ISEC TU Graz

Android Platform Security

Mobile Security 2025

Florian Draschbacher
florian.draschbacher@tugraz.at

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

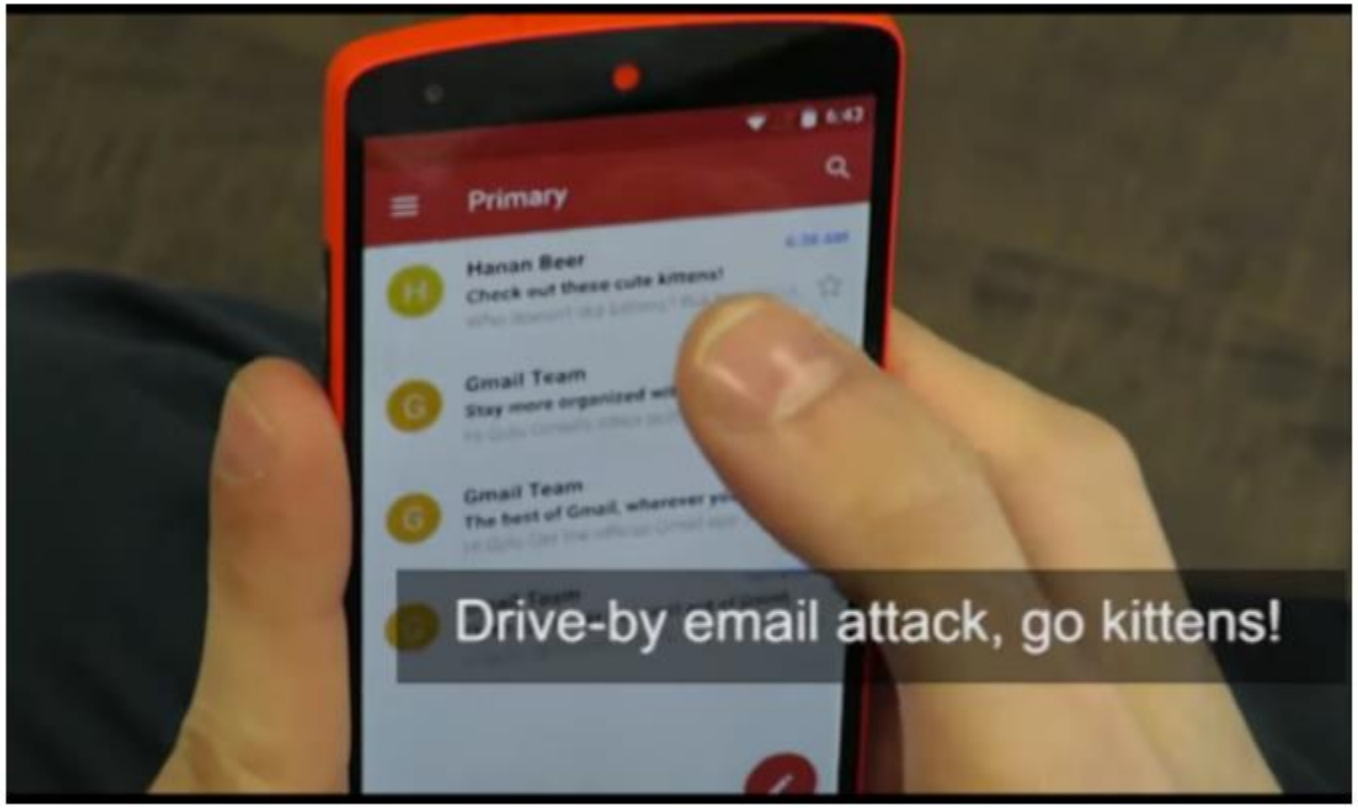
Outline

- 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

STAGEFRIGHT

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

ISEC TU Graz

SECURITYWEEK
CYBERSECURITY NEWS, INSIGHTS & ANALYSIS

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

Source: securityweek.com

ISEC TU Graz

ZDNET

Home / Tech / Security

8% of all Google Play apps vulnerable to old security bug

Devs have not updated a crucial library inside their apps, leaving users exposed to dangerous attacks. Some of the vulnerable apps include Microsoft's Edge browser, Grindr, OKCupid, and Cisco Teams.

Written by **Catalin Cimpanu,** Contributor on Dec. 3, 2020

Source: zdnet.com

What?

Attackers could inject code into Android apps

How?

- Vulnerabilities in Google Play Core library
  - **Inject DEX file through unprotected service**
  - **Exploit path traversal to enforce trust into DEX file**

Result

- Remote code execution in context of vulnerable app
- Needs to be patched by app devs

ISEC TU Graz

SECURITYWEEK
CYBERSECURITY NEWS, INSIGHTS & ANALYSIS

Google Pixel Vulnerability Allows Recovery of Cropped Screenshots

A vulnerability in Google Pixel phones allows for the recovery of an original, unedited screenshot from the cropped version.

By Ionut Arghire
March 21, 2023

A vulnerability lurking in Google's Pixel phones for five years allows for the recovery of an original, unedited screenshot from the cropped version of the image.

Referred to as aCropalypse and tracked as CVE-2023-21036, the issue resides in Markup, the image-editing application on Pixel devices, which fails to properly truncate edited images, making the cropped data recoverable.

Reverse engineers Simon Aarons and David Buchanan, who identified the bug, point out that the bug has existed since 2018 and that it was the result of a code change that Markup did not adhere to.

Specifically, when switching from Android 9 to Android 10, the *parseMode()* function was modified to overwrite a file with a truncated one if the argument 'wt' was passed

What?

Cropped screenshots still contained parts of uncropped contents

How?

- An Android update changed the default value for an API
- File accesses through the ContentProvider API no longer truncated file by default

Result

- Attacker could extract sensitive information from cropped screenshots

Source: securityweek.com

BLEEPINGCOMPUTER

Exploit released for Android local elevation flaw impacting 7 OEMs

By **Bill Toulas**

January 31, 2024 02:15 PM 2

A proof-of-concept (PoC) exploit for a local privilege elevation flaw impacting at least seven Android original equipment manufacturers (OEMs) is now publicly available on GitHub. However, as the exploit requires local access, its release will mostly be helpful to researchers.

Tracked as CVE-2023-45779, the flaw was discovered by Meta's Red Team X in early

What?

Attackers could install malicious updates to system modules

How?

- Multiple vendors used test keys to sign APEX files
- These test keys are publicly available in the Android source code

Result

- Attackers could sign malicious APEX files using keys accepted by production devices

Source: bleepingcomputer.com

ISEC TU Graz

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, …)

