Special Topics: Android Security

Univ.-Prof. Dr. René Mayrhofer
Institute of Networks and Security

Mobile devices

- This course mostly talks about smart phones, but other mobile device form factors exist!
  - Think about watches
  - Think about cars
  - … and anything in between
- “Smart” phones are about apps
  - Large ecosystem, only useful if desired apps exist
  - Complex balance between multiple stakeholders
    - **Users**
    - **App developers** and **content providers**
    - **Device manufacturers** (OEMs, ODMs, chipset manufacturers)
    - **Mobile network operators** (MNOs)
    - **Regulators** and **law enforcement**
  - A platform needs to be useful to users and provide incentives to developers/providers

Different ecosystem philosophies

- **Closed ecosystems** (e.g. Apple iOS)
  - Closed source
  - OS can only run on devices made by one (or very few) manufacturers
  - Apps can only be installed from a single store
  - Advantages:
    - Better coherence in product design
    - Easier to maintain security posture (e.g. updates from a single place)
  - Disadvantages:
    - Censorship (political, economic, or moral) trivial to implement, hard to circumvent
    - Product cost
    - Homogeneous produce landscape → a single exploit will work on many devices

- **Open ecosystems** (e.g. Android)
  - Core parts are open source (AOSP)
  - AOSP can run on arbitrary devices
  - Apps can be sideloaded
  - Advantages:
    - No single point of failure / censorship
    - Easy to analyze / adapt
    - Open for new experiments (devices, apps, and user changes to OS)
    - Diverse ecosystem makes single exploit chains harder
  - Disadvantages:
    - Ecosystem is harder to update quickly (many players need to adapt)
    - Differences in user interaction make switching products harder

This course is about Android (but some abstract parts apply to other platforms as well)

- Will mostly ignore other operating systems and platforms
- Will mostly talk about AOSP (Android Open Source Project), not much about Google specific services and components
- Will not talk about specific devices and their peculiarities
- Will only give a high-level overview, not a technical deep-dive into code – see e.g.:
  - https://android.com/security
  - https://source.android.com/security
  - https://developer.android.com/training/articles/security-tips
  - https://android-developers.googleblog.com/search/label/Security
- If not explicitly stated otherwise, will refer to Android 12 (released October 2021)
- **Required reading**: "The Android Platform Security Model" at https://arxiv.org/abs/1904.05572 or https://dl.acm.org/doi/10.1145/3448609
- Material in these slides inspired by many sources and many great people – thanks to all!

Mobile Security?
It’s not so different to arbitrary systems security...

A system is considered secure when the cost of successfully attacking it is higher than the potential gain.

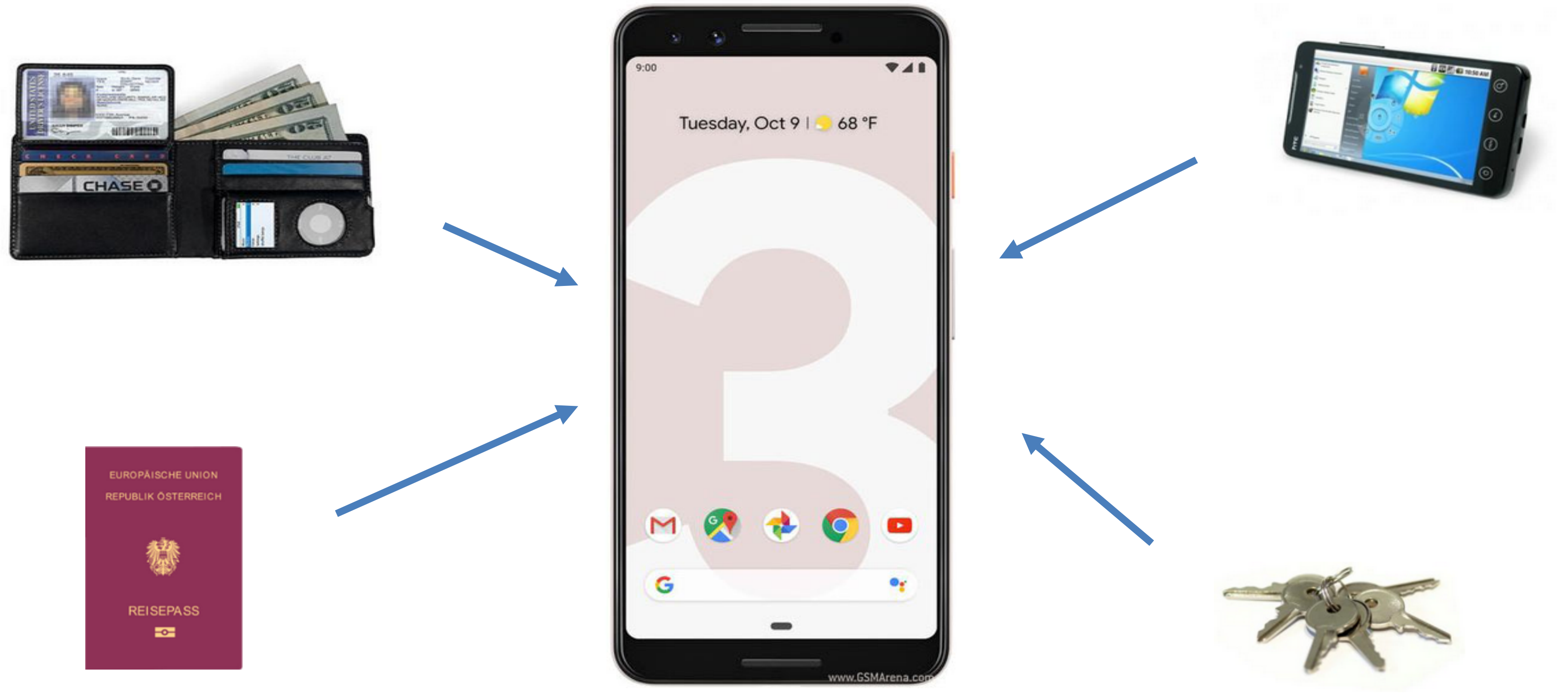
Remember that there is no perfect security.

Trends in mobile security

- Growing use of mobile devices
- Cloud-based applications (Skype, Dropbox, etc.)
- De-perimeterization (no firewalls)
- External devices on own networks (guests, partners, etc.)

