

Android Platform Security

Mobile Security 2026

Florian Draschbacher
florian.draschbacher@tugraz.at

Some slides based on material by **Johannes Feichtner**

Outline

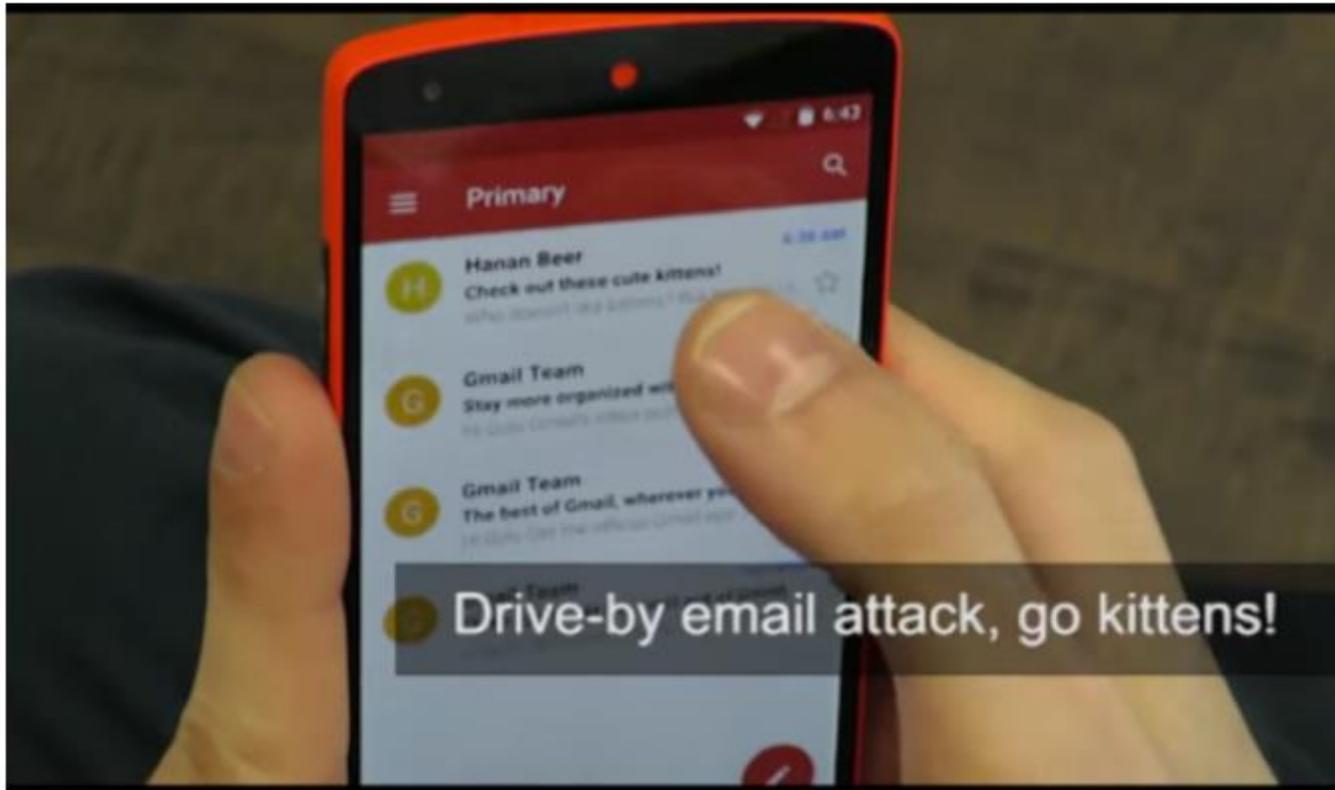
- Android Platform Fundamentals
- Low-level System Security
- Encryption System
- Android OS Security
- Key Management
- Rooting



275 million Android phones imperiled by new code-execution exploit

Unpatched "Stagefright" vulnerability gives attackers a road map to hijack phones.

DAN GOODIN - 3/18/2016, 9:26 PM



Source: <https://goo.gl/9fgYSc>



What?

Bugs in Android's libstagefright and libutils

How?

- Attacker embeds shellcode in harmless multimedia file
- Message is downloaded (e.g. via MMS)
- Exploit is executed

Result

- Attacker can execute any code on remote device

[RESEARCH](#)

28 February 2025

Cellebrite zero-day exploit used to target phone of Serbian student activist

Amnesty International's Security Lab, in collaboration with Amnesty's European Regional Office, has uncovered a new case of misuse of a Cellebrite product to break into the phone of a youth activist in Serbia. The attack closely matches the form of attack that we previously documented in a report, '[A Digital Prison](#)', published in December 2024. This new case provides further evidence that the authorities in Serbia have continued their campaign of surveillance of civil society in the aftermath of our report, despite widespread calls for reform, from both [inside Serbia](#) and [beyond](#), as well as an investigation into the misuse of its product, announced by Cellebrite.

Source: [amnesty.org](https://www.amnesty.org)

What?

Exploits do happen, harming real people

How?

- There are exploit kits
- Sold to law enforcement agencies
- But also misused against activists

Technique

- Exploit vulnerabilities in the USB stack to install malware despite locked screen

Android Security Bulletin—March 2026

Published March 2, 2026 | Updated March 9, 2026

This Android Security Bulletin contains details of security vulnerabilities that affect Android devices. Security patch levels of 2026-03-05 or later address all of these issues. To learn how to check a device's security patch level, see [Check and update your Android version](#).

Qualcomm components

These vulnerabilities affect Qualcomm components and are described in further detail in the appropriate Qualcomm security bulletin or security alert. The severity assessment of these issues is provided directly by Qualcomm.

CVE	References	Severity	Subcomponent
CVE-2026-21385	A-478214401 QC-CR#4387106 [2]	High	Display

★ **Note:** There are indications that CVE-2026-21385 may be under limited, targeted exploitation.

What?

Vulnerabilities that are actively exploited are constantly discovered

Here:

- Integer overflow in GPU driver
- The GPU is reachable from unprivileged processes

Result:

- Unprivileged apps can escalate to kernel privileges

MOBILE & WIRELESS

Project Zero: Samsung Mobile Chipsets Vulnerable to Baseband Code Execution Exploits

Critical security flaws expose Samsung's Exynos modems to "Internet-to-baseband remote code execution" attacks with no user interaction. Project Zero says an attacker only needs the victim's phone number.



By [Ryan Naraine](#)
March 16, 2023



What?

18 vulnerabilities in baseband code of Exynos SoC

4 did not require special network access

How?

- Attacker send maliciously crafted message
- Triggers heap overflow in baseband code

Result

- Attacker can intercept and manipulate cellular communication

Android Platform Fundamentals

Android

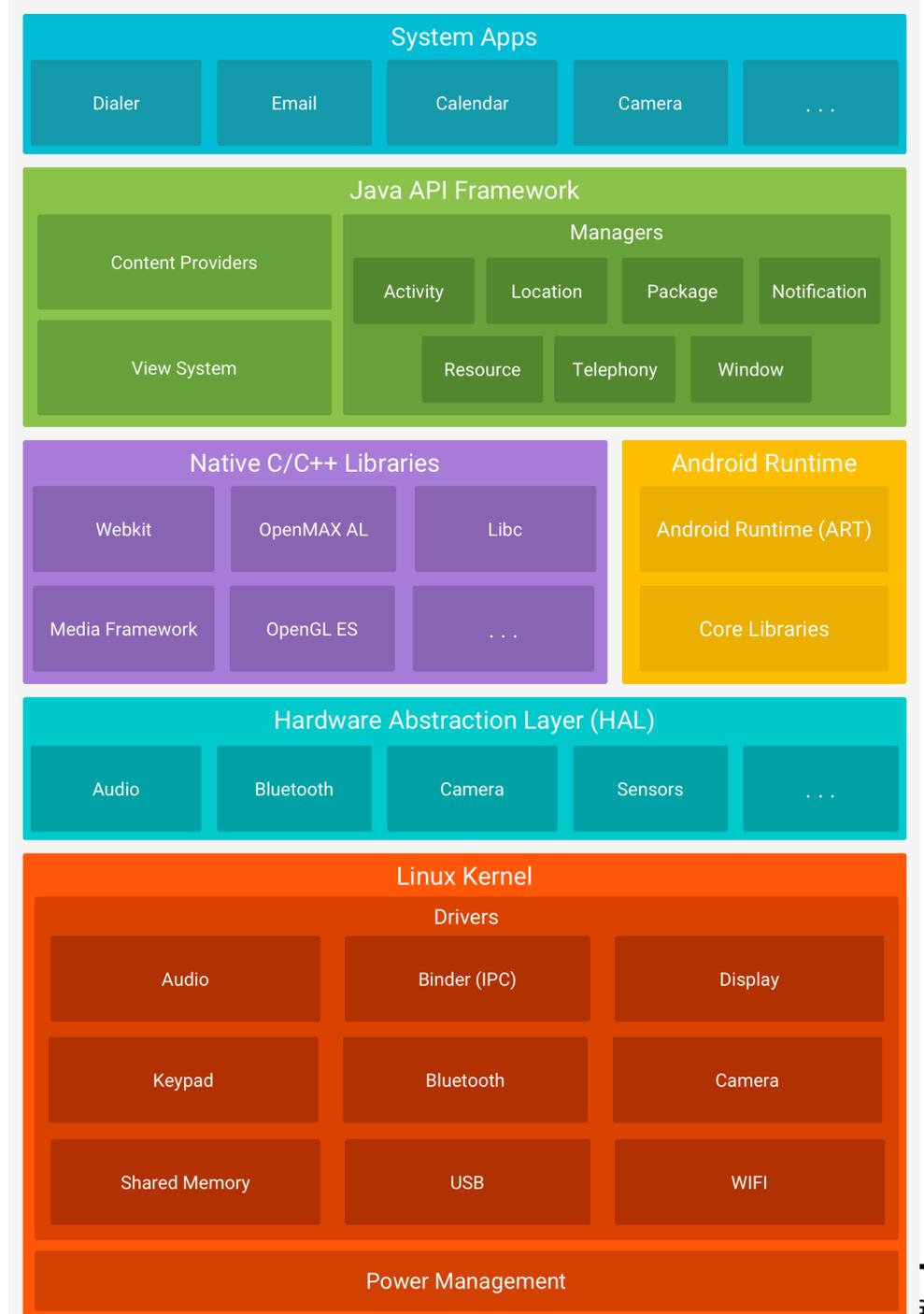
- Open-Source OS developed mainly by Google
 - Linux kernel: GNU GPLv2, Rest: Apache 2.0
 - Many implementation details can be studied from source code!
- Wide device support
 - CPU architectures, hardware features, ...
 - Used by various device manufacturers
 - Proprietary additions, modifications, forks
- Compatibility Test Suite ensures compatibility
 - Requirement for access to Google Mobile Services (Play Store, ...)