Source: source.android.com

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

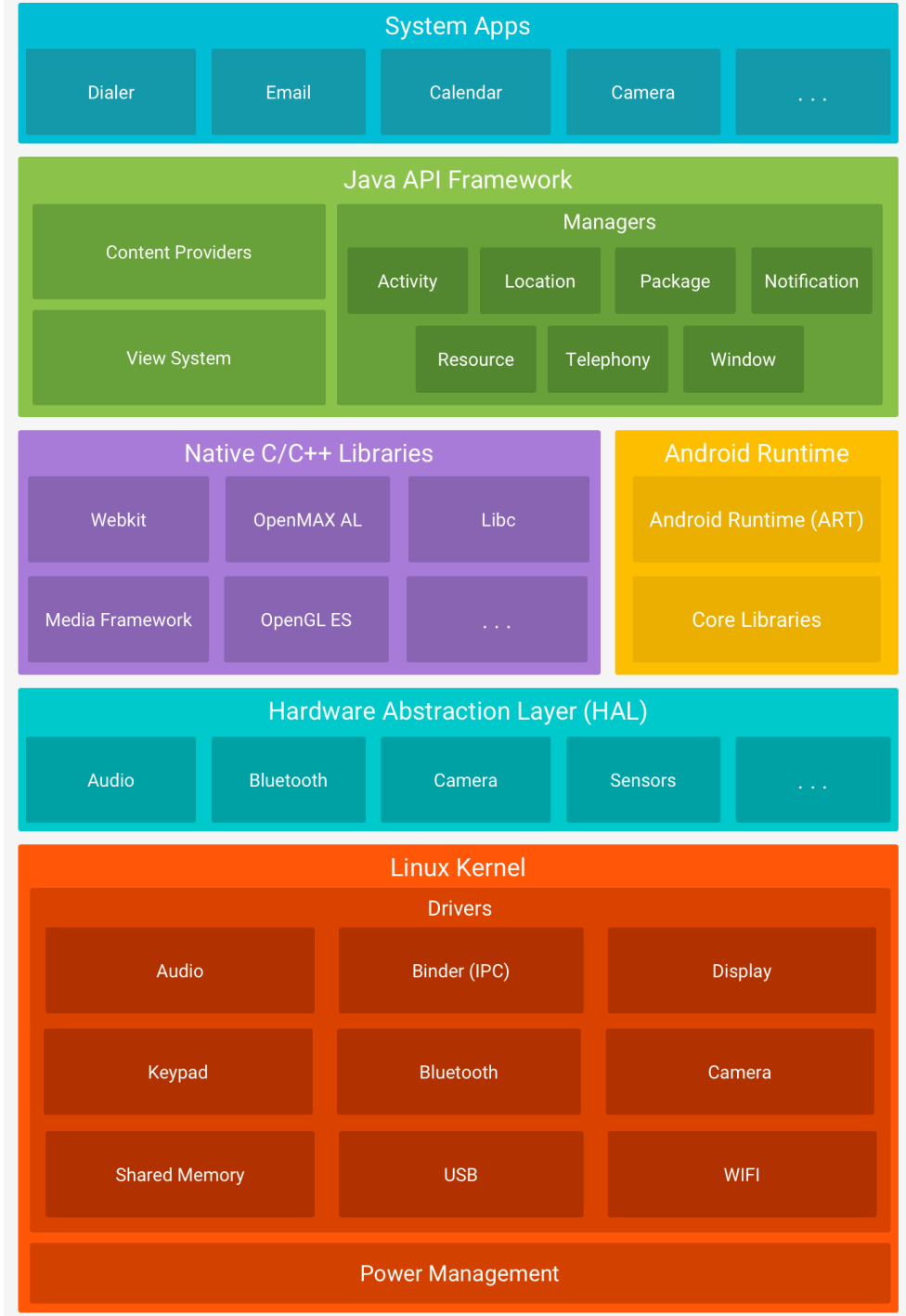
- Secure Element
  - Secure environment runs on a dedicated CPU
  - E.g. Google Titan M2 in Google devices starting from Pixel 6

Sources: security.googleblog.com, security.googleblog.com, developer.android.com

Android System Architecture

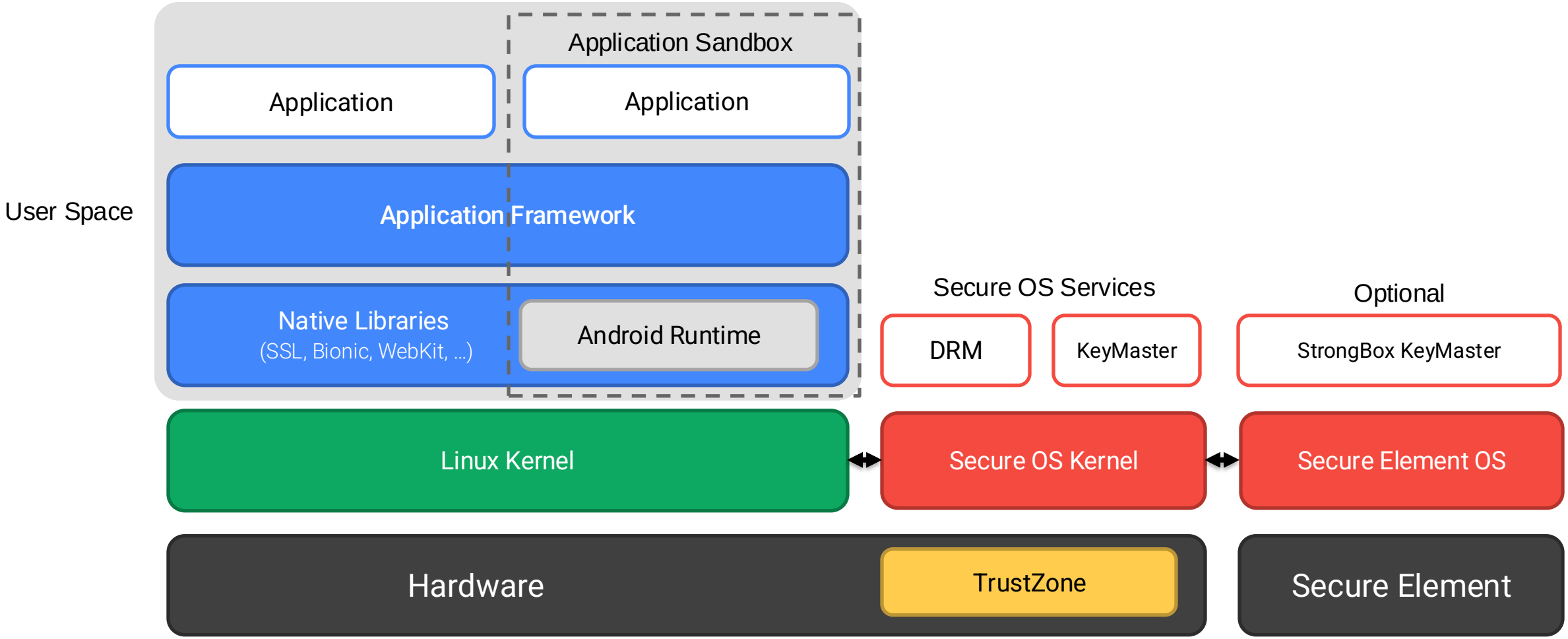
- Linux kernel
  - Device drivers
  - POSIX interface
  - Binder IPC
  - Low Memory Killer

- Userspace
  - HAL (Hardware Abstraction Layer)
  - Android Runtime
  - System Services
  - Application Framework



Picture: android.com / Apache 2.0

Android Security Architecture

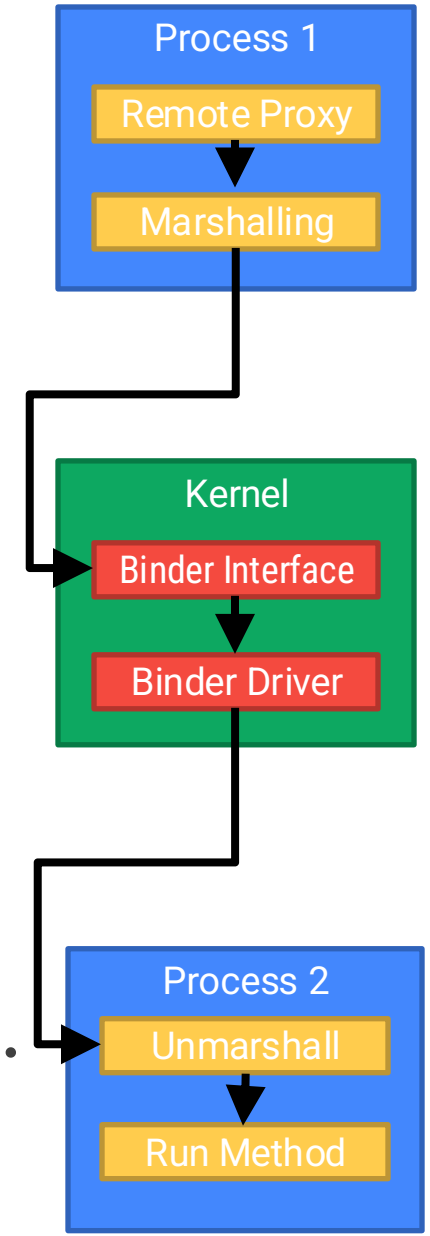


Binder

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!