→ **Many new applications are “mobile first” and often only usable with proprietary apps**

Convergence of security-critical services



Context: The Android ecosystem

… is massive, diverse, and constantly changing

- >1.300 brands
- >24.000 devices
- >1.000.000 apps
- >2.500.000.000 users

(https://www.blog.google/around-the-globe/google-europe/android-has-created-more-choice-not-less/)

Context: The Android ecosystem

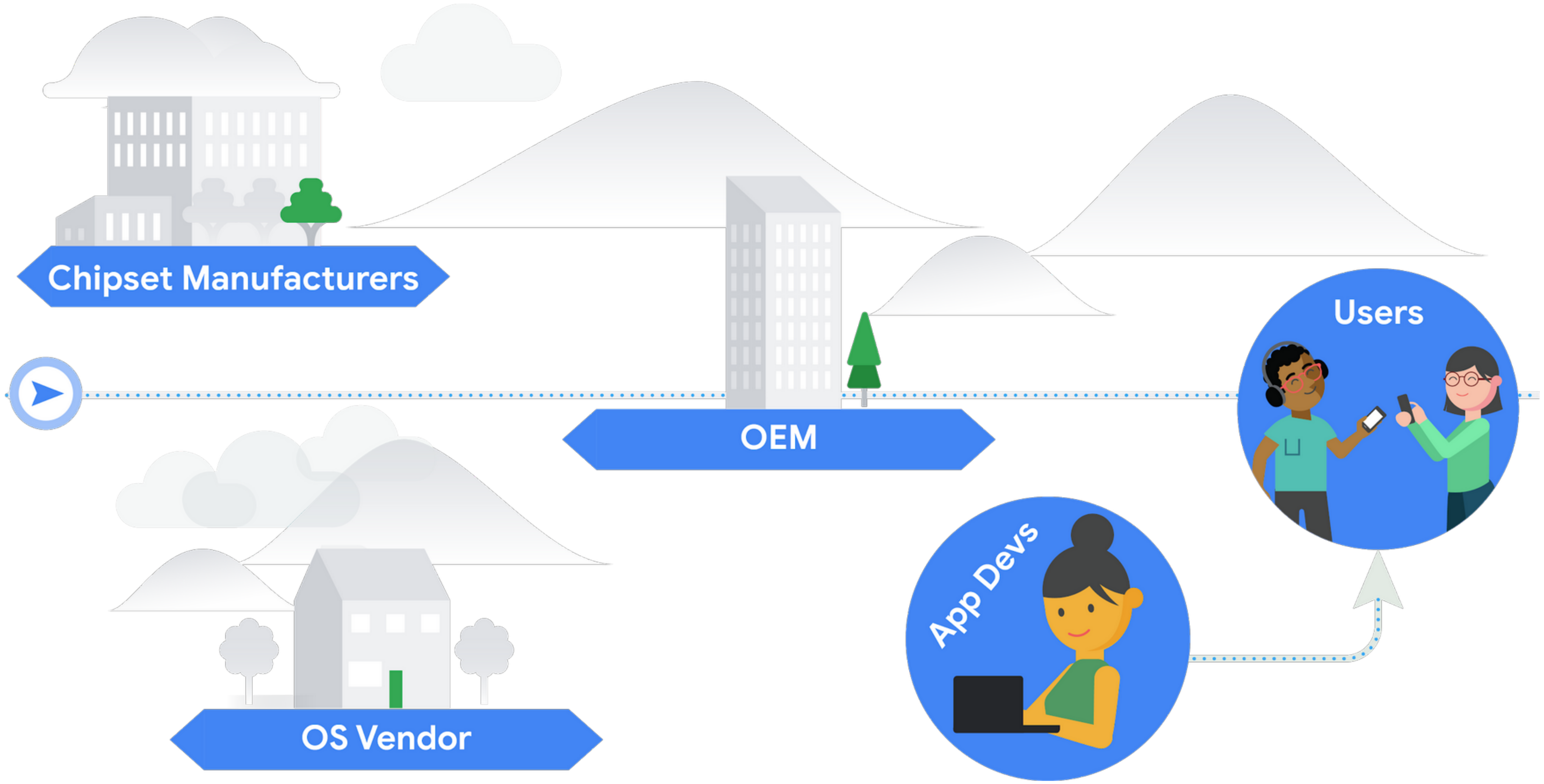


Image credit: Google

Context: The Android ecosystem

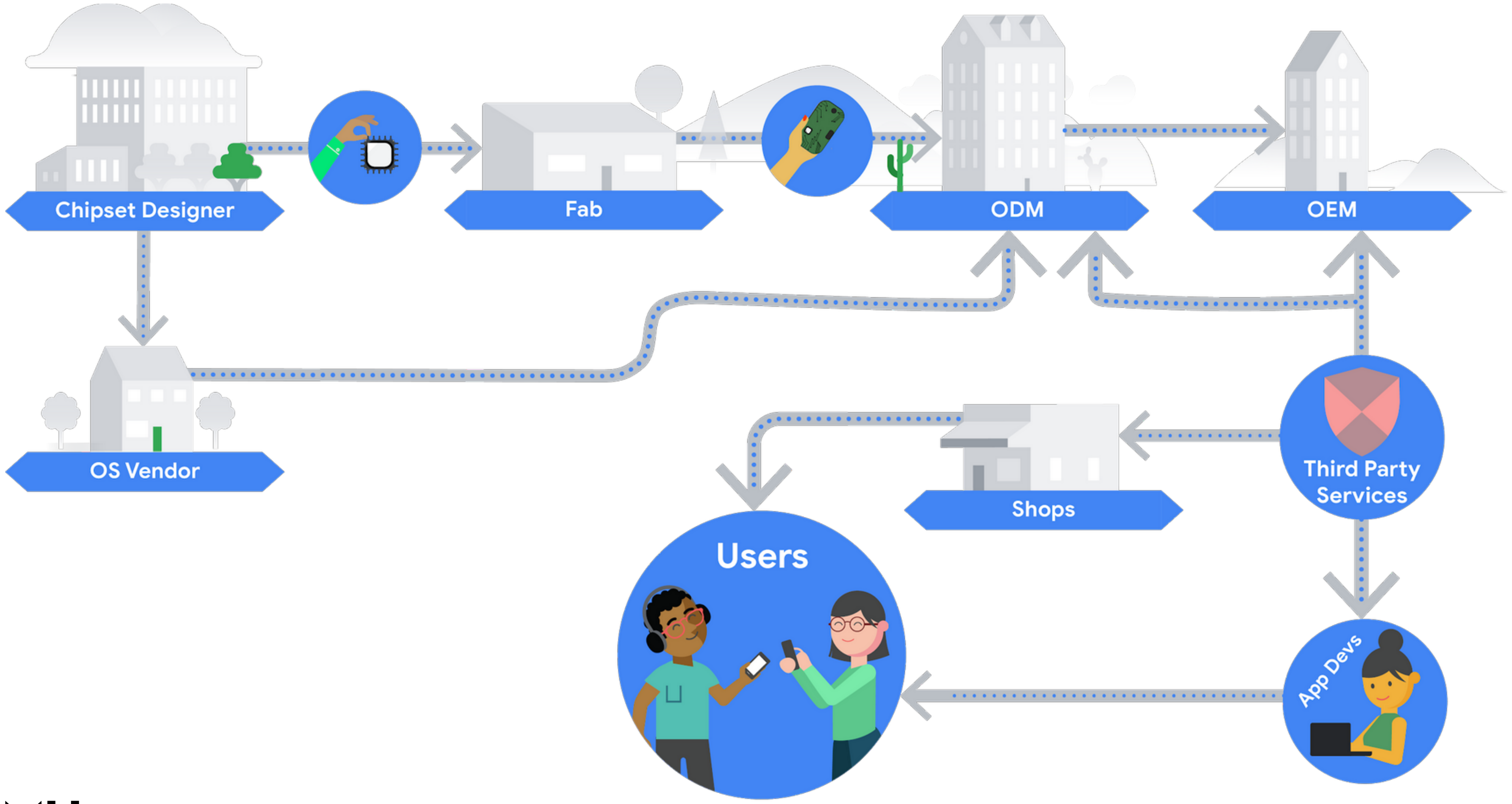


Image credit: Google

The Android Platform Security Model: Security Goals

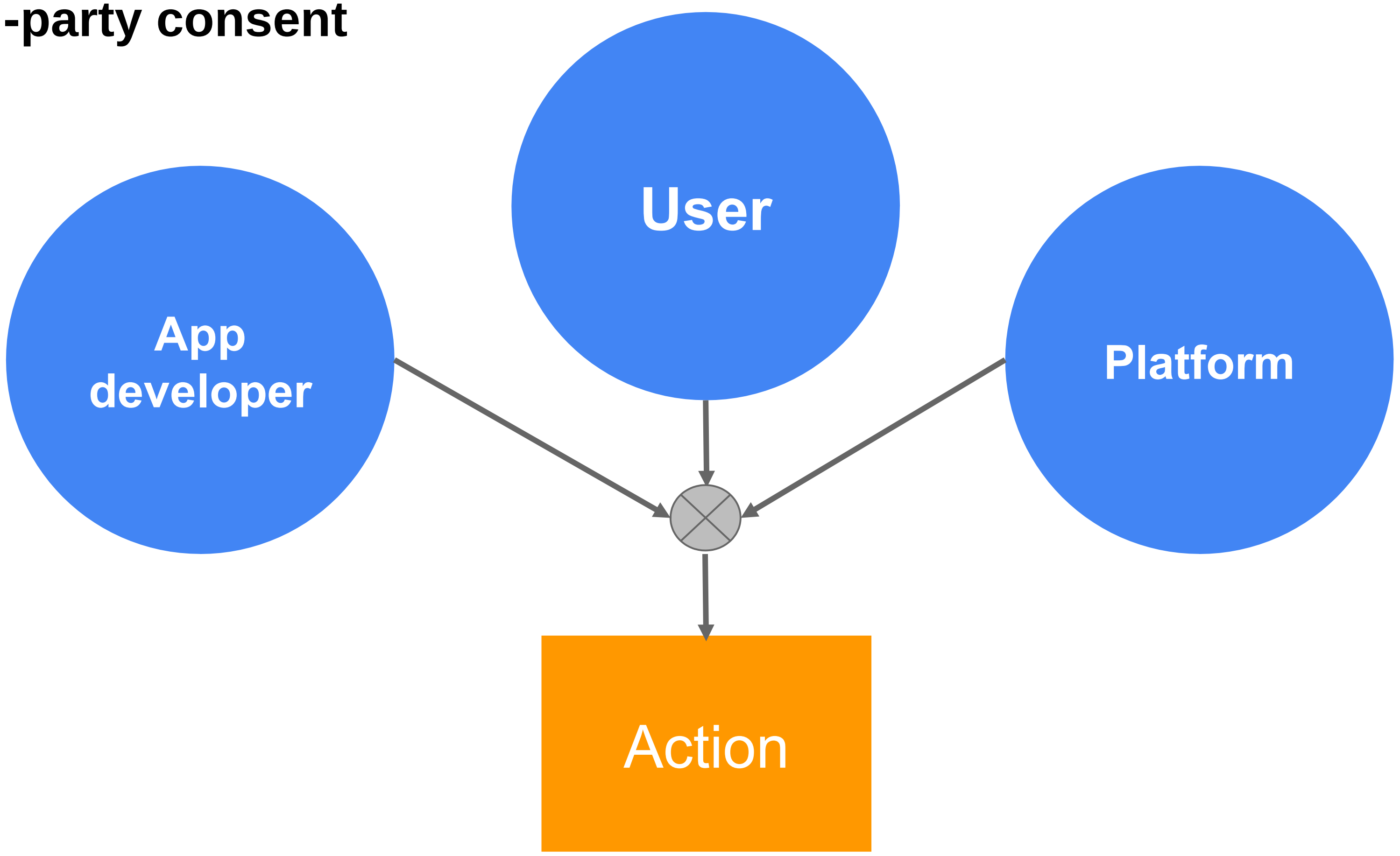
1) Protecting **user data**
   - ☐ Usual: device encryption, user authentication, memory/process isolation
   - ☐ Upcoming: personalized ML on device
2) Protecting **device integrity**
   - ☐ Usual: malicious modification of devices
   - ☐ Interesting question: against whom?
3) Protecting **developer data**
   - ☐ Content
   - ☐ IP

The Android Platform Security Model: Threat Model

- Hardware
  - Lost, stolen, lent/borrowed, etc.
  - Direct physical access must be assumed possible in different device states: *Powered off*, *screen locked*, *screen unlocked by different user*, *physical proximity*
- Operating System / Platform
  - Malicious apps due to **use of untrusted apps** → privacy threats for user, security threats for OS
  - Direct attacks over network interfaces (cell network, WLAN, Bluetooth, NFC) due to **use of untrusted networks**
  - **New**: **Insiders** can get access to signing keys
- Apps
  - Secure coding practices not used widely
  - Exploiting bugs in apps (or OS) due to **use of untrusted content**
  - Privacy violations by apps (e.g. location services) or unintentional **automatic synchronization of local content to untrusted cloud services**
- User
  - Authentication to mobile device is hard to make secure **and** usable
  - User needs to be able to trust smartphone → problem of **masquerading apps**