Android Device Architecture

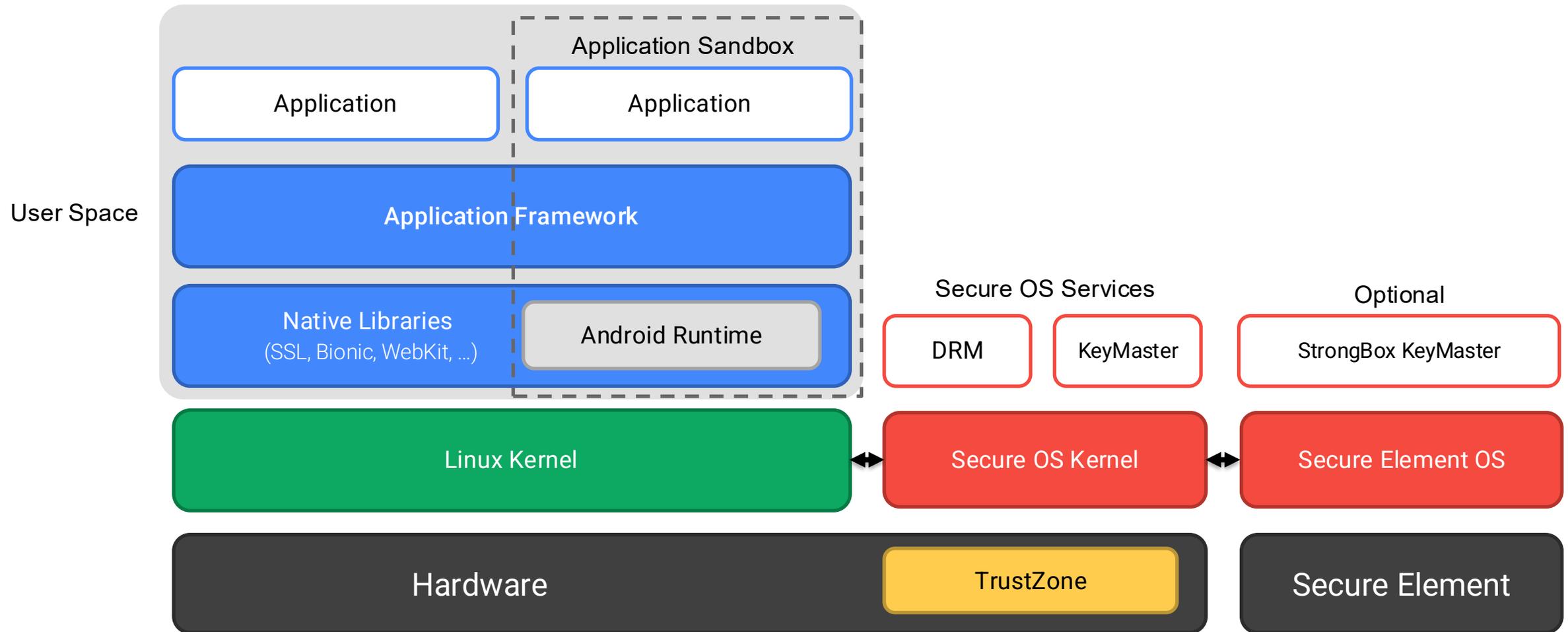
- **Most** Android devices feature a main CPU and some secure environment
 - Secure Key Storage
 - Handling biometric unlock (Fingerprint, ...)
- ARM TrustZone
 - Secure environment runs in a separate execution environment on main CPU
- Secure Element
 - Secure environment runs on a dedicated CPU
 - E.g. Google Titan M2 in Google devices starting from Pixel 6

Android System Architecture

- Linux kernel
 - Device drivers
 - POSIX interface
 - Binder IPC
 - Low Memory Killer
- Userspace
 - HAL (Hardware Abstraction Layer)
 - Android Runtime
 - System Services
 - Application Framework



Android Security Architecture

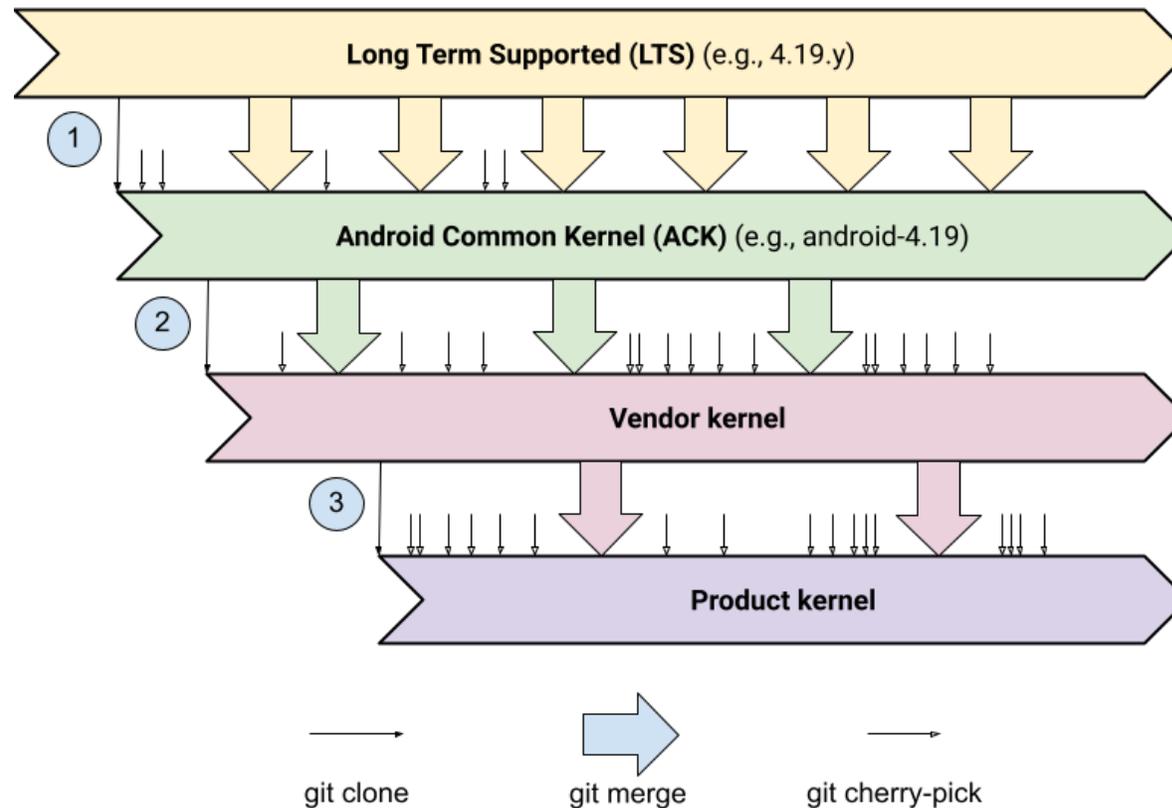


Android Fragmentation

- Android is shipped by many different device manufacturers
 - Different CPU architectures, HW peripherals, UI modifications, ...
- Releasing OS update for a device used to be time-consuming
 - Obtaining updated firmware from peripheral vendors
 - Porting modifications to new base
- Situation improved with Project Treble (Android 8.0 / 2017)
 - Low-level vendor implementation untouched in Android updates
- Further improvements with Project Mainline (Android 10.0 / 2019)
 - System components can be updated through Google Play