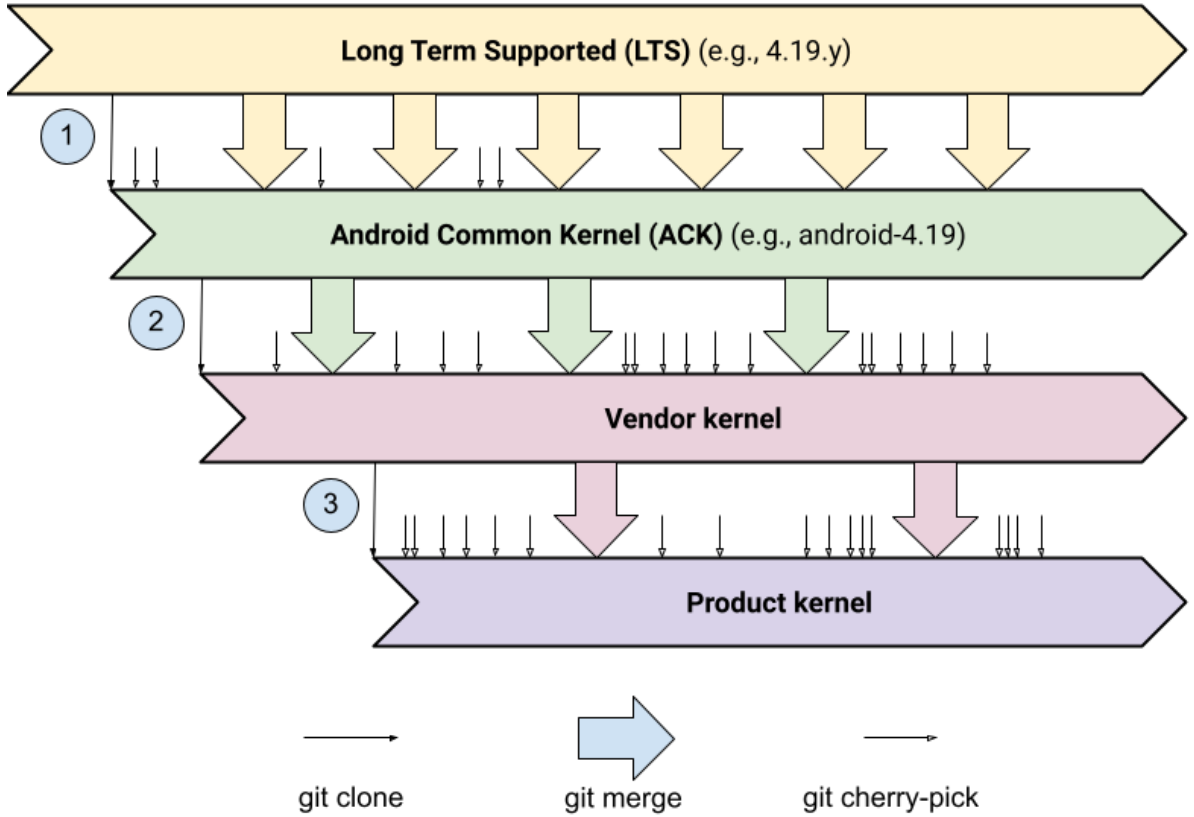
Android Fragmentation

- 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

Sources: android-developers.googleblog.com, source.android.com

Generic Kernel Image

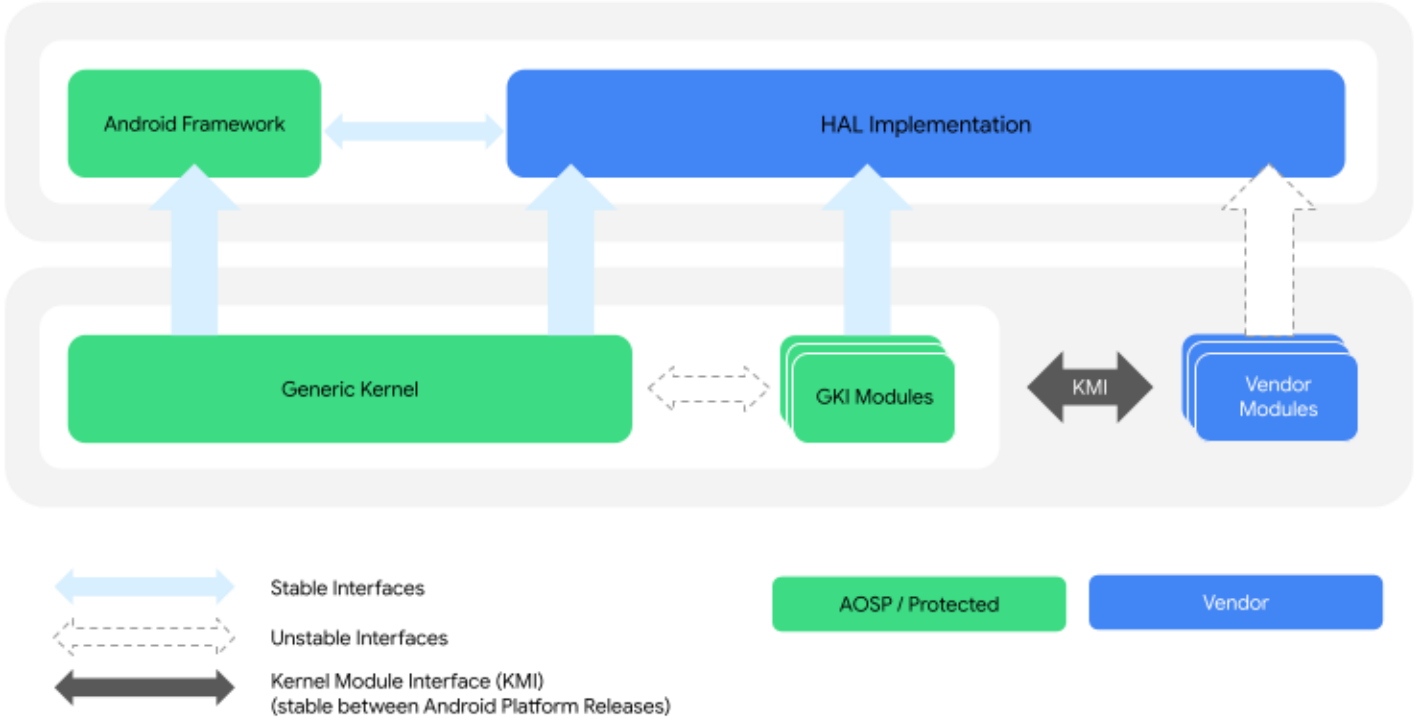
- Originally, every Android vendor maintained its own Linux kernel tree
  - Very slow until security updates trickled down to product kernel



Sources: source.android.com

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



Sources: source.android.com

Android Fragmentation Today

More than 50% of devices run an OS release that is older than 4 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!