The Android Platform Security Model: Rules

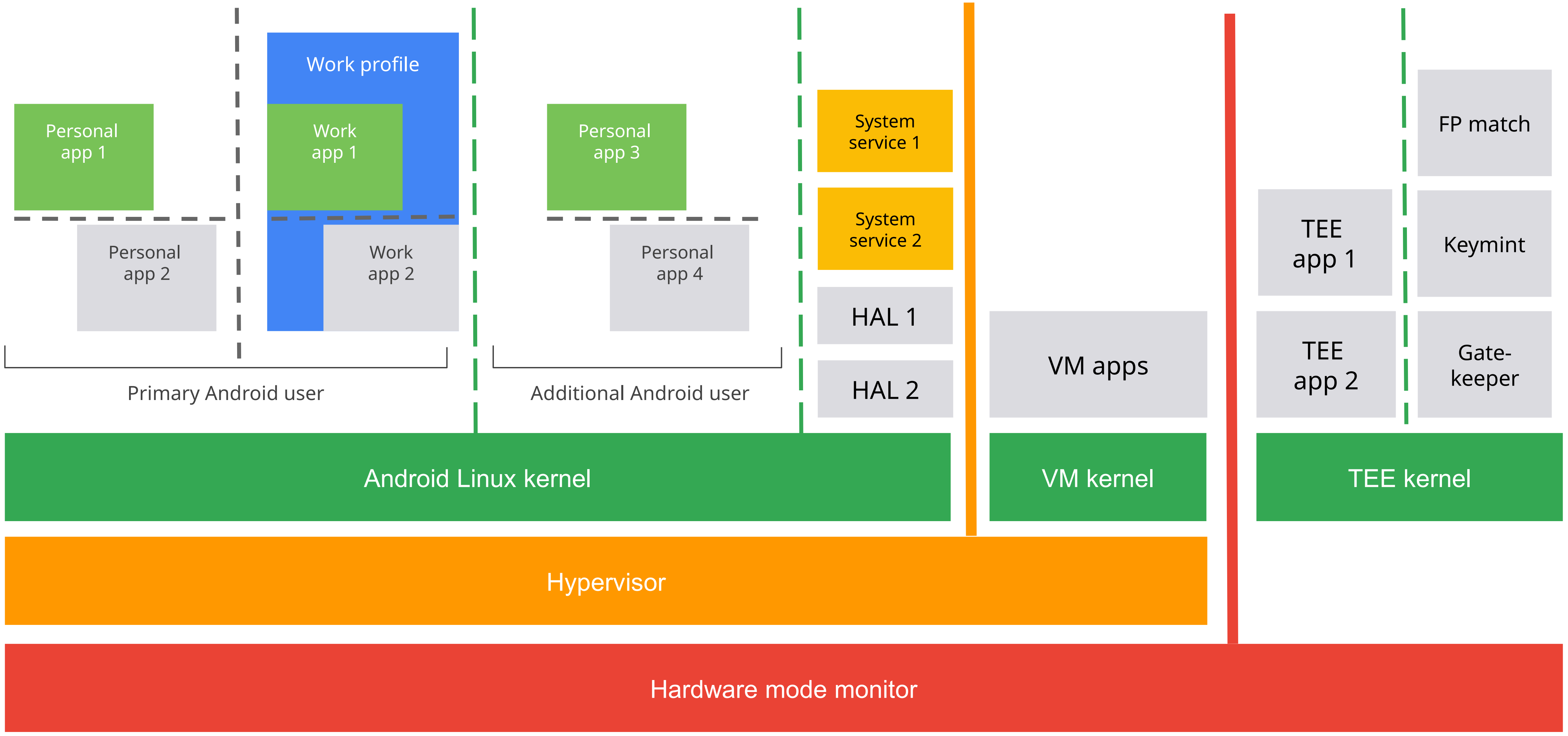
- Rule 1: **Multi-party consent**



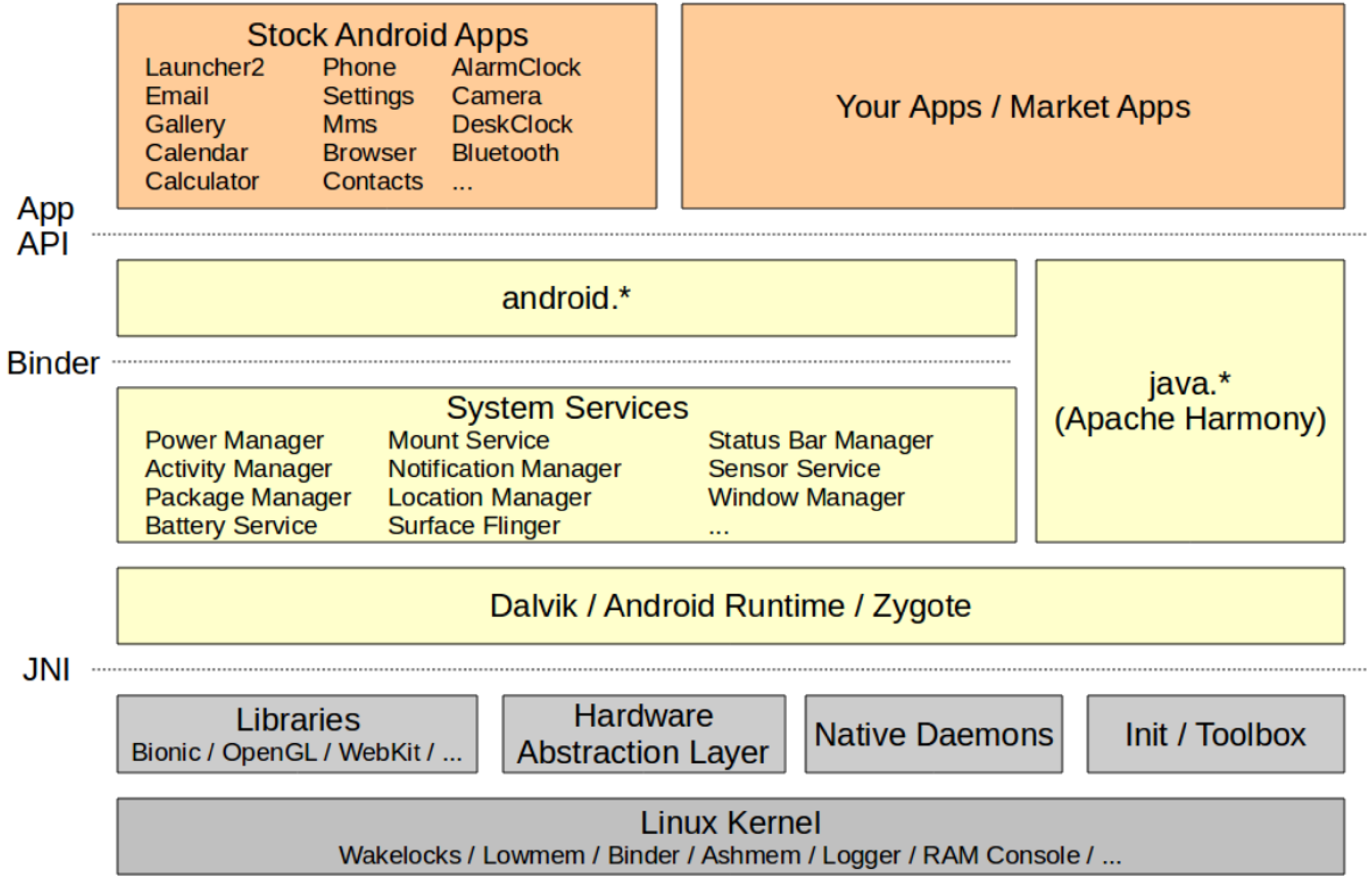
The Android Platform Security Model: Rules