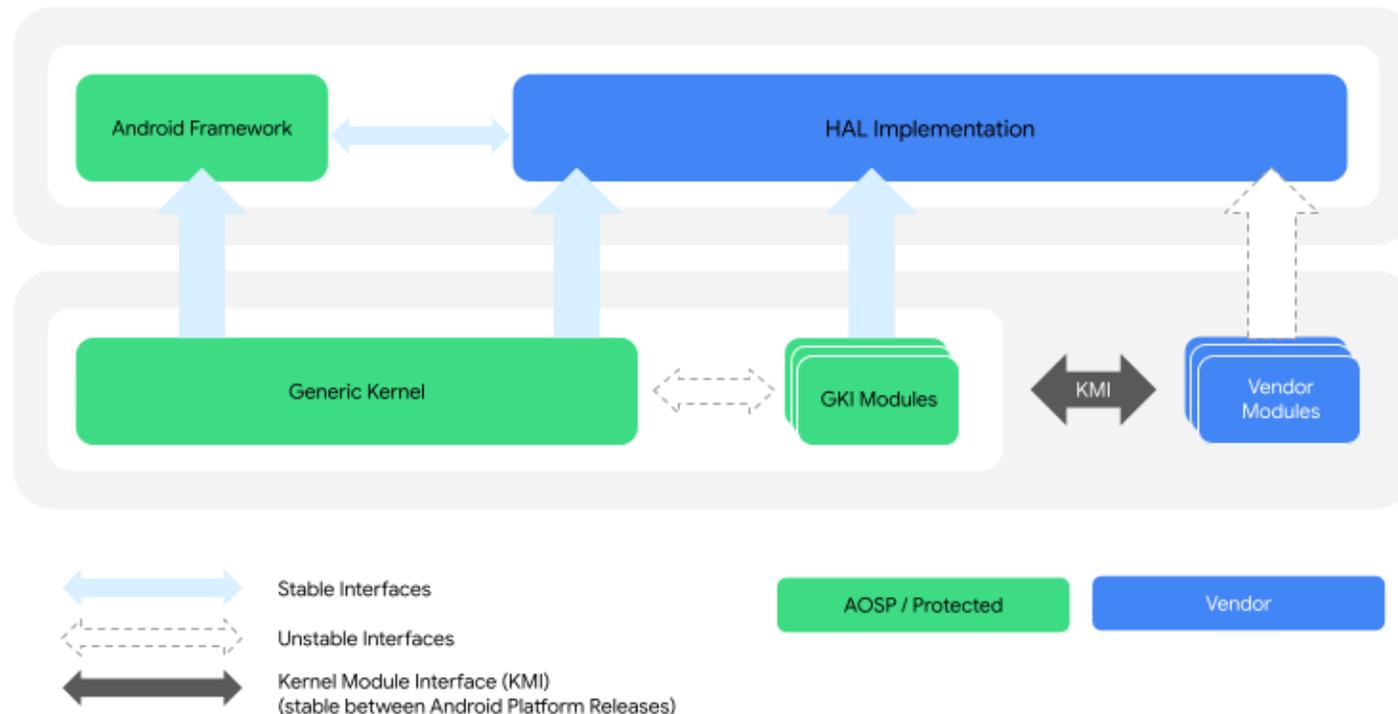
Generic Kernel Image

- Originally, every Android vendor maintained its own Linux kernel tree
 - Very slow until security updates trickled down to product kernel



Generic Kernel Image

- Kernel split into Core Kernel and Vendor Modules
 - All devices run this same Core Kernel
 - Kernel Module Interface for vendor modules
- GKI mandatory for modern phones (running Android 12+ and Linux Kernel 5.10+)
- Result: Kernel can be updated as Project Mainline module



Android Fragmentation Today

More than 50% of devices run an OS release that is older than 3.5 years!

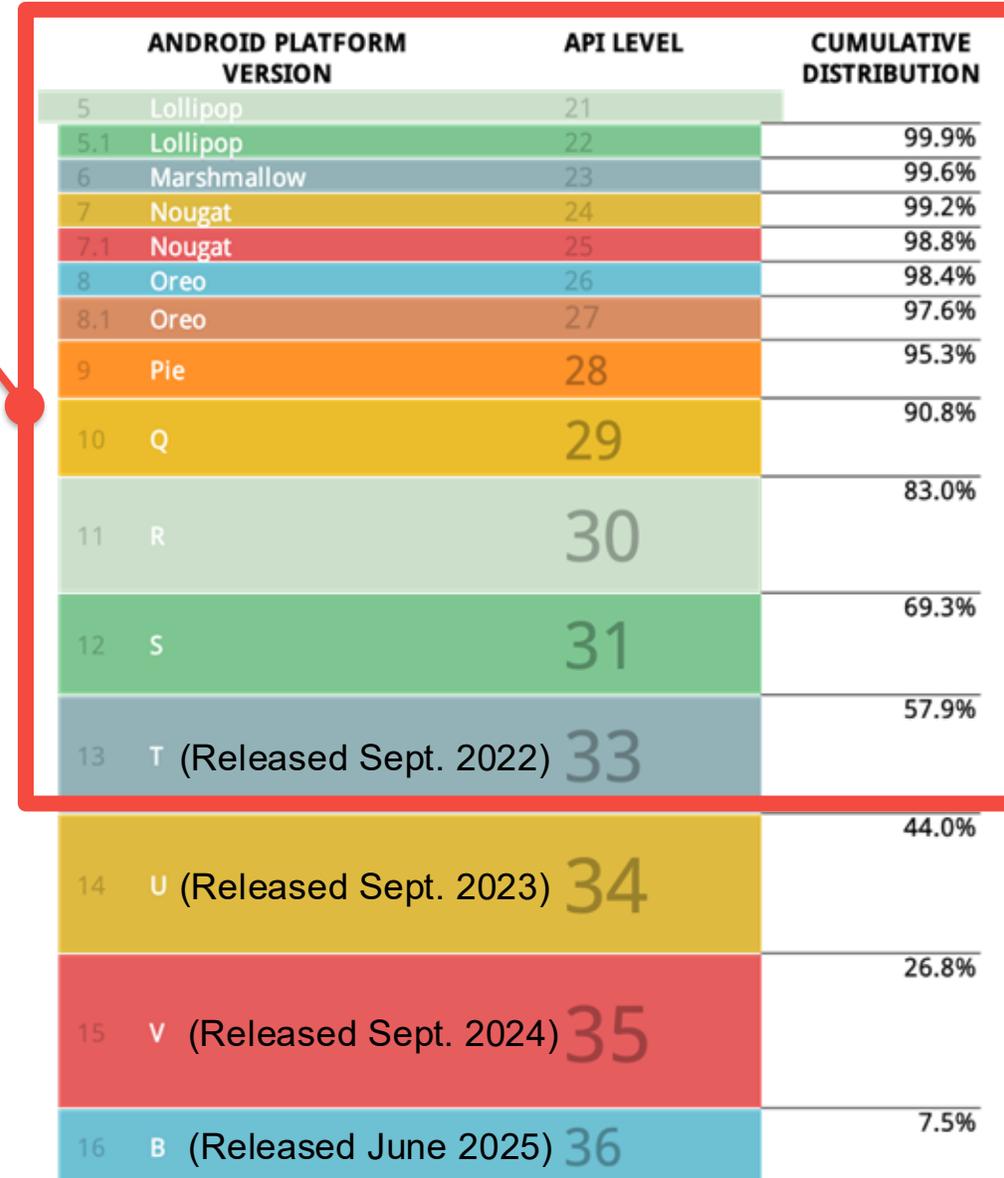
The situation is probably not that bad though

Android Security Updates

Major manufacturers release monthly security updates even after the last Android version update

Still, many devices run legacy OS versions

- Particularly cheap devices
- Known vulnerabilities!



Last updated: December 1, 2025

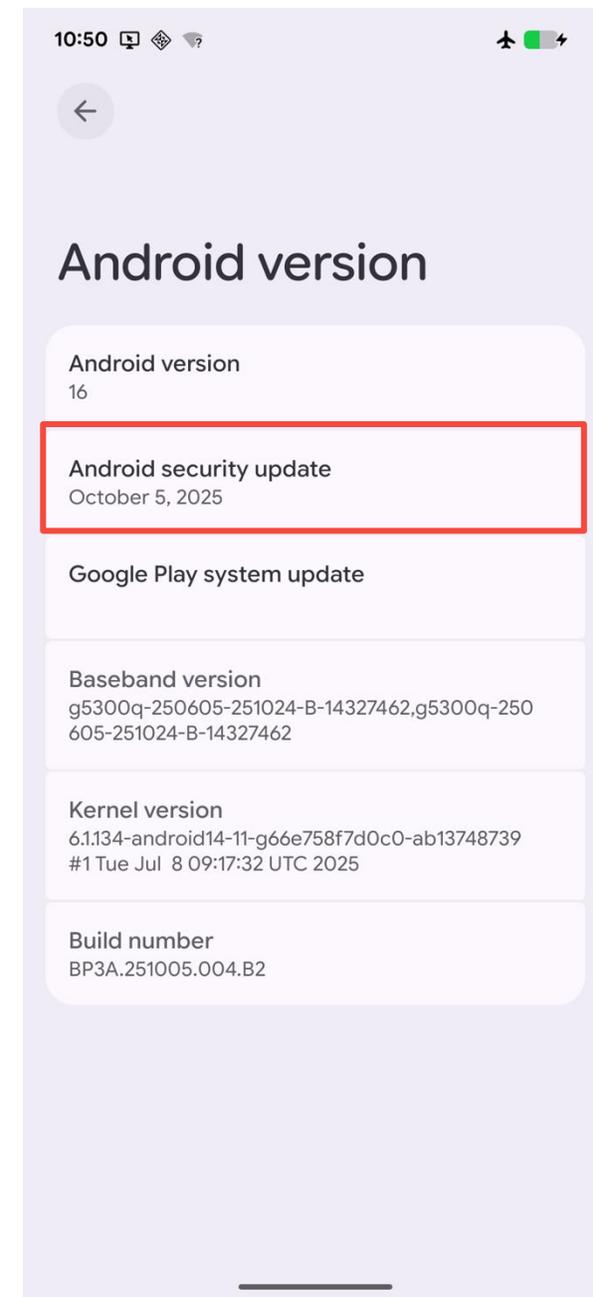
Source: Android Studio

Android Security Bulletins

- Google publishes recent security fixes on a monthly basis
 - First disclosed to vendors for publishing device updates
 - Then published on Android website

Source: <https://source.android.com/docs/security/bulletin/asb-overview>

- **Security Patch Level can be seen in Android settings**
- **However: Some vendors skip individual fixes**
 - SPL not always accurate indicator for device security
 - Source: <https://www.srlabs.de/blog-post/android-patch-gap>
- **Test your device: SnoopSnitch**
 - Abandoned nowadays