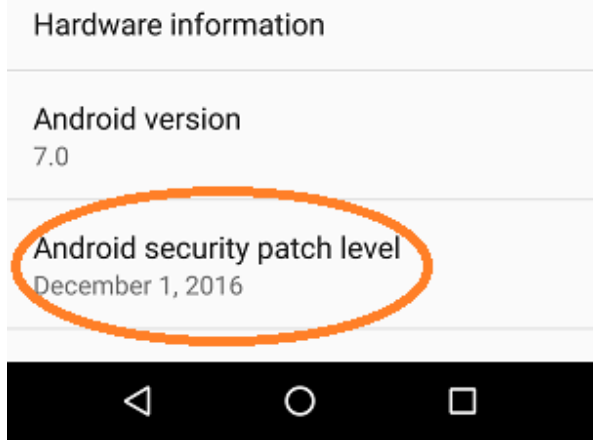
ANDROID PLATFORM VERSION		API LEVEL	CUMULATIVE DISTRIBUTION
4.4	KitKat	19	
5	Lollipop	21	99,7%
5.1	Lollipop	22	99,6%
6	Marshmallow	23	98,8%
7	Nougat	24	97,4%
7.1	Nougat	25	96,4%
8	Oreo	26	95,4%
8.1	Oreo	27	93,9%
9	Pie	28	89,6%
10	Q	29	81,2%
11	R (Released Sept. 2020)	30	67,6%
12	S (Released Sept. 2021)	31	48,6%
13	T (Released Sept. 2022)	33	33,9%
14	U (Released Sept. 2023)	34	13,0%

Last updated: May 1, 2024

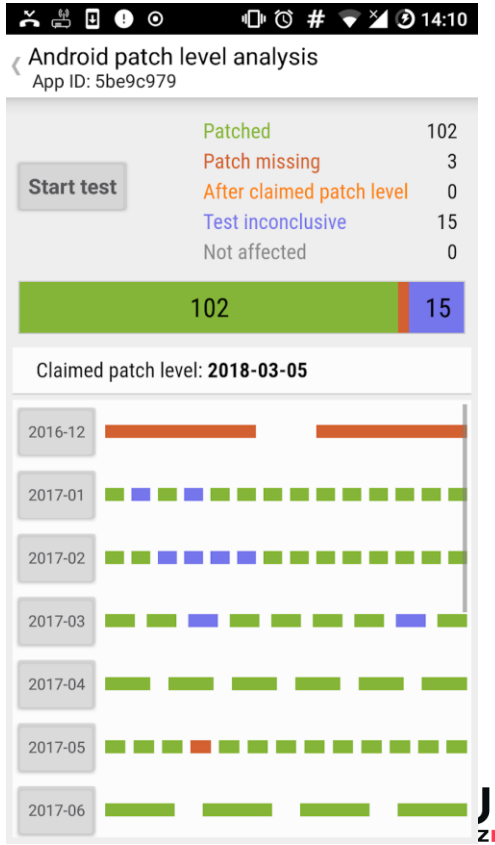
Source: Android Studio

ISEC TU Graz

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 on device**



- **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**

Low-Level System Security

Verified Boot

Chain of Trust from lowest-level bootloader to system partition

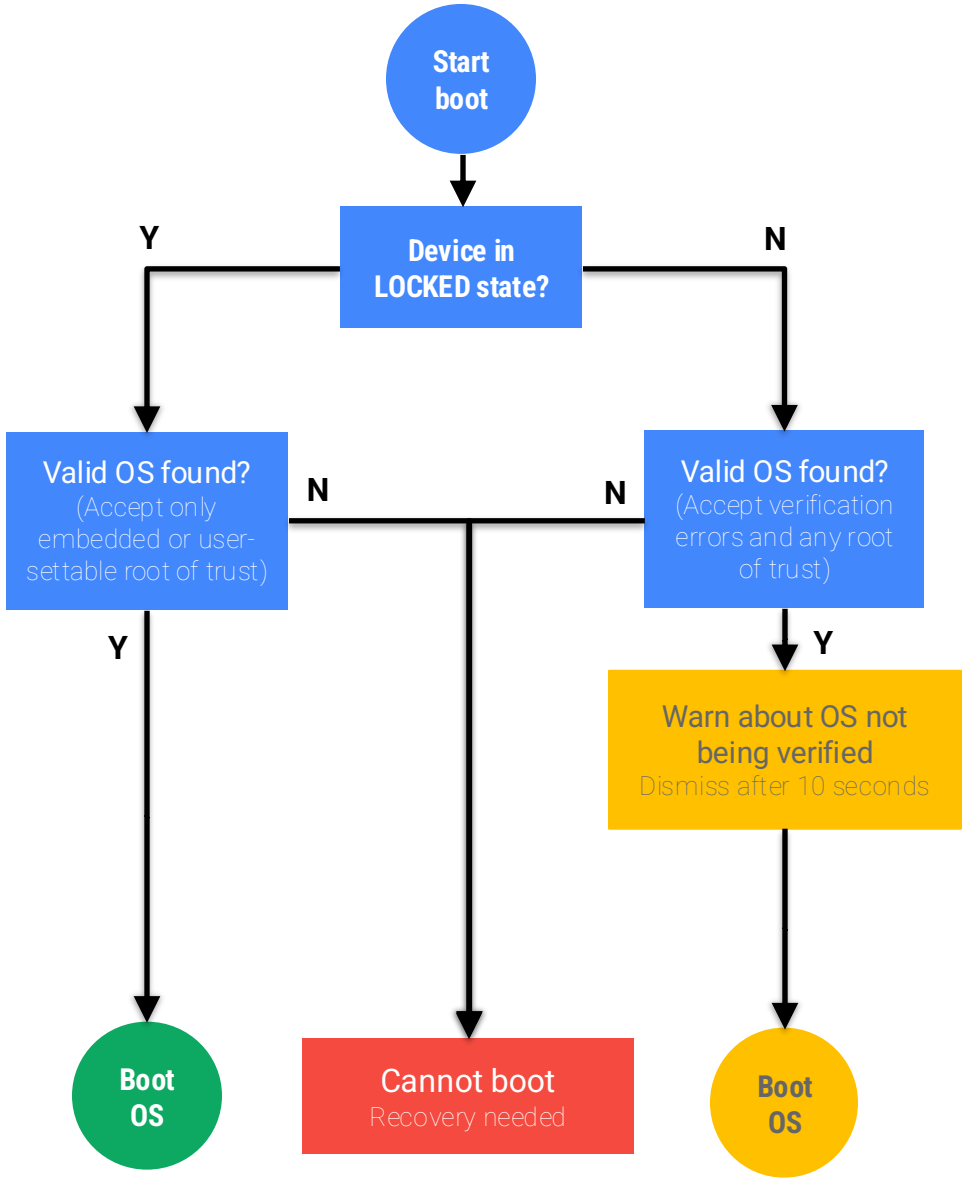
1. Device vendor embeds Root of Trust certificate in read-only storage
2. Bootloader checks signature of boot partition against Root of Trust
3. Kernel checks signature of system, vendor (& oem) partitions

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

Source: source.android.com

Verified Boot Flow



This flow is simplified

- Some devices allow changing 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