- Rule 2: **Open ecosystem access**
- Rule 3: **Security is a compatibility requirement**
- Rule 4: **Factory reset restores the device to a safe state**
- Rule 5: **Applications are security principals**

Android architecture: layers of isolation (on main CPU)

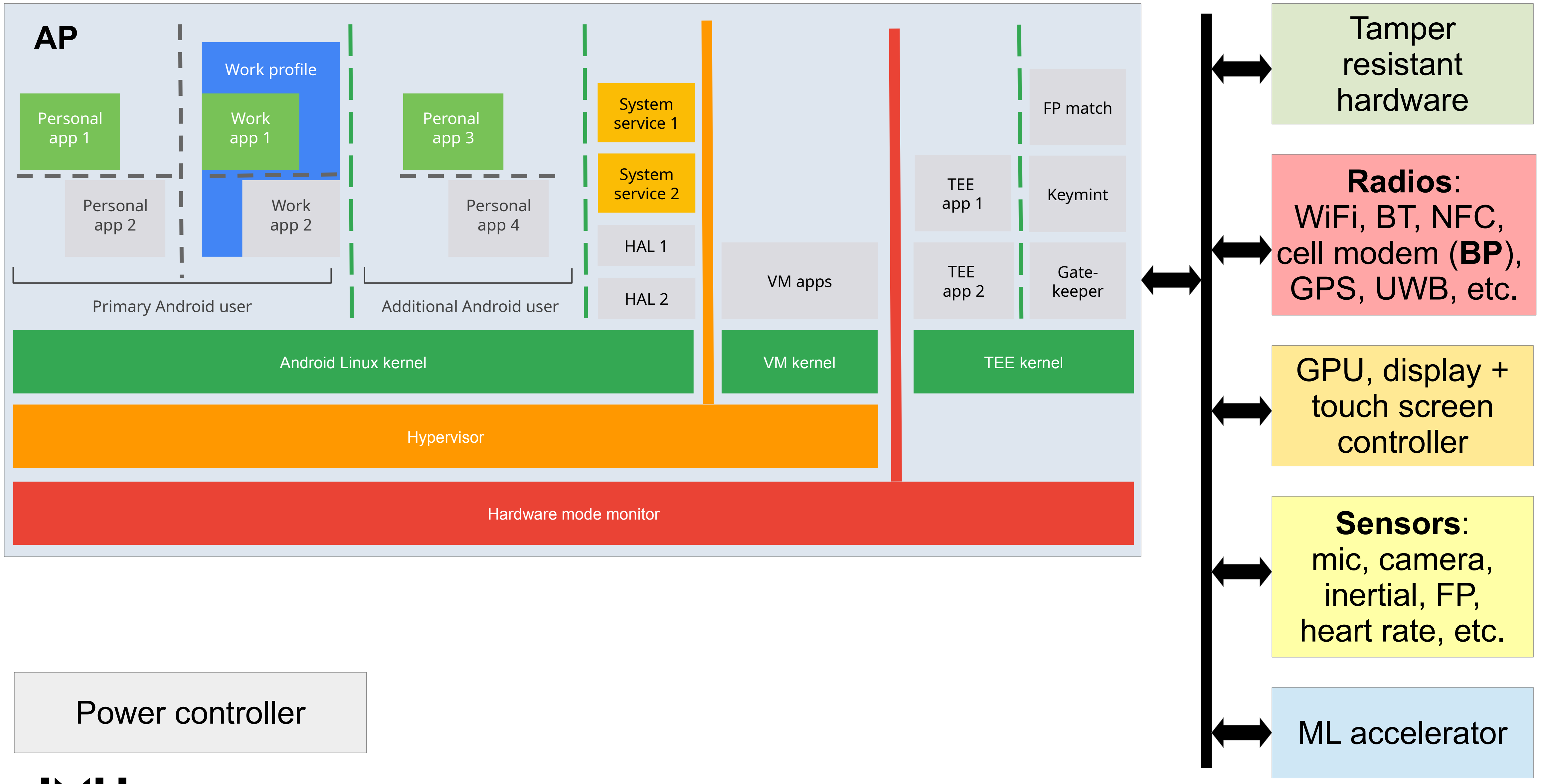


Android architecture: HLOS software layers



Source by Karim Yaghmour (Opersys Inc.) http://www.slideshare.net/opersys/inside-androids-ui (CC Share-Alike 3.0 license)

Android architecture: isolation between hardware modules



Power controller

JKU

Android security architecture

- **Applications must be signed for installation**
  - May be self-signed by the developer, therefore no requirement for centralized application Q/A or control
    - Note: Play-signed apps hold their private signing keys on the Google Play store
  - Signature supports non-repudiability (if the public key/certificate is known)
  - Signature by same private key allows applications to share data and files
  - Automatic application updates possible when signed by same private key

- **Otherwise, open eco-system**
  - Users may install arbitrary applications (directly from APK files or from different markets)
  - No specific rules for or control of applications in Google Android market
  - DRM and application copy protection introduced some time ago (Android 2.2 and newer market API), but optional

Android code signing

- **All Android apps (system and user-installed) must be signed**
  - typically, firmware updates are also signed by OEM, boot loader may only allow to flash and/or boot “correctly” signed images
  - recovery mode often applies only updates signed by same OEM
  - newer Android versions verify signatures during boot and run-time (*dm-verity*)
- Signing is done with private keys held by developers / organizations, public keys embedded in individual apps, system image, and/or in boot loader for image signatures
- Signing key types:
  - individual developer keys (self-signed) for apps
  - `platform`, `shared`, `media` and `testkey` in AOSP tree
    - `platform` is used for “core” Android components with elevated privileges (see later discussion of permissions)
  - `releasekey` for release type image builds, must by kept private
  - more details at https://source.android.com/devices/tech/ota/sign_builds.html and http://nelenkov.blogspot.co.at/2013/05/code-signing-in-androids-security-model.html

Android security architecture

Upon installation, package manager creates a dynamic user ID for each application
⇒ **Application sandbox**

- All application files and processes are restricted to this UID
- Enforced by Linux kernel and therefore same restrictions for all code (Java + native)
- Starting with Android 4.4 (introduced in 4.3 with `permissive` mode, 4.4 switches to `enforcing`), augmented with **SELinux** policy for kernel level mandatory access control (MAC)
- By default, even the user and debugging shells are restricted to a special UID (`SHELL`)
- Permissions granted at installation time allow to call services outside the application sandbox

“rooting” to gain “root” access (super user / system level access on UNIX without further restrictions, but may be limited by SELinux MAC)