Low-Level System Security

Android Verified Boot

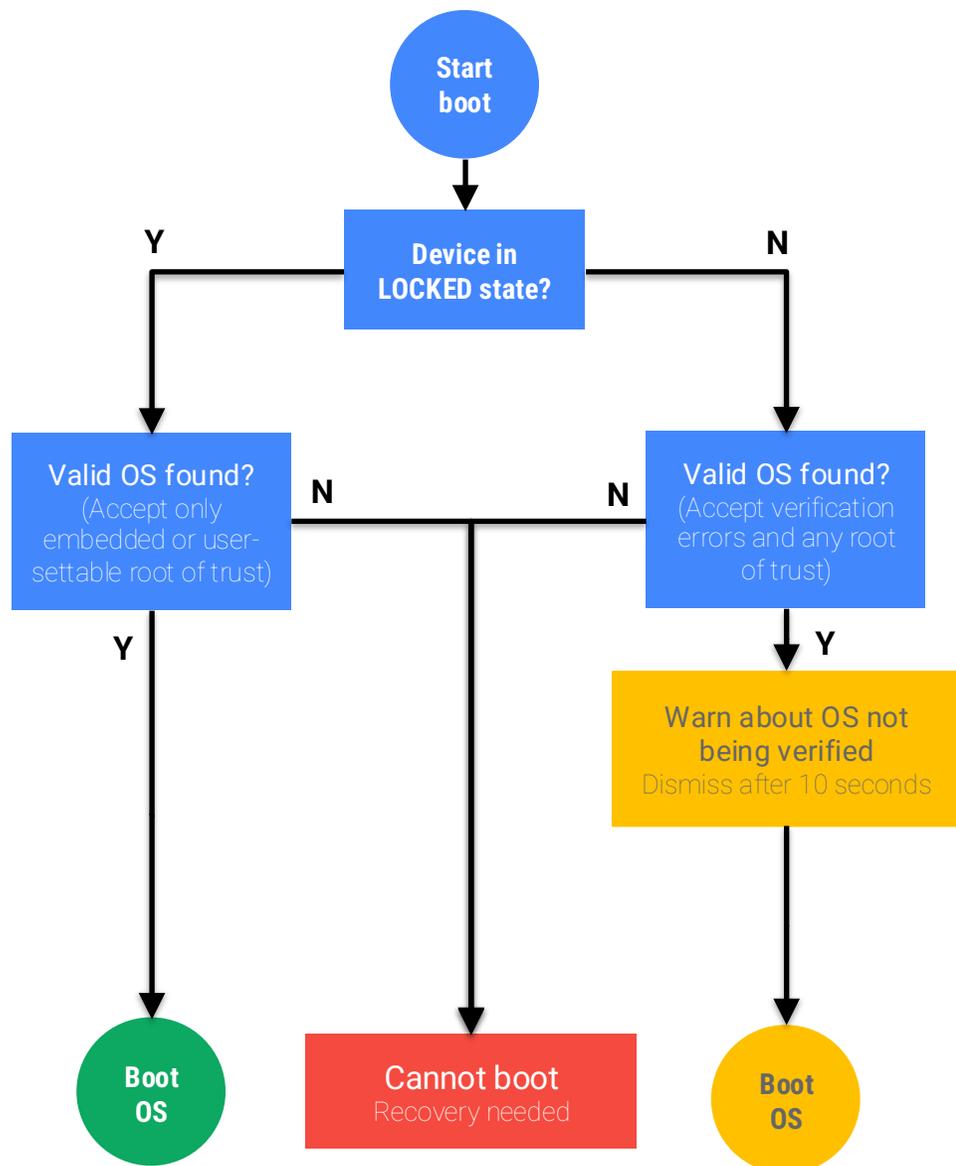
Chain of Trust from lowest-level bootloader to system partition

1. Device vendor embeds **Root of Trust** certificate in read-only storage
2. Bootloader checks signature of boot partition against Root of Trust
3. Kernel checks signature of system, vendor (& oem) partitions

How to efficiently check the signature of relatively large partitions?

- Use the Device Mapper verity (dm-verity) feature in Linux kernel
- Transparent **real-time integrity checking** of block devices
 - Prevent persistent rootkits

Verified Boot Flow



This flow is simplified

- Some devices allow user-settable Root of Trust
- Additionally: Rollback protection
- dm-verity error may reboot device

Device / bootloader state

- LOCKED/UNLOCKED
- Unlocking effectively disables signature check
- State changes erase all user data

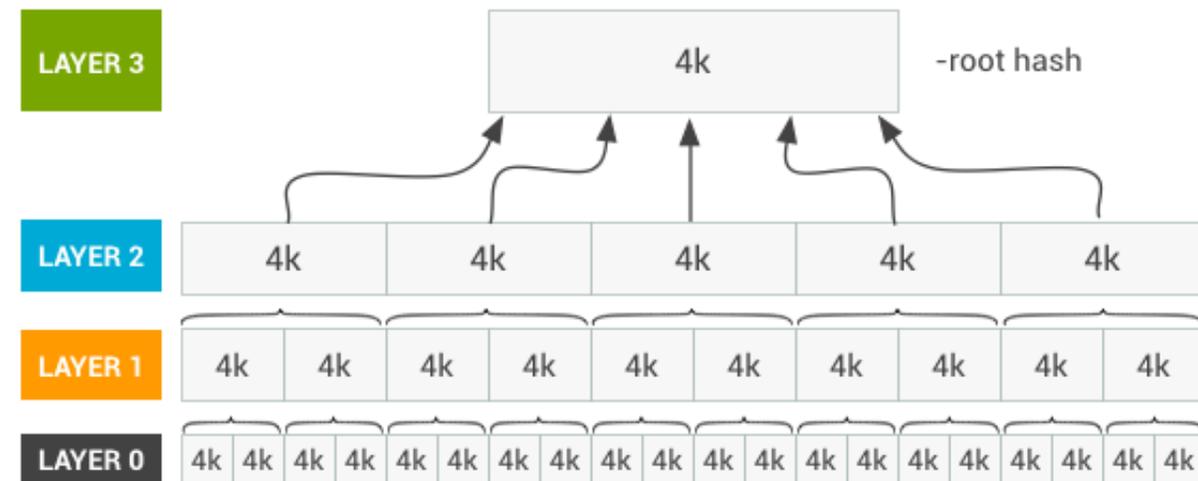
Boot state

- GREEN/YELLOW/ORANGE/RED
- Yellow (Not displayed): Custom Root of Trust
- Only red stops boot

dm-verity – Insight

Idea: Look at block device and storage layer of file system using a hash tree

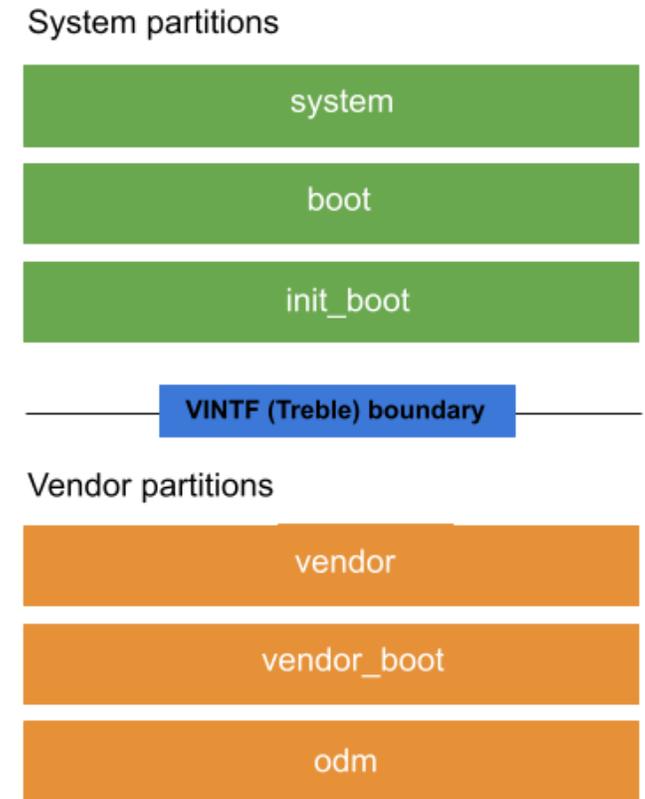
- Hash values stored in tree of pages
 - Only „root hash“ must be trusted to verify rest of tree
- Hash of a page is checked by kernel when it is accessed (always or first time)
- Modification of any 4k-block would change the „root hash“
- Verify signature of „root hash“ using public key included on boot partition
→ Confirm that device’s system partition is unchanged



Picture: source.android.com/ / Apache 2.0

Device Partitions (examples)

- System partitions (updated with OS):
 - boot: Generic kernel (GKI)
 - system: Userspace system components
- Vendor partitions (might never be updated):
 - vendor_boot: Device-specific boot resources (DTB)
 - recovery: Booted during OTA updates
 - vendor: Vendor-specific userspace system binaries
 - radio: Firmware for the baseband chip
- Nonupdatable partitions (will never be updated):
 - userdata: Installed apps, created user files, ...
 - cache: Temporary data



Seamless Updates

- OS updates used to be a critical operation
 - Failure might require setting up afresh, even broken device possible
- Some Android devices have two sets (“slots”) of partitions
 - For boot, system, vendor, radio partitions
 - Slot A can be written while slot B is booted
- Safe system updates
 - Install to inactive slot
 - Try to boot it
 - On failure, boot back to original slot

Verified Boot Flaws

- Device vendors are responsible for implementing Android Verified Boot (AVB)
- There have been vulnerabilities in vendor implementations before
- Example: Some vendor implementations accept AOSP test keys
 - Flash custom firmware despite locked bootloader

AVBTestKeyInTheWild: Bypassing Android Verified Boot Using A Firmware Supply Chain Vulnerability on Locked Devices

Ernst Leierzopf
Johannes Kepler University Linz
Linz, Austria
ernst.leierzopf@ins.jku.at

René Mayrhofer
Johannes Kepler University Linz
Linz, Austria
rene.mayrhofer@ins.jku.at

Stefan Kempinger
Johannes Kepler University Linz
Linz, Austria
stefan.kempinger@ins.jku.at

Michael Roland
Johannes Kepler University Linz
Linz, Austria
michael.roland@ins.jku.at