ISEC TU Graz

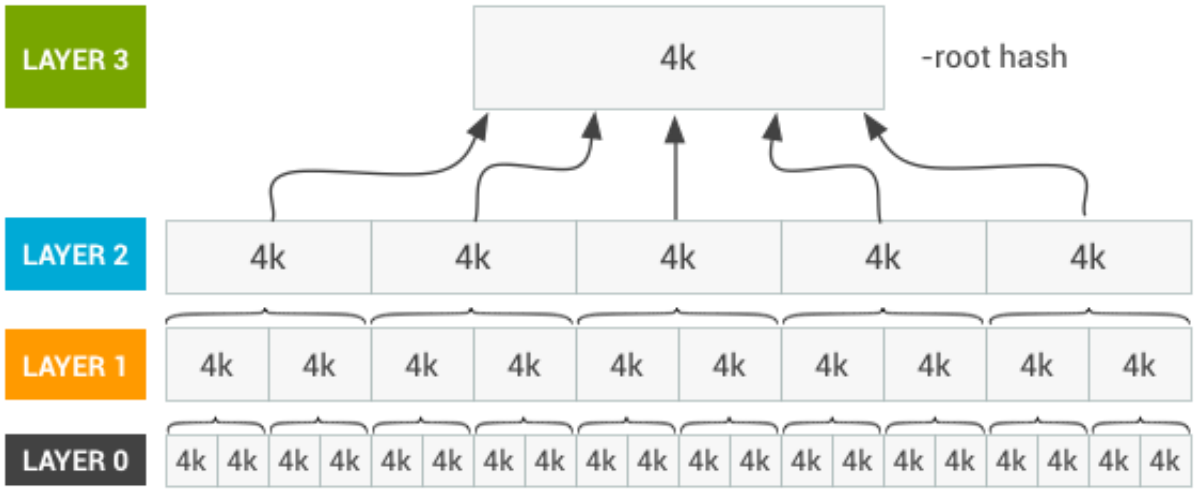
dm-verity – Insight

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

Source: source.android.com

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

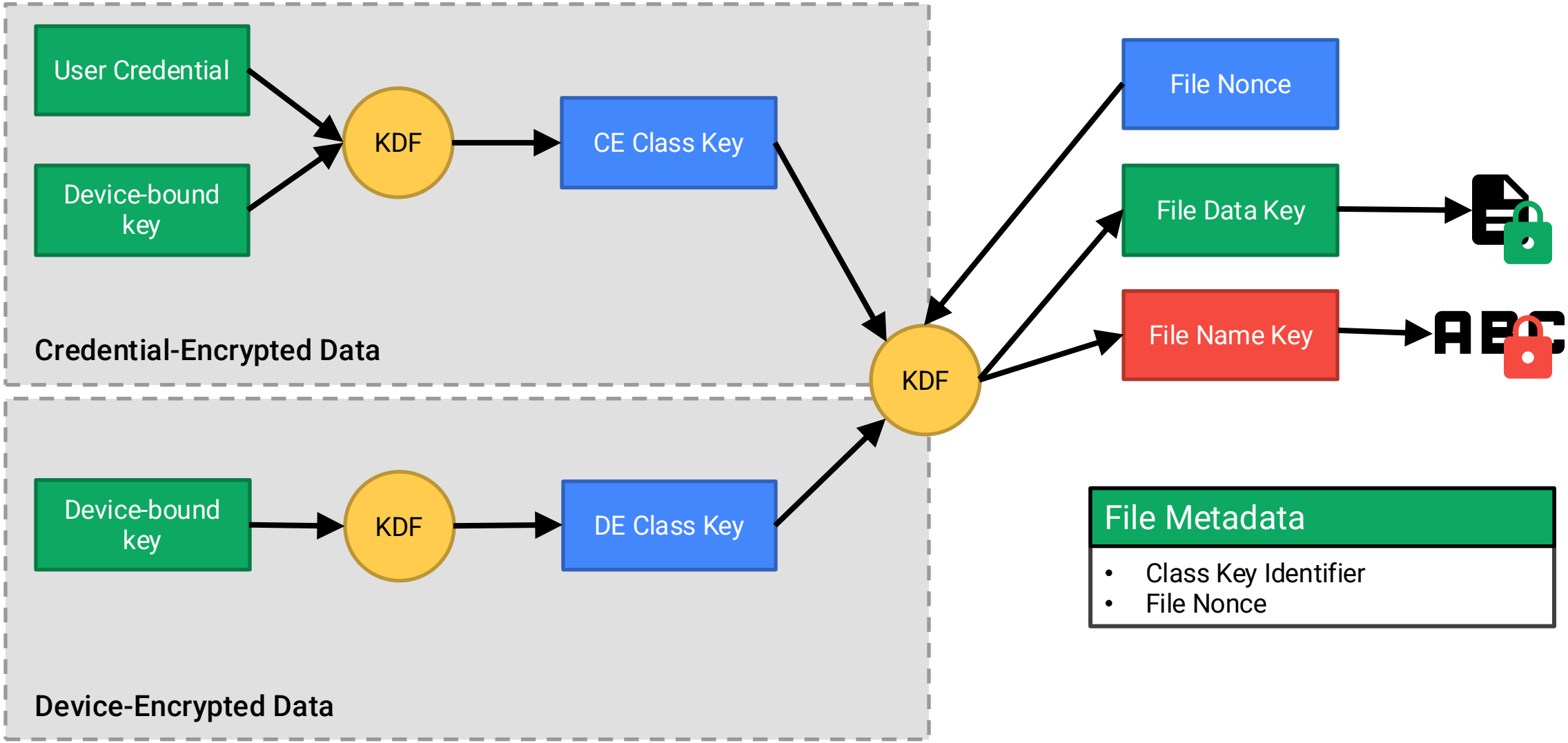
The exact encryption process is highly configurable

- 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 Metadata

- Class Key Identifier
- File Nonce

File-Based Encryption: Flaw 1

From Android's developer documentation:

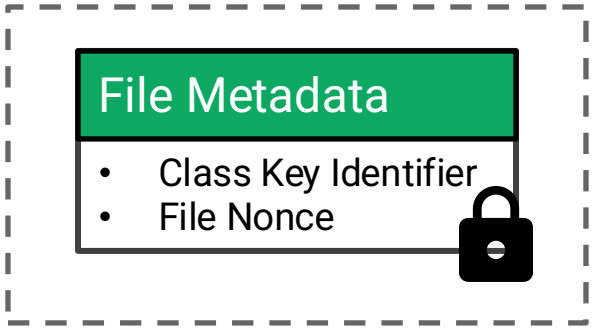
> Credential encrypted storage is only available after the user has successfully unlocked the device, up until when the user restarts the device again. If the user enables the lock screen after unlocking the device, **this doesn't lock credential encrypted storage**.

- CE keys **are not evicted until the next reboot**!
- Protection is only really effective
  - While device is completely shut down
  - Between boot and first unlock

- Key difference to how iOS Data Protection works!