Android security architecture

- Applications may install data on flash memory
  - `/data/data/<package name>` with sub-directories for files, databases, shared preferences (XML format, simple datatypes), etc.
  - API for centralized account data storage (one accounts database on system)
    - protected against normal applications
    - but no specific protection of this database on rooted devices (i.e. not encrypted)
  - All files in application directories only accessible by respective application UID (enforced with SELinux)
- Data on extended memory (typically MicroSD)
  - However, MicroSD cards typically formatted with VFAT (or exFAT)
    ⇒ DAC (Discretionary Access Control) by kernel only since Android 4.4 with filesystem/MAC policy tricks, before all files are accessible to all apps

Android filesystems and mount points

Partitions on Android devices (NAND flash)

- `(a)boot (loader)`: (first and) second stage boot loader code for early hardware initialization (including NAND flash) and selecting next partition to boot, highly specific to OEM/device
- **`boot`**: kernel+initramfs for normal boot, loaded by boot loader
  Pre-Android-9: initramfs is the root filesystem for the Linux kernel
  Post-Android-9: only kernel, `system` is mounted as root filesystem
- `rescue/`**`recovery`**: kernel+initramfs for rescue boot (typically for firmware update and reset to factory defaults, with custom rescue image also for backup/restore and flashing firmware modifications), loaded by boot loader
- **`system`**: main filesystem with Android user space binaries and libraries, mounted as `/system`, partially symlinked from root (e.g. `/etc`), starting with Android 9 *System-as-root* layout mounted directly as `/`
  ⇒ “the firmware”, “ROM”, etc.
- **`vendor`** and **`product`** (Android >=9): additions to `system` that are specific to chipset vendor, OEM, or Google
- **`userdata`**: user data stored on the device itself, mounted as `/data`
- Others: `efs` (on some devices for hardware identifiers like IMEI, Bluetooth/WLAN MAC), `radio` (baseband firmware image), `cache` (various cached data, e.g. Dalvik cache or OTA update packages, mounted as `/cache`), `splash` (on some devices for first image on bootup), …

Most of these partitions are formatted as ext4 (newer devices) or e.g. YAFFS2 (on very old devices)

On-device encryption

- Android 5.0 introduced **Full Disk Encryption** (FDE) – removed from AOSP in Nov. 2021
  - entangled with user knowledge factor (PIN/password), but can potentially be disabled (then encryption key only depends on device-unique key kept in TrustZone)
  - full data partition encrypted with same key, including meta data (e.g. file names)
  - all user accounts and profiles encrypted with same key
  - most system functions inaccessible until knowledge factor entered during reboot
- Android 7.0 introduced **File Based Encryption** (FBE)
  - different keys per users/profiles
  - difference between “device encrypted” (DE, only bound to unique device key) and “credential encrypted” (CE, entangled with user knowledge factor)
  - apps that are marked to use DE data storage can function after reboot before first unlock
  - Android 9 added meta data encryption
  - **Android 10 made FBE mandatory for all new devices**
  - Android 11 introduced Resume-on-Reboot, extended with Android 12 to support server time-based retrieval: https://source.android.com/devices/tech/ota/resume-on-reboot

Android security boundaries

Android sandbox has **two main layers of permissions models**

- **File system entries and some other kernel resources**
  - enforced by DAC (standard filesystem permissions) and in newer versions MAC (SELinux) ⇒ enforced on kernel level
  - very restrictive compared to standard Linux distributions
  - Android ID (AID) is used as both UID (user ID, for installed applications) and GID (group ID, for accessing resources)
  - commonly referred to with the term “Android sandbox” (although this is not the full picture)
- **Permissions on API calls**
  - enforced by DalvikVM/ART and Android framework/libraries, as well as specific apps
  - allow bridging the security boundary created by the first layer enforced by kernel sandbox
- Plus other mechanisms for specific purpose (e.g. Linux capabilities and `seccomp` filters)

For interplay between DAC, MAC, and CAP see e.g. [Hernandez et al.: “*BigMAC: Fine-Grained Policy Analysis of Android Firmware*”, USENIX Security 2020], online at https://www.usenix.org/conference/usenixsecurity20/presentation/hernandez

Android IDs

- AIDs represent both UIDs and GIDs, but not all make sense as UID or GID
- e.g. `AID_SDCARD_RW` maps directly to `sdcard_rw` group that is used as supplemental group for accessing external storage mount points (which are mounted with this GID)
- e.g. `AID_INET` group membership is used in one of the Android Linux kernel patches to allow opening `AF_INET` and `AF_INET6` sockets (sometimes called “Paranoid Networking” patch)
  → by default (without the `INTERNET` permissions), apps cannot connect to the network
- e.g. `AID_INET_ADMIN` grants kernel-level `CAP_NET_ADMIN` capability to configure network interfaces and routing tables

All processes (system or app) run as a specific AID

⇒ memory and startup time are minimized by pre-initialized `zygote` process from which all other Java processes are forked, and first task of `zygote` after fork (before running newly loaded app code) is to drop privileges to those representing the AID

Crossing the app sandbox (process) boundary

- Apps invoke Android APIs as libraries linked in their own process (with the app AID)
- Privileged processes (services) run in different process (other, more privileged AID)
- Crossing the boundary required IPC (Inter Process Communication)
- On Android, implemented by **Binder**
  - Patch to Linux kernel, part of the Android Common Kernel
  - Can be called from unprivileged processes
  - Calls registered objects in other processes
  - Transports objects (shared memory) from one process to another
  - Object-oriented call and arguments interface defined by AIDL (Android Interface Definition Language) ⇒ Details see https://developer.android.com/guide/components/aidl
- **One of the core security components in AOSP** ⇒ bugs in Binder often lead to universal Android exploits

Android IDs: Examples

Defined in `system/core/include/private/android_filesystem_config.h` in AOSP

```
#define AID_ROOT             0  /* traditio:nal unix root user */
#define AID_SYSTEM        1000  /* system server */
#define AID_RADIO         1001  /* telephony subsystem, RIL */
#define AID_BLUETOOTH     1002  /* bluetooth subsystem */
…
#define AID_AUDIO         1005  /* audio devices */
#define AID_CAMERA        1006  /* camera devices */
#define AID_LOG           1007  /* log devices */
#define AID_COMPASS       1008  /* compass device */
…
#define AID_SDCARD_RW     1015  /* external storage write access */
…
#define AID_MEDIA_RW      1023  /* internal media storage write access */
…
#define AID_NFC           1027  /* nfc subsystem */
#define AID_SDCARD_R      1028  /* external storage read access */
#define AID_SHELL         2000  /* adb and debug shell user */
#define AID_CACHE         2001  /* cache access */
#define AID_DIAG          2002  /* access to diagnostic resources */
…
/* The 3000 series are intended for use as supplemental group id's only.
 * They indicate special Android capabilities that the kernel is aware of. */
#define AID_NET_BT_ADMIN  3001  /* bluetooth: create any socket */
#define AID_NET_BT        3002  /* bluetooth: create sco, rfcomm or l2cap sockets */
#define AID_INET          3003  /* can create AF_INET and AF_INET6 sockets */
#define AID_NET_RAW       3004  /* can create raw INET sockets */
#define AID_NET_ADMIN     3005  /* can configure interfaces and routing tables. */
…
#define AID_APP          10000  /* first app user */
#define AID_ISOLATED_START 99000 /* start of uids for fully isolated sandboxed processes */
#define AID_ISOLATED_END   99999 /* end of uids for fully isolated sandboxed processes */
#define AID_USER        100000  /* offset for uid ranges for each user */
#define AID_SHARED_GID_START 50000 /* start of gids for apps in each user to share */
#define AID_SHARED_GID_END   59999 /* start of gids for apps in each user to share */
```