We identify a critical vulnerability, which we name *AVBTestKeyInTheWild*, in the Android firmware supply chain that enables attackers to flash modified firmware images onto locked devices without wiping the userdata partition. By exploiting weak signing practices, such as the use of Android Open Source Project test keys in production firmware, attackers can bypass bootloader integrity checks, retain user data, and compromise device security without user interaction. The vulnerability affects multiple manu-

Encryption Systems

Android Data Encryption Systems

- Full-Disk Encryption (FDE) Android 5.0 - 9.0
 - Encrypts complete user data partition
 - Using key derived from user passcode
 - Passcode must be entered before the device can fully boot
- File-Based Encryption (FBE) Android 7.0+
 - Every file is individually encrypted using different keys
 - If hardware support: Additional encryption of file metadata
 - Device can boot without requiring passcode (*Direct Boot*)
 - Limited context until passcode provided

File-Based Encryption

Two Areas

- Device Encrypted (DE)
 - Immediately available after device turn-on
 - „*Direct boot*“ mode: Receive phone calls, set alarms, ...
- Credential Encrypted (CE)
 - Available after user entered authentication credentials

Keys stored in `/data/misc/vold/user_keys`

→ Different subdirectory in ce and de per Android user id

```
$ ls -R /data/misc/vold/user_keys
+ ce/0/current:
    - encrypted_key
    - keymaster_key_blob
    - salt
    - secdiscardable
    - stretching
    - version
+ de/0:
    - encrypted_key
    - keymaster_key_blob
    - secdiscardable
    - stretching
    - version
```

File-Based Encryption

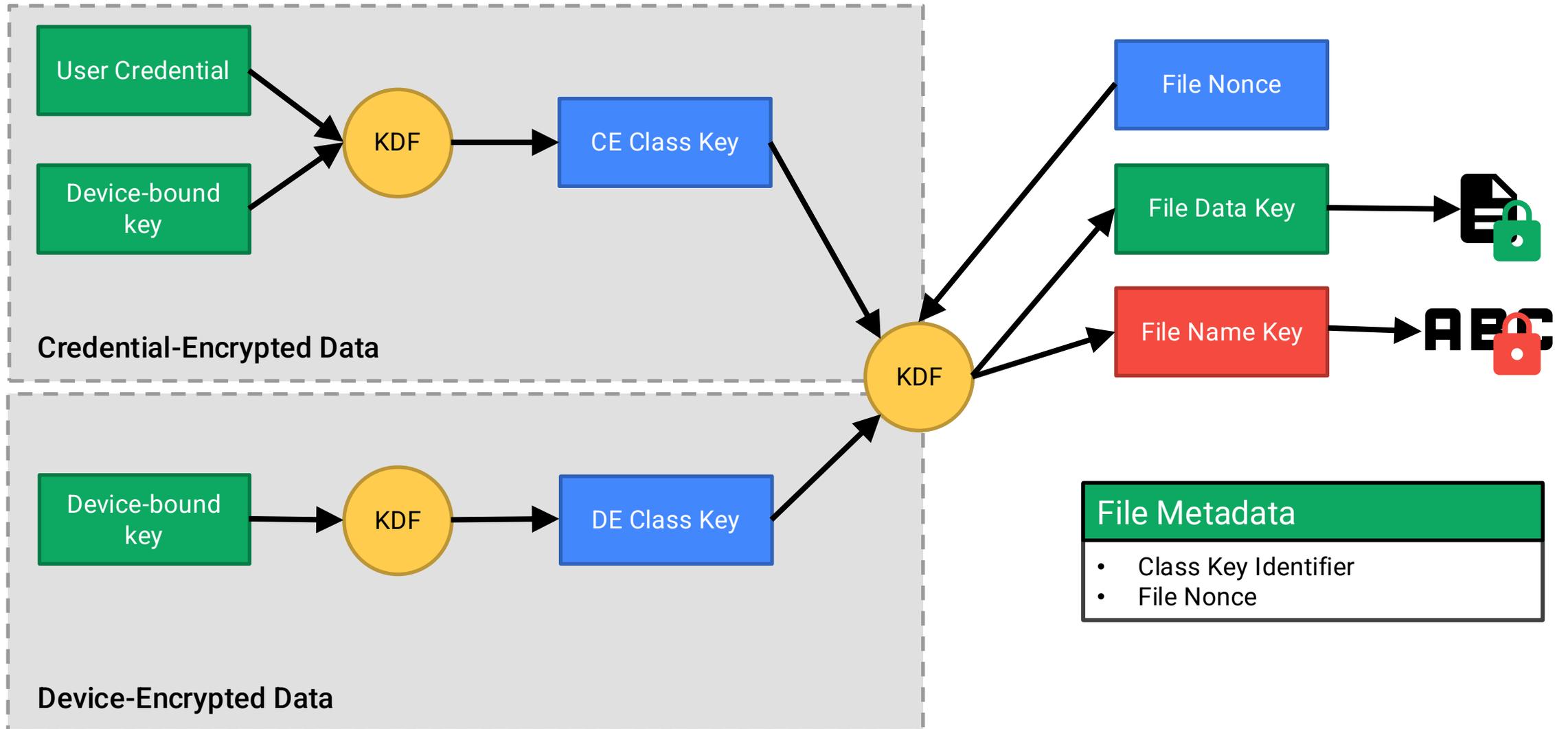
The exact encryption process is highly configurable

- Differs between vendors and Android versions

Core principles

- Lowest-level file encryption is implemented using fscrypt
 - Common Linux kernel API for file encryption across different file systems
 - Encryption metadata stored as FS attributes
- File name and contents encrypted using separate keys
 - Derived from master key and a file-specific nonce
- Master keys here: DE and CE class keys

File-Based Encryption (Simplified)



File-Based Encryption: Flaw 1

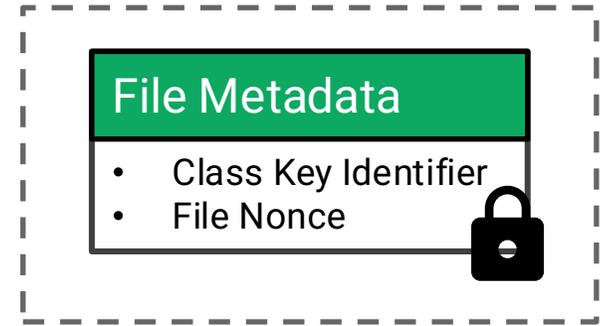
From Android's developer documentation:

Credential encrypted storage is only available after the user has successfully unlocked the device, up until when the user restarts the device again. If the user enables the lock screen after unlocking the device, this doesn't lock credential encrypted storage.

- CE keys **are not evicted until the next reboot!**
- Protection is only really effective
 - While device is completely shut down
 - BFU = Before First Unlock = Between boot and first unlock
- Key difference to how iOS Data Protection works!

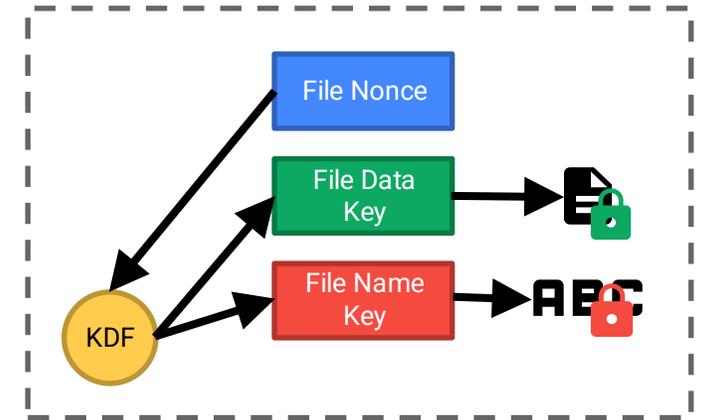
File-Based Encryption: Flaw 2

- Early implementations: File metadata not encrypted
 - File size, creation and access date
- Solution: **Metadata Encryption** Android 9+
 - Similar scheme as FDE, but only for file system metadata
 - Metadata decrypted at boot time
 - Wrapped key stored on special partition
 - Key protected by TEE, only unlocked if Verified Boot succeeds
 - Mandatory in Android 11 and later



File-Based Encryption: Flaw 3

- Class keys derived inside TEE
 - ARM TrustZone
 - Device-bound key cannot be extracted
- However, class keys may be processed by the main CPU
 - For deriving file-specific keys in kernel
 - May be compromised by vulnerable kernel
- Solution: **Some** devices employ Hardware-Wrapped Keys
 - Ephemeral wrap all keys as they pass through CPU
 - Requires inline crypto hardware for storage accesses