Source: developer.android.com

File-Based Encryption: Flaw 2

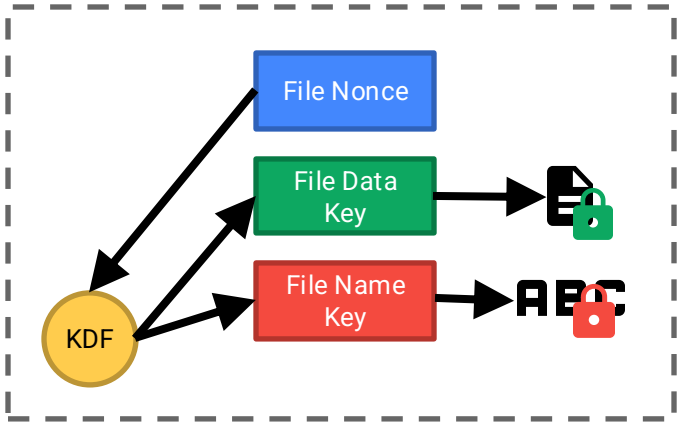


- Early implementations: File metadata not encrypted
  - File size, creation and access date

- 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

Source: source.android.com

File-Based Encryption: Flaw 3



- Class keys derived inside TEE
  - ARM TrustZone
  - Device-bound key cannot be extracted

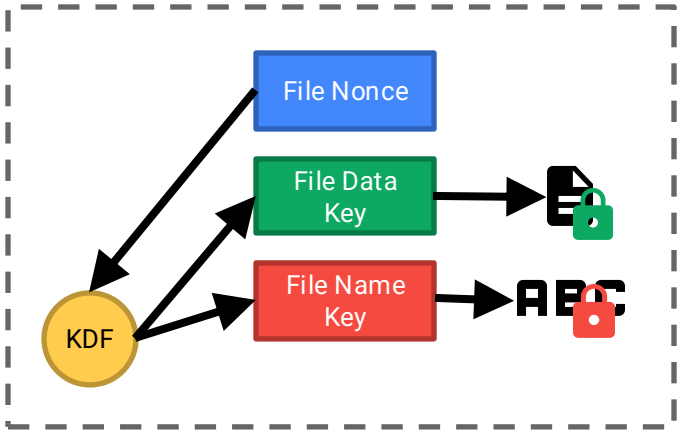
- However, class keys may be processed by the main CPU
  - For deriving file-specific keys in kernel
  - May be compromised by vulnerable kernel

- Solution: **Some** devices employ Hardware-Wrapped Keys
  - Ephemerally wrap all keys as they pass through CPU
  - Requires inline crypto hardware for storage accesses

Android 11+

Source: source.android.com

File-Based Encryption: Flaw 4



- Insecure KDF for deriving file keys

$$DEK_f = AES^{ECB}_{nonce_f}(MK)$$

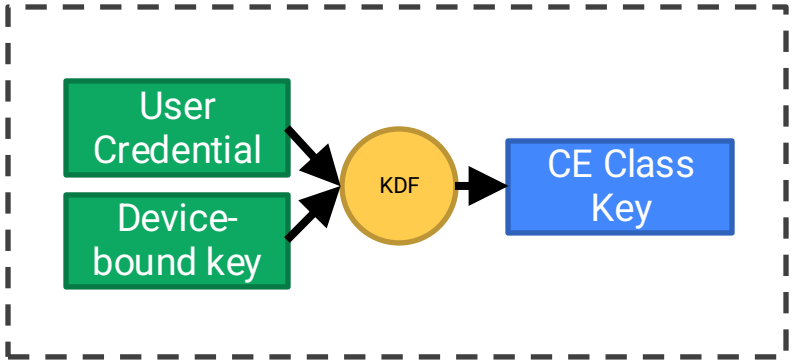
- Which can be inverted as

$$MK = AES^{ECB}_{nonce_f}(DEK_f)$$

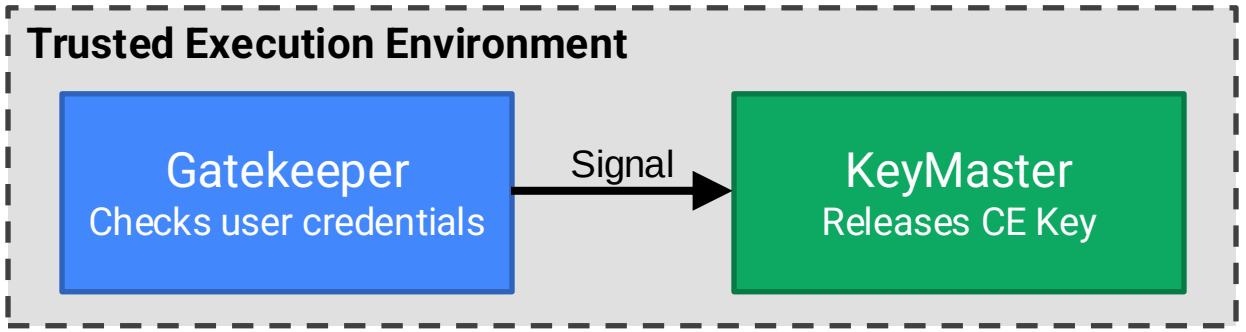
- Attack: Identify and collect all $nonce_f$ and DEK_f from memory dump
  - (Assumes Hardware Key Wrapping and Metadata Encryption is not used)
  - From dump it's not obvious which of the $nonce_f$ and DEK_f belong together
  - Calculate all potential MK candidates
  - If the same potential MK is found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

Source: Tobias Groß et al.: One key to rule them all: Recovering the master key from RAM to break Android's file-based encryption

File-Based Encryption: Flaw 5



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



(Simplified)

- 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)

Source: qualcomm.com

Android OS Security

Android Security Model

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

App Sandbox

- Android assigns unique Linux user ID to each application → separate processes

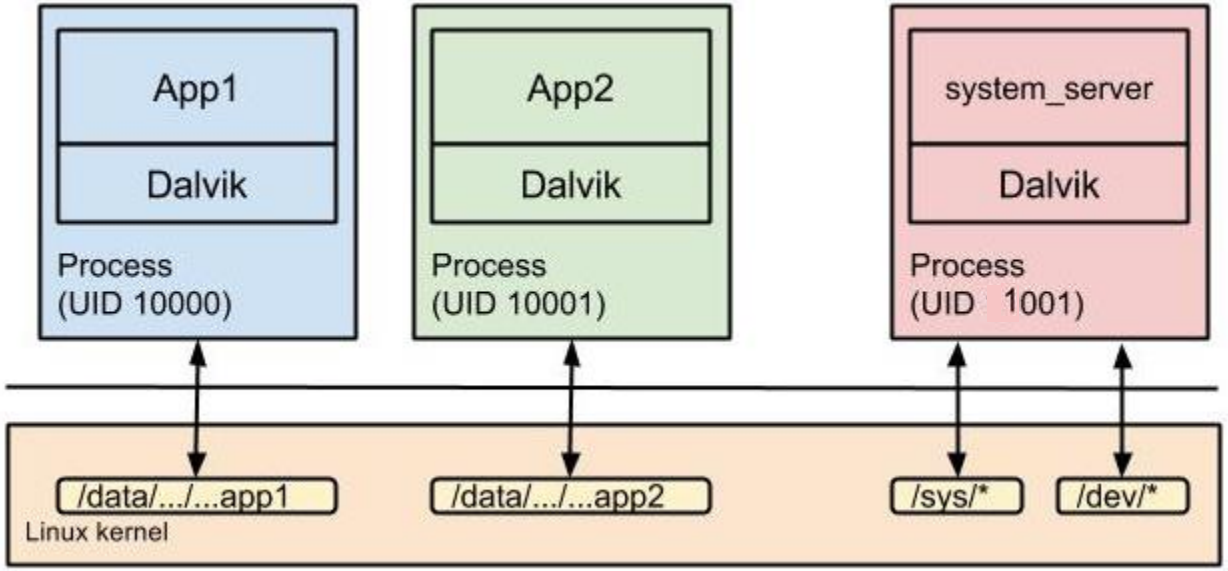
→ Kernel-level application sandbox

- 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 → needs same UID
  - Apps signed with same developer certificate
  - And explicitly sharing same UID in AndroidManifest.xml

```
1  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
2    package="com.android.nfc"
3    android:sharedUserId="android.uid.nfc">
```

