, 133
protected VM, 59, 104
PSI (pressure stall information), 18, 41, 43
ptrace, 40, 52, 87, 109, 110

R

recovery, 125, 134, 137
release configs, 3, 10, 13, 46, 54, 74, 81, 104, 120, 124
Remote Key Provisioning (RKP), 103, 104, 126, 132
repo (tool), 9, 15
RescueParty, 65, 136, 137
`run-as`, 76, 77, 80, 81, 82, 83, 110
Rust, 3, 6, 12, 16, 18, 22, 45, 46, 47, 48, 50, 87, 90, 119, 120, 121, 122, 127, 128
    in the kernel, 3, 6, 16, 20, 46, 50, 90, 127, 128

S

SafeSetID, 18, 54, 64, 65, 66, 91
Scudo, 34
seccomp, 4, 34, 51, 53, 55, 64, 66, 82, 90, 91, 93, 94, 106, 110, 115
`secdiscardable`, 96, 99
SELinux, 3, 4, 5, 7, 11, 12, 16, 19, 28, 30, 31, 32, 37, 39, 40, 45, 47, 48, 49, 51, 52, 54, 58, 63, 64, 66, 73, 75, 76, 79, 80, 81, 82, 85, 86, 87, 88, 90, 91, 92, 93, 94, 98, 106, 110, 113, 115, 116, 119, 120, 121, 122, 124, 125, 126, 127, 130, 131, 132, 133, 135
    AVC denials, 40, 50, 94, 132
    domains, 4, 12, 18, 28, 29, 30, 31, 32, 36, 37, 40, 45, 48, 50, 51, 52, 54, 58, 59, 63, 66, 73, 75, 82, 85, 86, 87, 89, 92, 110, 120, 121, 123, 125, 126, 127, 128
    enforcing and permissive, 11, 19, 29, 31, 39, 41, 86, 89, 90, 91, 92, 93, 110, 121, 126, 127, 130
    `file_contexts`, 29, 121, 122
    MLS categories, 40, 65, 76, 77, 80, 82, 85, 86,

87, 89, 98, 107
neverallow, 12, 31, 37, 47, 52, 58, 63, 66, 85, 88, 89, 92, 93, 127, 132
neverallow, 86, 121, 126
policy sources (sepolicy), 113
seapp_contexts, 54, 75, 110
seapp_contexts, 52, 75, 86
service_contexts, 48, 50, 121, 122
untrusted_app, 48, 54, 86, 87, 89
servicemanager, 39, 45, 47, 48, 82, 120, 121
setuid, 54, 55, 64, 82, 84, 91
sideloading, 108, 134
SMS one-time codes, 97, 107, 109, 110, 111, 132
Soong, 3, 11
strace, 55
StrongBox, 7, 127
superproject, 9
SurfaceFlinger, 48, 71
synthetic password, 96, 100
system properties, 24, 27, 30, 31, 32, 93, 103, 104, 105, 110, 128, 130
SYSTEM_ALERT_WINDOW, 69, 71, 72, 106, 107, 108, 111
system_server, 7, 45, 48, 49, 51, 52, 54, 58, 63, 64, 65, 66, 68, 69, 79, 80, 82, 86, 88, 121, 128, 137, 138
system_server, 63, 120

T

targetSdkVersion, 10, 36, 37, 38, 61
tombstones, 40, 42, 43, 91, 94
Trusted Execution Environment (TEE), 7, 25, 29, 96, 100, 101, 102, 103, 124, 126, 127, 128, 129, 132
Trusty, 17, 100, 101, 103, 126

U

U-Boot, 22, 24, 27
ueventd, 28, 31, 39
UID, 4, 18, 29, 39, 40, 41, 47, 51, 52, 53, 54, 55, 63, 64, 65, 73, 75, 76, 79, 81, 82, 83, 86, 89, 91, 120
UnlockedDeviceRequired, 97, 102, 105, 126
update_engine, 3, 26, 134, 135, 136
update_verifier, 136, 137, 138

V

vendor hooks, 17, 20
vendor modules, 16, 17, 18, 19, 114, 117, 127, 129, 131, 132
Vendor Test Suite (VTS), 15, 18, 87, 114, 115, 116, 117, 121, 122, 123, 132, 133
vendor_boot, 22, 125
vendor_dlkm, 5, 124, 127, 131
vendor_dlkm, 16
VINTF, 3, 5, 12, 18, 48, 113, 114, 115, 116, 117, 119, 120, 121, 122, 123, 126, 127, 129, 132
compatibility matrices, 5, 18, 102, 114, 115, 117, 119, 122, 127, 130, 133
device manifest, 115, 116, 117
VNDK, 36
vold, 3, 35, 39, 40, 136
vold, 96

W

Watchdog, 65, 66, 137
Weaver, 100, 101
windows, 17, 51, 55, 57, 58, 68, 69, 71, 72, 102, 135, 138

Z

zygote, 4, 28, 34, 42, 47, 51, 52, 53, 54, 55, 57, 58, 59, 60, 63, 64, 65, 66, 80, 82, 86, 88, 91, 93, 109
app zygote, 52, 54, 64, 65, 66
USAP, 53



ABOUT THE AUTHOR

Regan Russell

Regan Russell has been writing software professionally since 1986. His 35 years span defence, banking, telecommunications, pay television, government and embedded systems: mainframes and device drivers, then mobile, security and Web3.

Through his company, Pymble Software, he publishes Android apps on Google Play, including Pocket CPM, Pocket ARM, Web3 Inspector, iRelayChat, Shieldkeep, Orbital Fury, Noir Descent, Rescue Rapunzel, Plate Panic and Ticker Tape Tycoon.

He is also the author of *Thirty-five Years in Software Development*.

Take Android apart, from the bootloader up.

Most books describe Android. This one reads it. *Android Internals: Security and Porting* takes a single, pinned release of Android 17 and follows it from the first instruction the bootloader runs to the app on the screen, through every layer and every security boundary on the way.

Every claim is anchored in the source tree, its git history, or an experiment you can repeat on Cuttlefish or a retail Pixel. Nothing rests on folklore. You'll see how:

- verified boot, the kernel and **init** build a trusted system from nothing;
- **Binder**, the **zygote** and ART turn one process into thousands of apps;
- **SELinux**, seccomp and the app sandbox stack defences in depth;
- keys, the TEE and file-based encryption protect data at rest;
- malware is taken apart, and what Android 17 does to stop it;
- Treble, VINTF and HALs let a vendor port Android to new hardware;
- A/B and Virtual A/B updates keep a device safe for years.

Written for security engineers, platform and porting engineers, reverse engineers and curious developers who want to know how Android *really* works.

AOSP · Build & Porting · Security · Boot & OTA · Reverse Engineering