Android 11+

File-Based Encryption: Flaw 4

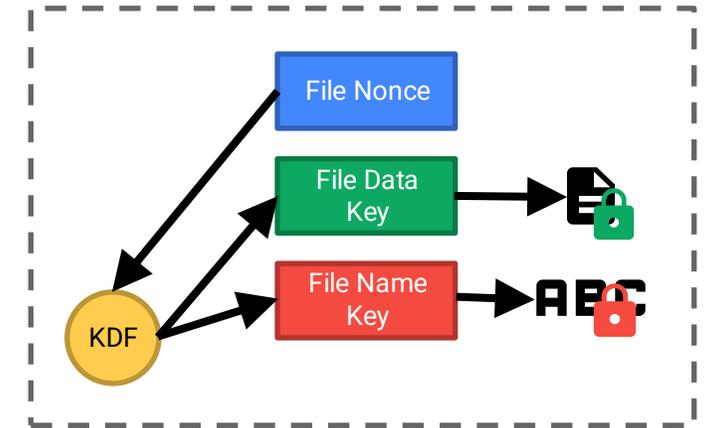
- Insecure KDF for deriving file keys

$$DEK_f = AES_{nonce_f}^{ECB}(MK)$$

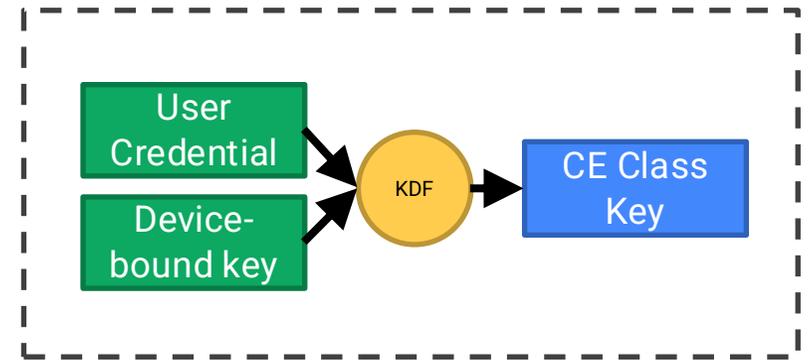
- Which can be inverted as

$$MK = AES_{nonce_f}^{ECB}(DEK_f)$$

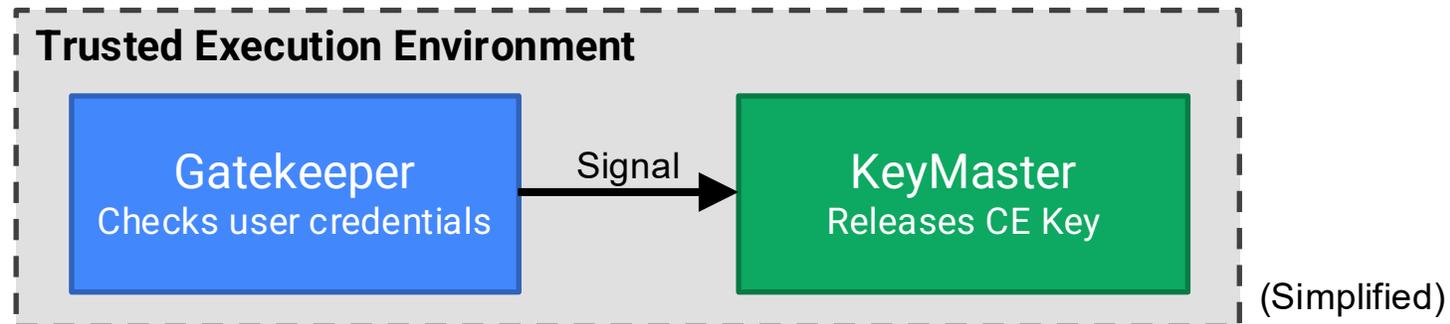
- Attack: Identify and collect all $nonce_f$ and DEK_f from RAM dump
 - (Assumes Hardware Key Wrapping is not used)
 - From dump it's not obvious which of the $nonce_f$ and DEK_f belong together
 - Calculate all potential MK candidates
 - If the same potential MK is found for two combinations of $nonce_f$ and DEK_f
 - Actual MK found!



File-Based Encryption: Flaw 5



- In some implementations, the CE key is not cryptographically bound to the user credentials



- Problem: If vulnerability in TEE found → Release CE key without credentials
- Solution: Ensure there is a cryptographic relation between user credentials and CE Class key (via KDF)

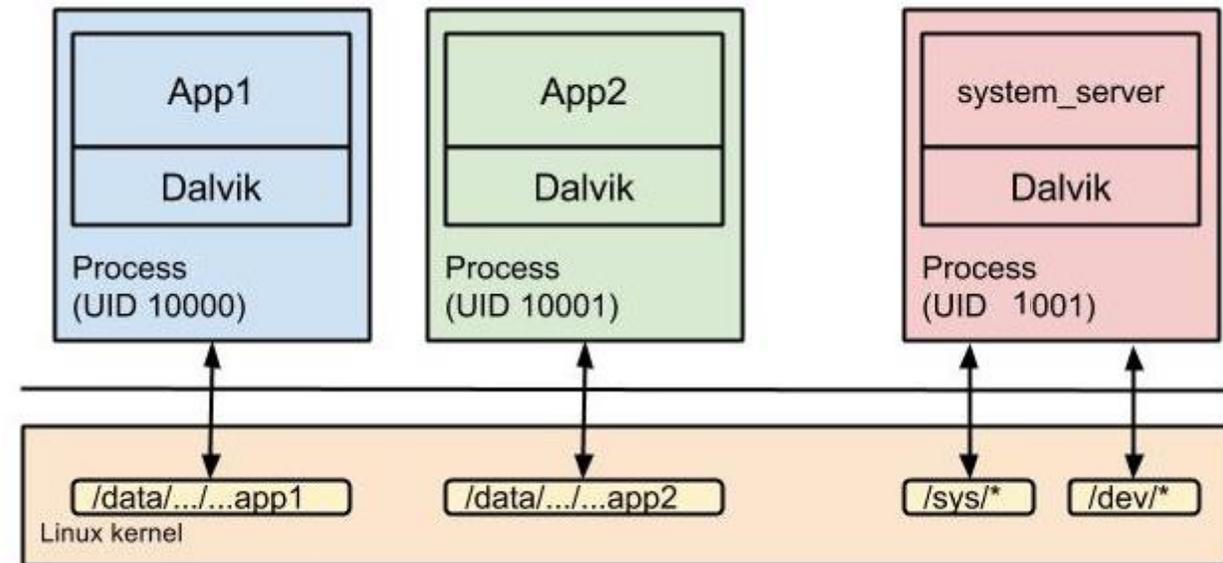
Android OS Security

Android Security Model

- Kernel-based application sandbox
 - DAC (UID, GID-based access control)
 - Dedicated, per-application Linux User ID
 - MAC (SELinux type enforcement)
- Secure IPC (Binder, Intents)
- Package Signing
 - Application packages (APKs)
 - OS update packages (OTA packages)
- Permissions: System and custom (per app)

App Sandbox

- Android assigns unique Linux user ID to each application → separate processes
→ Kernel-level application sandbox
- Security enforced at process level through standard Linux facilities (UID, GID)
- Sandbox at kernel level
→ Security model extends to native code and OS applications too
- FS permissions as a mechanism to keep files / folders separate



App Sandbox

- **Installing new apps**

- Creates new directory /data/data/<Package name>/
 - E.g. /data/data/com.whatsapp/

```
$ ls -l /data/data/  
drwx-----  4 u0_a97          u0_a97          4096 2017-01-18 14:27 com.android.calendar  
drwx-----  6 u0_a120         u0_a120         4096 2017-01-19 12:54 com.android.chrome  
...
```

- **Accessing other apps' directory → Impossible**

- Only possible for root user
- Not even possible for system applications
 - Exception: System apps with BACKUP permission can read data from apps that allow backups

Mandatory Access Control: Deny any access that is not explicitly allowed
Subjects are unable to modify the policy (cf. Discretionary Access Control!)

- Implemented as **Linux Security Module**: Hooks into kernel syscall code
- **Subject**: A Linux process
- **Object**: A system resource (file, socket, ...)
- **Domain**: Label identifying a process or set of processes
- **Modes**: Permissive (only log violations), Enforcing (disallow violations)
- **Policy**: Define allowed operations for a subject/domain and specific object

SELinux on Android

Goal: Limit the power of privilege-escalation attacks

Example: If process netd (running as root) is compromised, still do not allow it to access files only intended for process system_server

- Since Android 5.0: Enforcing Mode
- Harden Android Sandbox
- More than 60 different domains
- Policies improved with every new OS release

SELinux on Android – Sample Rules

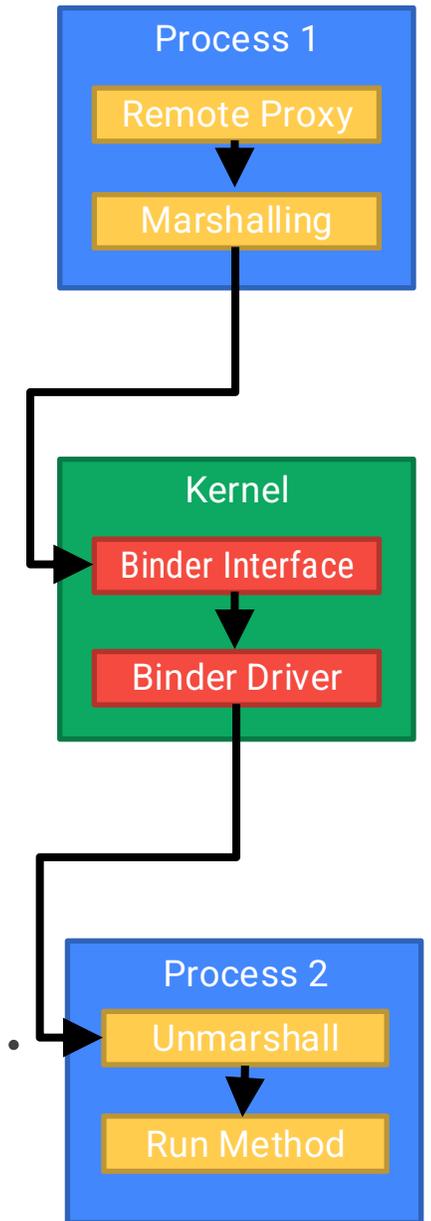
- No unlabeled files
- No ptrace
- No device node creation
- No raw I/O
- No mmap zero
- No mac_override
- No setting security properties
- No access to /data/security and /data/misc/keystore
- No /dev/mem or /dev/kmem access
- No /proc usermode helpers
- No ptrace of init
- No access to generically labeled /dev/block files
- Restrictions on mounting filesystems
- No execute of files from outside of /system
- No access to /data/properties
- No writing to /system or rootfs
- No registering of unknown services
- No entering init domain
- No /sys/kernel/debug read access
- No apps acquiring capabilities
- No raw app access to camera, microphone, NFC, radio, etc.
- No app-generic socket access
- No app/proc access to different security domains
- No access to GPS files
- Cannot disable SELinux