SELinux

Security-Enhanced Linux

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

Source: source.android.com

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

Source: source.android.com

SELinux on Android – Sample Rules

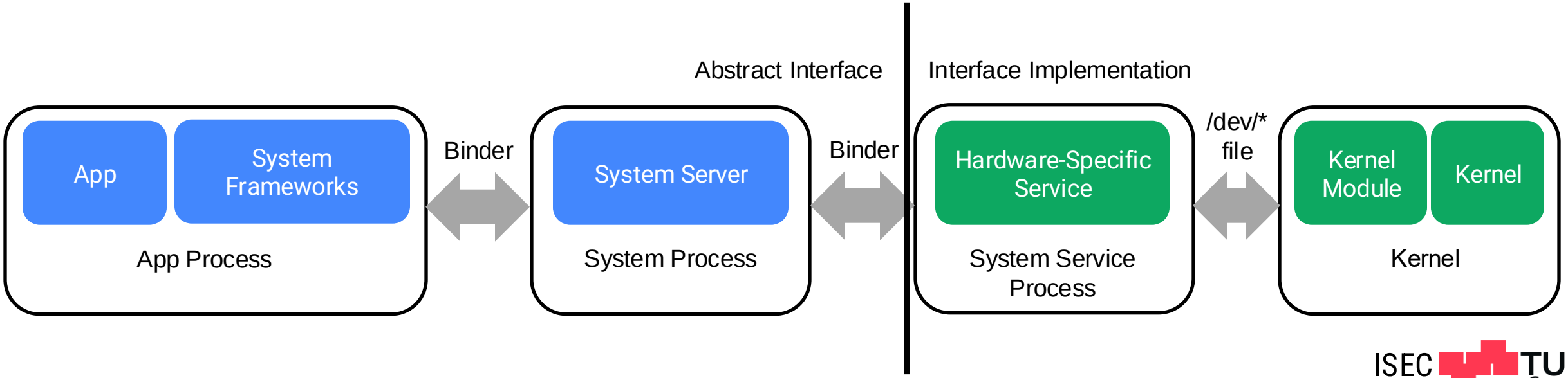
- No unlabeled files
- No ptrace
- No device node creation
- No raw I/O
- No mmap zero
- No mac_override
- No setting security properties
- No access to /data/security and /data/misc/keystore
- No /dev/mem or /dev/kmem access
- No /proc usermode helpers
- No ptrace of init
- No access to generically 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

Hardware Abstraction Layers (HAL)

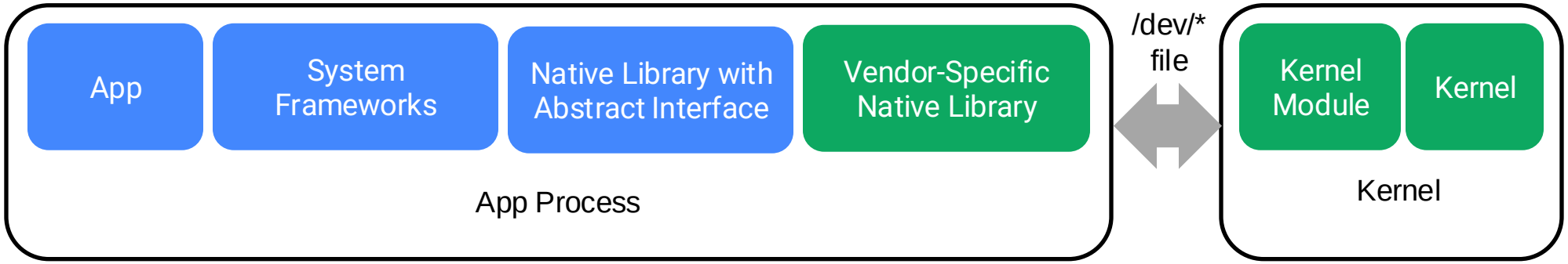
- Android runs on a multitude of different devices
  - Still: Share as much code between them as possible

- 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

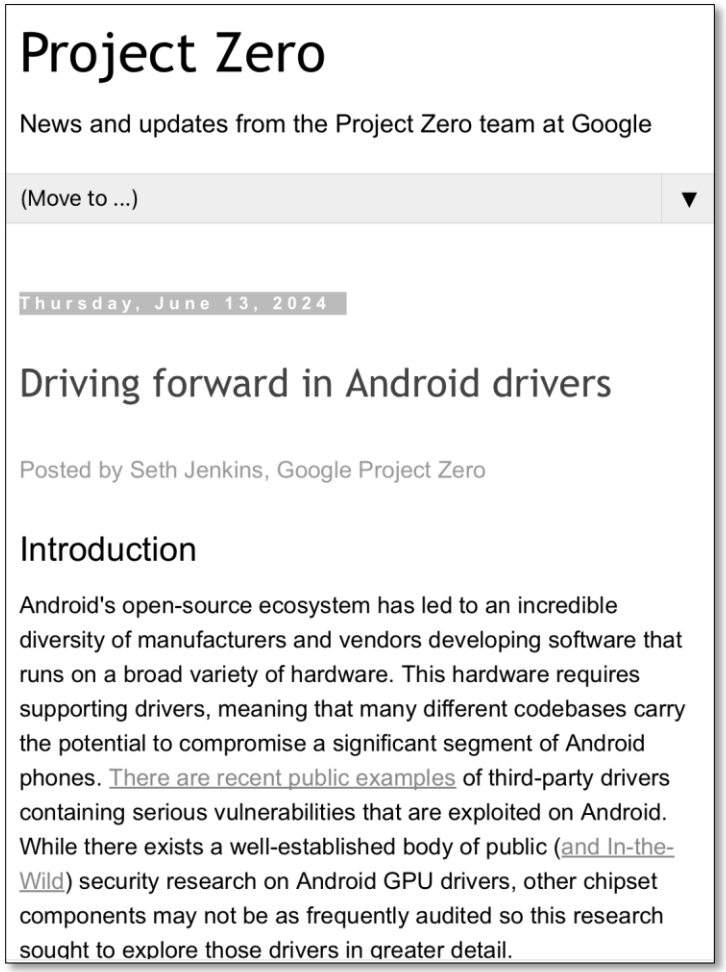


Hardware Abstraction Layers (HAL)