Leaving the sandbox: Requesting permissions

- For example, an application that needs to monitor incoming SMS messages would specify:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.me.app.myapp" >
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    …
</manifest>
```

- All requested permissions need to be granted at application installation (or otherwise, the application will not be installed) by explicit user consent
- Permissions stored by **package manager service** in `/data/system/packages.xml` are world readable, writable by user `system` (and `root`, obviously)
- Permissions checked (partially) at run-time by the respective services being called, some permissions are mapped to AIDs that apps receive as supplementary groups
  - "low-level" mapping in `/etc/permissions/platform.xml`
  - e.g. `android.permissions.INTERNET` maps to `AID_INET` as supplementary group given to respective processes

Permissions: Protection levels

Permissions are associated to **protection level**

- **normal**: granted by default, user is not asked (and cannot revoke selectively)
  e.g. `ACCESS_NETWORK_STATE` or `GET_ACCOUNTS` but also `INTERNET` and many others
- **dangerous**: user is asked (at installation time for Android <6 and at runtime for Android >=6 with targetSdkVersion>=23) and needs to grant all such (grouped) permissions
  e.g. `READ_SMS` or `CAMERA`
- **signature**: only granted to applications that have been signed by the same key as the app that declared this permission
  e.g. `NET_ADMIN` or `ACCESS_ALL_EXTERNAL_STORAGE`

  Note: for built-in permissions (declared by `/system/framework/framework-res.apk`) with signature protection level, requesting app needs to be signed with the same key that signed the firmware (specifically the `platform` key).
- **signatureOrSystem**: granted to applications that are either part of the system image (until Android 4.3 located under `/system/app`, since Android 4.4 under `/system/priv-app`) or signed with the same key as the app that declared the permission

Permissions: Problems with usability

How many users read (and understand) the list of permissions?

- Cyanogenmod >=7.1 and >=10 allows to selectively disable permissions for applications and fake data instead
- Xprivacy module for Xposed framework supports this on all rooted devices (you can try e.g. with Magisk)
- Android >=6 (Marshmallow) supports runtime permission prompts
  - Huge improvement for security/usability trade-off
    - Permission lists no longer all-or-nothing
    - Runtime prompts give context why app is asking this permission (group)
  - Difficult to add more permission groups ⇒ balance between over-prompting and selective control
- Details: https://developer.android.com/guide/topics/permissions/overview

Leaving the sandbox: Demanding permissions

- An application that wants to control who can start one of its activities could declare a permission for this operation:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.me.app.myapp" >
    <permission android:name="com.myapp.permission.DEADLY_ACTIVITY"
        android:label="@string/permlab_deadlyActivity"
        android:description="@string/permdesc_deadlyActivity"
        android:permissionGroup="android.permission-group.COST_MONEY"
        android:protectionLevel="dangerous" />
</manifest>
```

  ⇒ Allows user to decide if another application might call an activity.

- Declare activity as not being exported at all (can be launched only by components of the same application or applications with the same user ID)

```
<activity android:exported=”false” ...>
```

- Arbitrarily fine-grained permissions can be enforced with the `Context.checkCallingPermission()` method

Leaving the sandbox: Permission verification

Multiple points of permission checks in Android

- Permissions map to group (GID) membership from kernel point of view, kernel implements permission verification directly (see before, e.g. `INTERNET` permission mapping to group `AID_INET`)
- Android framework and services verify that caller (e.g. through direct API call or through Binder) holds the permission
- Apps verify permissions of caller explicitly

Leaving the sandbox: Permission verification

Android framework has some standard entry points that verify declarative permissions

- **Activity permissions** (applied to the `<activity>` tag) restrict who can start the activity
- **Service permissions** (applied to the `<service>` tag) restrict who can start/bind to the service
- **BroadcastReceiver permissions** (applied to the `<receiver>` tag) restrict who can send broadcasts to the associated receiver.
- **ContentProvider permissions** (applied to the `<provider>` tag) restrict who can access the data in a `ContentProvider`. Unlike the other components, there are two separate permission attributes you can set: `android:readPermission` restricts who can read from the provider, and `android:writePermission` restricts who can write to it.

⇒ **if the caller does not have the required permission then `SecurityException` is thrown from the call** (some exceptions, e.g. `BroadcastReceivers` without the required permission will just not receive the broadcast)

Special case: sharedUsedId

- Multiple apps can request to run with the same UID

```
<manifest xmlns:android="..."
  package="com.me.app.myapp-1"
  android:sharedUserId="com.me.app.shareduser">
```

- Package installer only allows apps to share UID if they are signed by the same key
- As multiple processes/apps will have the same UID (AID), no protections between them from kernel or framework point of view
  - Can access each other’s private data directories
  - Can access each other’s internal components

Android Debug Bridge (adb)

- Components
  - `adbd` running on Android devices (when “USB Debugging” option enabled), available over USB, WiFi, or locally on devices on ports 5555-5585
  - `adbd` running on development workstation (part of Android SDK) on port 5037
  - `adb` command-line tool acting as client to `adbd` on development workstation
- Using adb
  - Supports debugging, file transfer, package installation, reboot control, etc.
  - Normally runs as user `SHELL` (uid 2000)
  - Can be restarted as user `root` (uid 0)
    - with global property `ro.secure=0`, then can call `adb root` to restart
    - or use one of the known exploits to force adbd to retain root privileges
    - will then support e.g. `adb remount` to remount `/system` with read-write option (if `dm-verity` is disabled) and `adb shell` interactive access will run as `root`

ADB authentication

- Starting with Android 4.2.2, `adbd` on the Android device requires authentication by the host with public/private key
  - popup on connection to the device, can remember public key of host for future use without seeing the popup again
- Before that, no authentication was required, and ADB protocol was available via USB when “debugging” was enable on device developer settings
- Without authentication, any USB host device can potentially attack the Android device, e.g. masquerading as chargers (e.g. “Juice Jacking” attack by Robert Rowley) or directly from another Android device in USB OTG mode (e.g. “physical drive-by” attack by Kyle Osborn)
- Potential attack surface is large due to ADB feature set