Meanwhile > 250 Rules

Binder

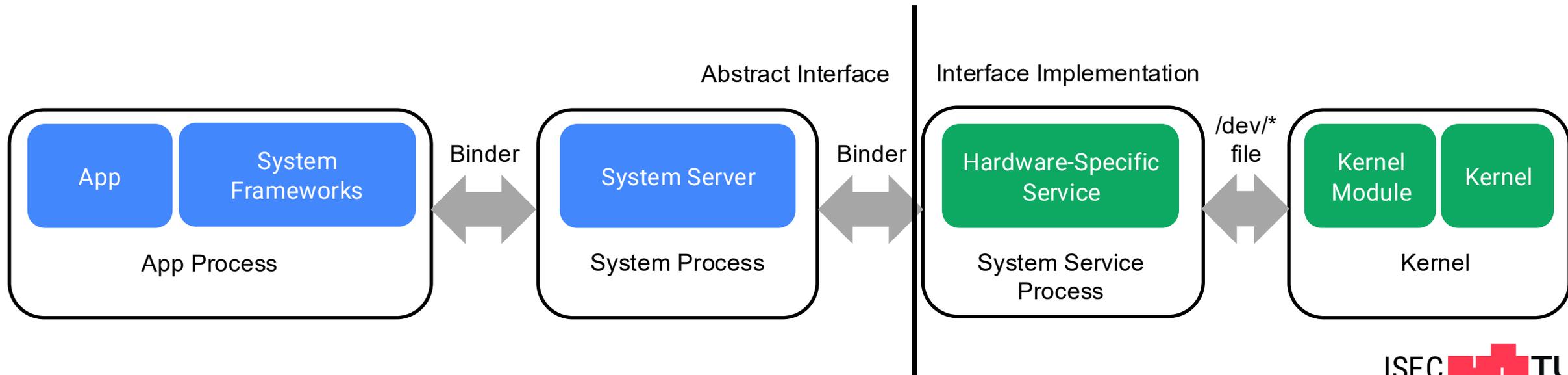
Android-specific implementation of secure and efficient RPC

- Supports passing objects and file descriptors
- Manages memory life cycle of shared objects
- Kernel passes UID of calling process to callee
 - Callee can check permissions of caller
- Proxy and Stub classes can be generated from AIDL
 - Android Interface Definition Language
- Intent, Parcel, Service, `Context.getSystemService()`, ...
 - All based on Binder functionality!



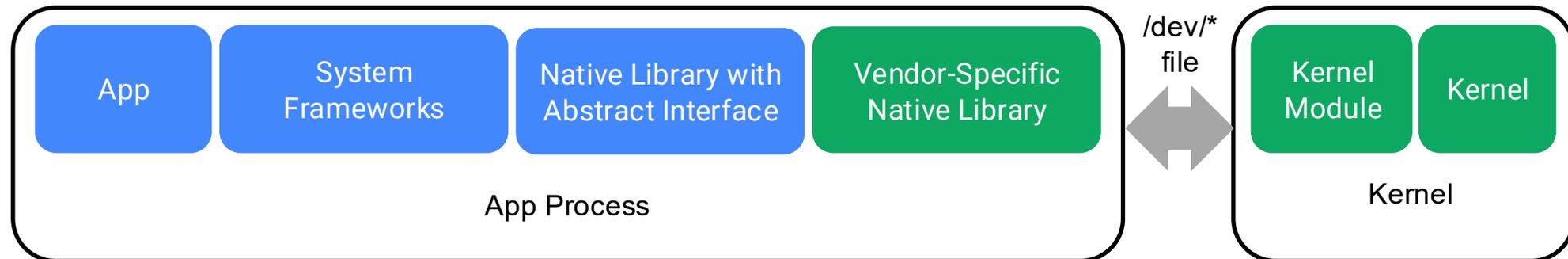
Hardware Abstraction Layers (HAL)

- Android runs on a multitude of different devices
 - Still: Share as much code between them as possible
- Solution: Android defines abstract hardware interfaces (HALs)
 - Vendors implement them for their specific hardware
 - Higher-level components do not have to deal with hardware specifics



Hardware Abstraction Layers (HAL)

- SELinux prevents apps from directly interacting with kernel device drivers
 - They have to use Binder and go via the System Server and a HAL Service
- However: The IPC is too much overhead for performance-sensitive HW
- Solution: Same Process HAL
 - Used e.g. for GPU (OpenGL/Vulkan) or NPUs



Hardware Abstraction Layers (HAL)

- Same Process HALs are detrimental to security
- Apps have direct access to **kernel attack surface**
- **Complex logic** and therefore likely vulnerabilities
- Device drivers are maintained by chip/device vendors

- SP HALs are actively exploited by attackers
 - Local Privilege Escalation (LPE)
 - From unprivileged app to kernel compromise

Project Zero

News and updates from the Project Zero team at Google

(Move to ...)

Thursday, June 13, 2024

Driving forward in Android drivers

Posted by Seth Jenkins, Google Project Zero

Introduction

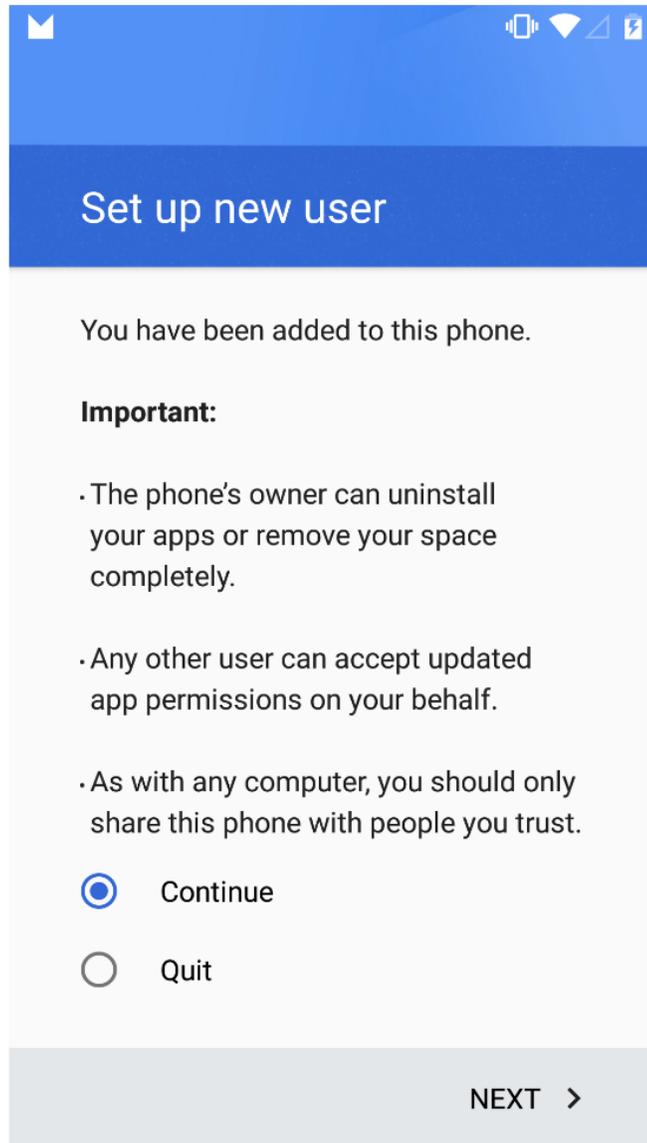
Android's open-source ecosystem has led to an incredible diversity of manufacturers and vendors developing software that runs on a broad variety of hardware. This hardware requires supporting drivers, meaning that many different codebases carry the potential to compromise a significant segment of Android phones. [There are recent public examples](#) of third-party drivers containing serious vulnerabilities that are exploited on Android. While there exists a well-established body of public ([and In-the-Wild](#)) security research on Android GPU drivers, other chipset components may not be as frequently audited so this research sought to explore those drivers in greater detail.

Source: [Google Project Zero](#)

Multi-User support

- Originally for tablets only, now for phones too (> Android 5.0)
- Users isolated by UID / GID and SELinux
- Separate settings & app data directories
 - System directory: `/data/system/users/<user ID>/`
 - App data directory: `/data/user/<user ID>/<pkg name>/`
- Apps have different UID and install state for each user
 - App UID: $uid = userId * 10000 + (appId \% 10000)$
 - Shared Apps: Install state in per-user `package-restrictions.xml`
- External storage isolation

User Types



- **Primary** user (owner)
 - Full control over device
- **Secondary** users
 - Restricted profile
 - Share apps with primary user
 - Only on tablets
 - Managed profile
 - Separate apps and data but share UI with primary user
 - Managed by Device Policy Client (DPC)
- **Guest** user
 - Temporary, restricted access to device
 - Data (session) can be deleted

Key Management

Android KeyStore

- System-managed, secure cryptographic key store
 - Hardware-backed: Trusted Execution Environment (ARM TEE)
 - Optionally: Use Secure Element if device has one („StrongBox“)
 - Accessible to apps through Java Crypto APIs
 - Create keys, perform crypto operations without exposing key material
 - Strict separation between keys of different applications
 - Though sharing keys between apps is possible
- Android OS defines the KeyMaster HAL interface
 - Vendors either provide their own KeyMaster Trusted Application (TA)
 - Or adopt the open-source Trusty OS reference implementation

KeyStore: Access Control

- Developers can limit how a new key may be accessed
 - Limit operations: E.g. only use key for signatures
 - Require user authentication (fingerprint or PIN)
 - Specify key expiration date
 - Request delay between accesses
- Some requirements are only checked in software
 - Depending on implementation

Key Store: Key Attestation

Goal: Cryptographically proof that a particular public key is hardware-backed
i.e. the corresponding private key can not be extracted

- KeyMaster can generate an X.509 certificate chain for the key
 - Also includes information on the device state, key access control, and caller app
- Chain includes device-specific certificate
- Root of chain: Google Hardware Attestation Root certificate
- **Best practice:**
 - Include the attestation certificate chain in communication to backend server
 - Only serve requests if chain successfully validated