- 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

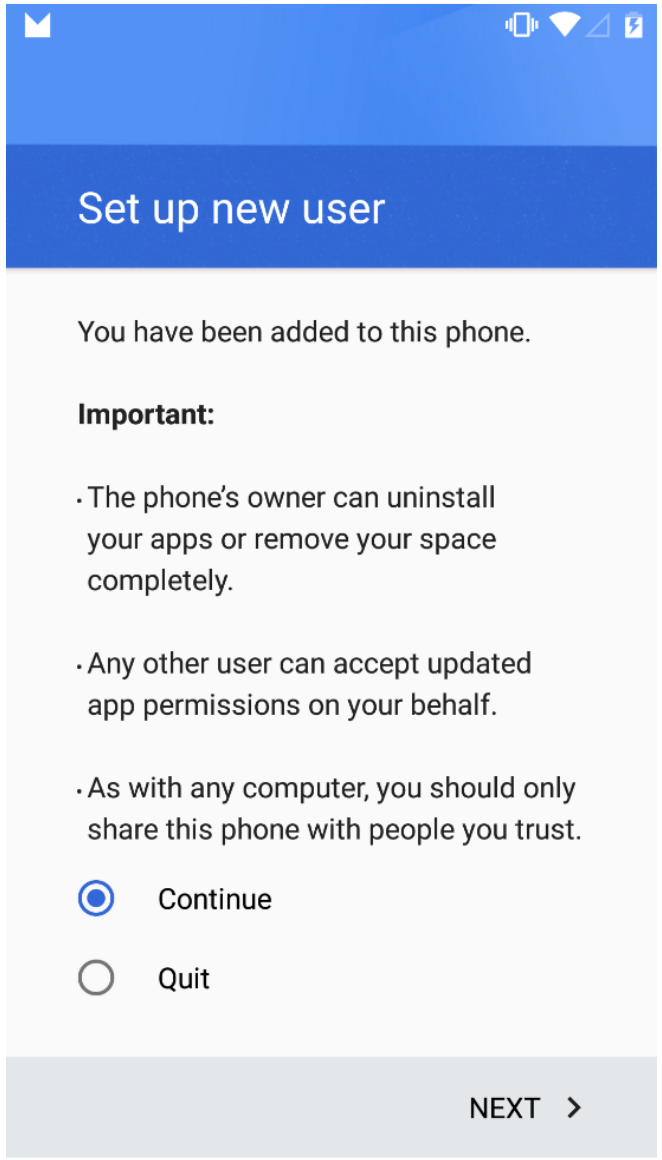
Android's open-source ecosystem has led to an incredible diversity of manufacturers and vendors developing software that runs on a broad variety of hardware. This hardware requires supporting drivers, meaning that many different codebases carry the potential to compromise a significant segment of Android 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: Additional Secure Element („StrongBox“)
  - Accessible to apps through Java Crypto APIs
  - Import keys, perform crypto operations without exposing key material
  - Strict separation between keys of different applications

- 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

Source: source.android.com

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

Source: source.android.com

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

Source: developer.android.com

Key Store: Fingerprint Authentication

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

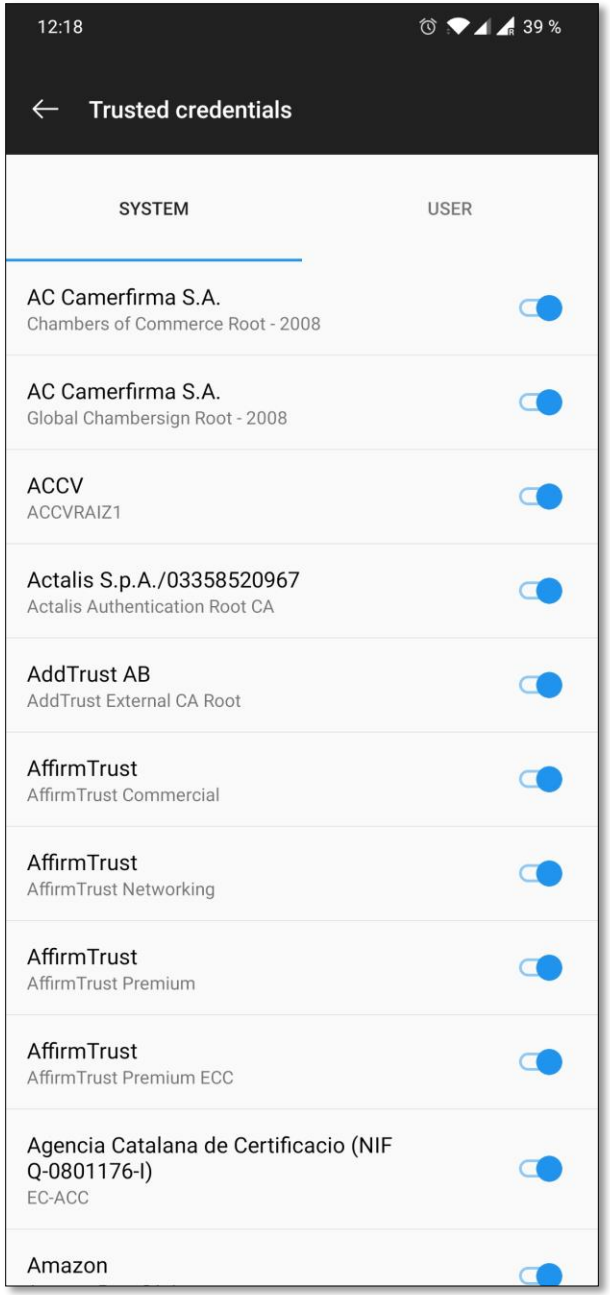
```
BiometricPrompt prompt = new BiometricPrompt.Builder(context).build();
prompt.authenticate(cryptoObject, executor, new BiometricPrompt.AuthenticationCallback() {
    @Override
    public void onAuthenticationSucceeded(BiometricPrompt.AuthenticationResult result) {
        // Authenticated!?
    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

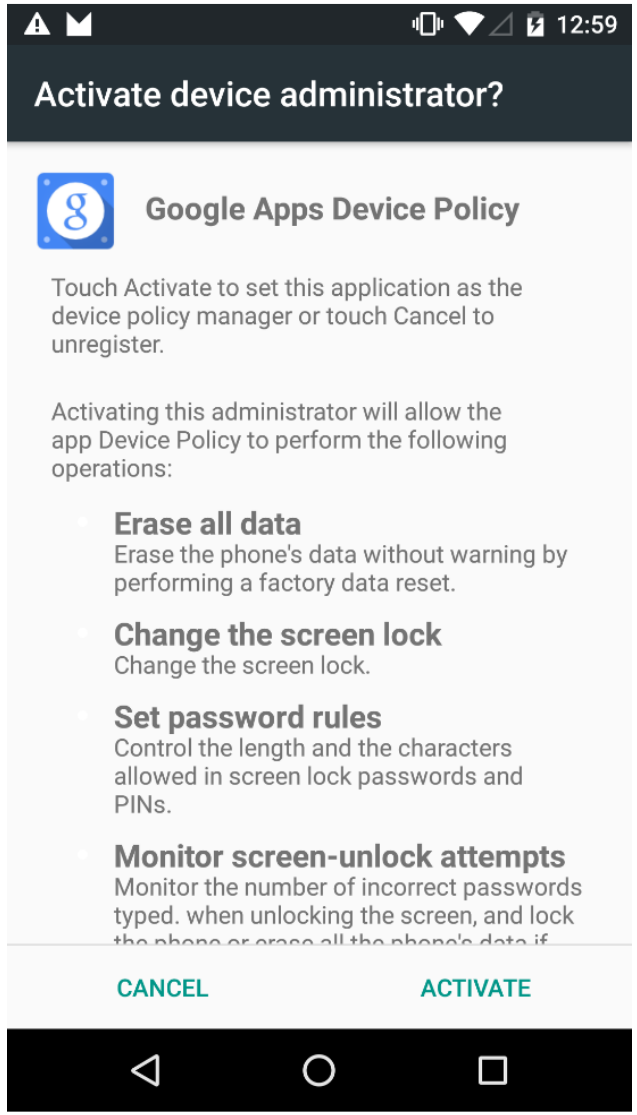
Source: Antonio Bianchi et al.: Broken Fingers, On the Usage of the Fingerprint API in Android

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 possible on legacy Android versions
    - SELinux

Systemless Root

- **Problem**: SELinux prevents any process from obtaining full root permissions
  - Even processes that run as root are restricted to a subset of capabilities
- **Problem**: Verified Boot requires the system partition to be read-only

- **Solution**: Start superuser daemon before SELinux is fully started
  - Set a custom init program that spawns SU daemon
  - Then hand over to Android's original init program

- This can be accomplished by just modifying the **boot partition**
  - System partition is untouched: OTA updates can still be installed
    - dm-verity hashes are unaffected
  - Example: Magisk

Outlook

- 21.03.2025
  - Application Security on Android I

- 04.04.2025
  - Application Security on Android II