Key Store: Fingerprint Authentication

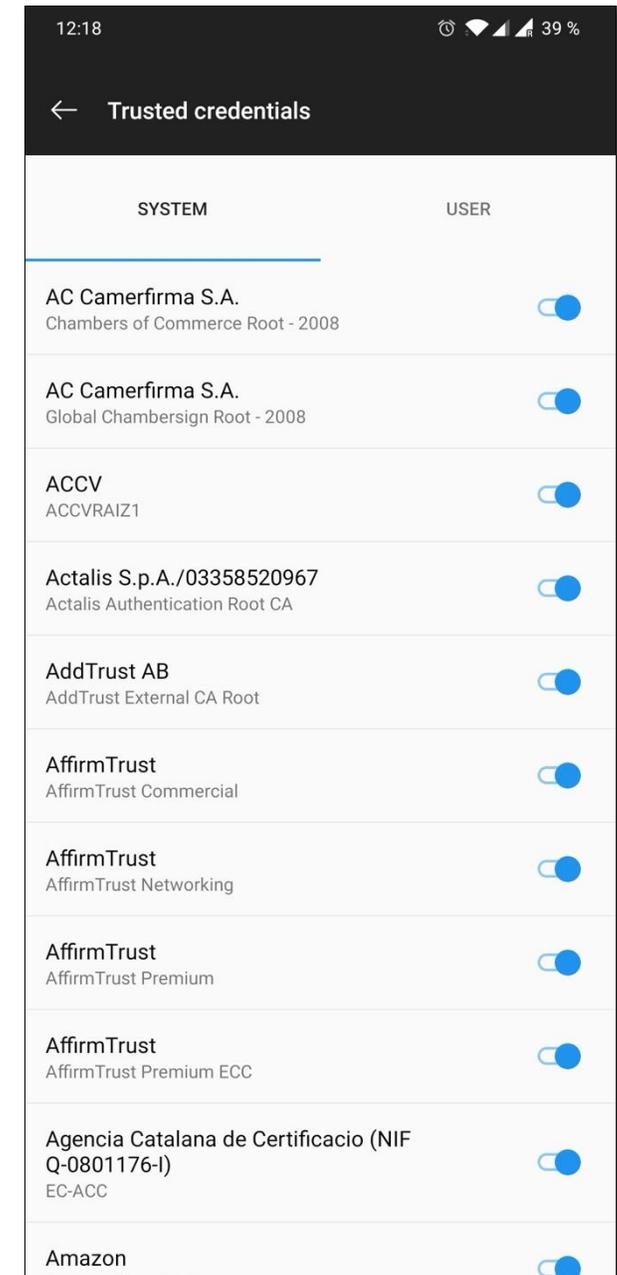
- Developers can require Fingerprint Authentication for sensitive operation
 - E.g. authorizing banking transactions
- Many app developers implement this insecurely

```
BiometricPrompt prompt = new BiometricPrompt.Builder(context).build();
prompt.authenticate(cryptoObject, executor, new BiometricPrompt.AuthenticationCallback() {
    @Override
    public void onAuthenticationSucceeded(BiometricPrompt.AuthenticationResult result) {
        // Authenticated!?
        // Only if we utilize the unlocked key in result.getCryptoObject()!!!
    }
});
```

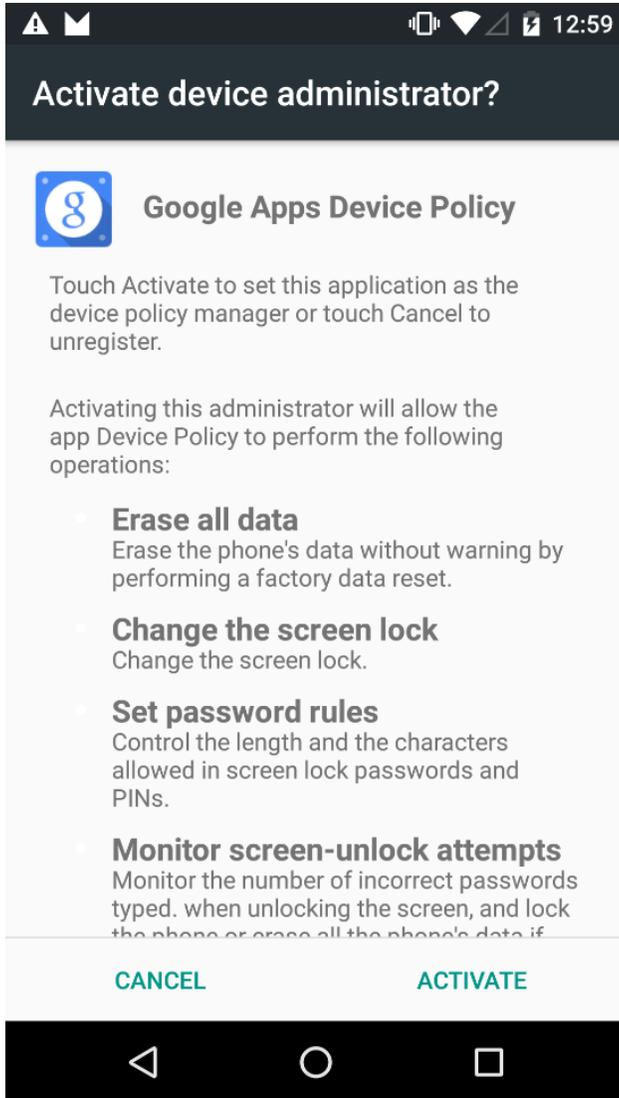
- Root attacker may modify app to just call the success callback
- **Solution:** Use the private key unlocked by the successful authentication
 - Sign server challenge, check on server, ensures TEE was actually involved

Certificates & PKI

- Android-specific trust store for TLS certificates
- Trust anchors (Root CAs)
 - Pre-installed („system certificates“)
 - User-installed („user certificates“)
- User certificates can be installed, but
 - Must be explicitly confirmed by user
 - May be rejected by individual apps



MDM



- Device security policy can be set by admin
 - Password / PIN policy
 - Device lock / unlock
 - Storage encryption
 - Camera access
- Needs to be activated by user
- Cannot be directly uninstalled
- May be required to sync account data
 - Microsoft Exchange (EAS)
 - Google Apps

Rooting

Android Rooting

Rooting refers to the process of obtaining root permissions – ie. the ability to run code (usually a shell) with **superuser** privileges.

- If bootloader unlockable:
 - Rooting doesn't require any privilege escalation exploits
 - Unlike jailbreaking on iOS
 - Simplest form of rooting: Flashing ROM that contains a su application
- Otherwise: Need privilege escalation exploit
 - “Soft-rooting”: Obtain root permission by exploiting vulnerable privileged process
 - Only really possible on legacy Android versions
 - SELinux

🛒 [Get \\$200 off the Aiper Experts Duo pool cleaner](#) >

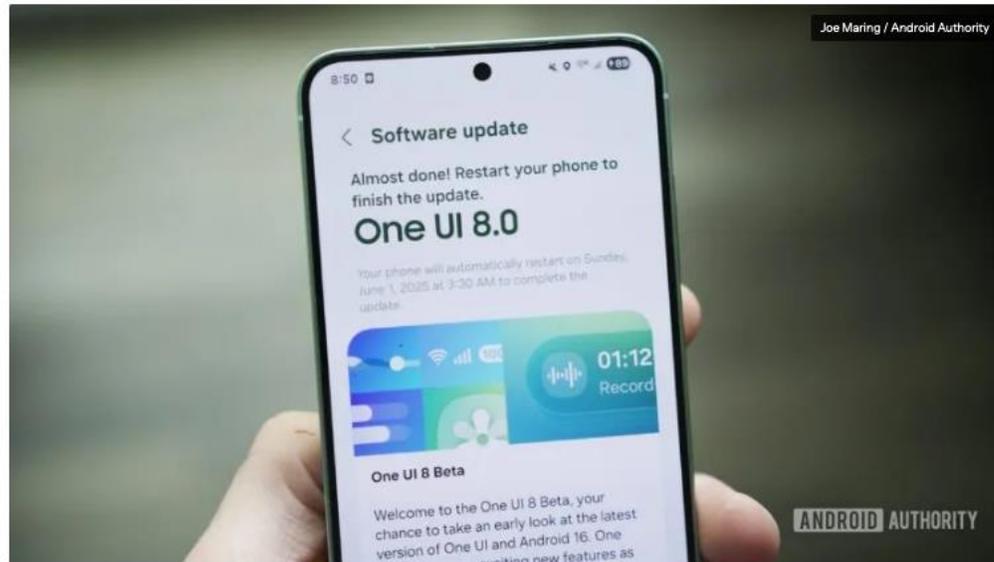
Affiliate links on Android Authority may earn us a commission. [Learn more.](#)

MOBILE

Samsung may be killing bootloader unlocking on all phones with One UI 8

Custom ROM fans, beware. Samsung may be locking down your phone for good.

By [Adamya Sharma](#) · July 27, 2025 · 11 · [Add AndroidAuthority on Google](#)



TL;DR

- Samsung appears to be disabling bootloader unlocking in One UI 8 on more devices than before.
- Previously, only US models lacked the OEM Unlock toggle, but the restriction now seems to be spreading to global models as well.
- A contributor on *XDA Developers* discovered that Samsung is likely stripping out the necessary code to unlock bootloaders entirely.

Magisk

- The most commonly used root method nowadays
- Performs system-less rooting: does not alter the system partition in any way
- **Advantage:** Improved compatibility
 - Some apps only work if system partition is original
- **How does it work?**
 - Modify boot partition to run modified init before original init
 - From there, start root daemon magiskd
 - Also change SELinux policy so that magiskd is exempt
 - Allow modules to overlay file system without actually changing it

Outlook

- 27.03.2026
 - Application Security on Android I

- 17.04.2026
 - Application Security on Android II