ADB debugging features

- DalvikVM supports live debugging of running apps
- to attach debugger (e.g. from within Eclipse / Android Studio) to running app, it needs the `android:debuggable` flag set to `true` (typically by building with debug target)
- on `user` type build of the whole Android tree and for in-production apps, flag is normally not set
- on `eng` type builds for the Android image:
  - `ro.secure` property is set to 0
  - `ro.debuggable` property is set to 1

  allows debugging of all apps independently of their own flag!
- on rooted devices, can change read-only properties with `setpropex` tool
  - need to restart `system_server` process after changing properties (device will appear to reboot, but only Android framework is reloaded and kernel is not influenced)
  - Pau Oliva's RootAdb app can automate this

Android framework

Android framework consists mostly of “framework managers” on top of DalvikVM/ART that run within `system_server` process

- Activity manager: intent resolution, app/activity launch, registration, etc.
- View system: views (UI compositions) in activities
- Package manager: information and actions about installed packages
- Telephony manager: telephony services, radio states, network information
- Resource manager: provides access to non-code app resources (e.g. graphics, UI layouts, strings, etc.)
- Location manager: provides access to GPS, cell/WiFi information, etc.
- Notification manager: event notifications including sounds, LEDs, status bar, etc.

All these run as `system` user, display e.g. with

```
ps -T -p <pid of system_server>
```

Volume daemon

Special system daemon: `vold`

- implemented natively (not in Java, does not run in `system_server` but as native Linux daemon)
- responsible for mounting and unmounting storage devices, e.g. SD cards or USB devices
- also handles full disk encryption for `userdata` and other partitions ⇒ has interfaces for password handling
- supports OBBs (opaque binary blobs): files shipped with apps, encrypted with shared secret, mounted for transparent use during app run-time
- Note: `vold` runs as user `root` (necessary for mounting/unmounting file systems), so any bugs are critical

User authentication

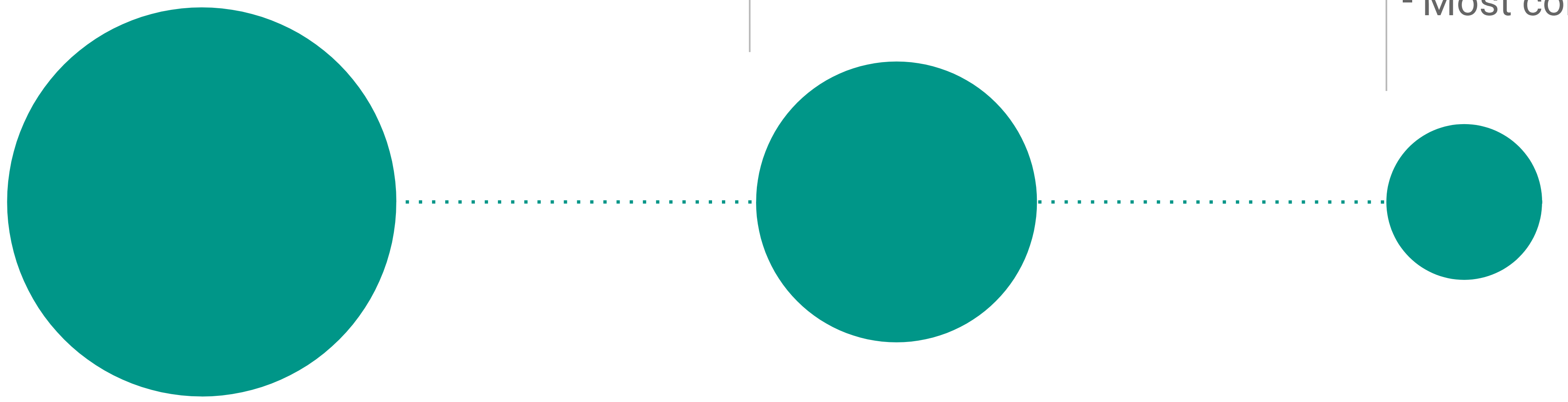
- On most mobile devices, the “lock screen” is the primary method of authentication
- (Mostly) binary distinction: locked or unlocked
  - some nuance with notifications and other information on lock screen
  - some functions can be used on locked phones (e.g. camera or emergency call)
- Can integrate with key management (KeyMaster/KeyMint / StrongBox)
- But implemented by Android user space ⇒ cannot defend against root adversaries

Tiered authentication model

Primary Auth
- Knowledge-factor based
- Most secure

Secondary Auth
- Needs primary auth
- Less secure
- Somewhat constrained

Tertiary auth
- Needs primary auth
- Least secure
- Most constrained



Android rooting: Why?

- Development
  - Installing custom root, recovery, and system images (note: different from rooting the running system!)
  - Debugging the Android framework and system daemons
  - [on earlier devices without dm-verity] changing files on the `/system` partition
- General
  - Reading all files on `/system` and `/data` partitions (including centralized system databases, e.g. accounts)
  - Changing kernel settings, e.g. CPU over-/underclocking, IPv6 address privacy
  - Custom routing, WiFi/3G connection sharing on devices that do not support “tethering” out of the box or have it disabled by the manufacturer/carrier, (re-)activating SIP, etc.
  - Installing/using applications that require root access, e.g. backup/restore, Webkey, SSH daemon, etc.

Rooting overview

1) **Achieve temporary root privileges**
   - ☐ on “developer” devices (Google Pixel, Nexus series) simply with `fastboot oem unlock`
   - ☐ booting into recovery mode on devices that allow this without restriction
   - ☐ exploiting an implementation bug
     - either during normal system run-time
     - or in the boot loader code to get into recovery mode (see above), e.g. flawed signature checking allows booting custom recovery image and from within that running image, flashing a new system image
2) **Achieve permanent root privileges**
   - ☐ by flashing a new `system` image with pre-installed binaries for root access
   - ☐ by installing a `/system/(x)bin/su` binary with “setuid” Bit set and the “SuperUser” application that will ask the user on each (first-time) access
   - ☐ by modifying the root image, e.g. to set the global property `ro.secure=0`
   - ☐ when SELinux is in enforcing mode, then need to modify policy as well
3) **Secure system against abuse by other applications**

Details: Linux kernel

- Android is based on Linux kernel
  - “root” is the super-user, equivalent to system-level access
  - Kernel implements DAC (Discretionary Access Control) on filesystem (includes kernel virtual files, e.g. /proc, /sys, etc.)
  - Optional MAC (Mandatory Access Control) schemes available in upstream kernel
    - SELinux (NSA, flexible, comprehensive, but complex)
      – used since Android 4.3, in strict (enforcing) mode since Android 4.4, now main enforcing tool
    - SMACK (simpler, path-based, used on MeeGo)
    - TOMOYO (path-based, more flexible than AppArmor)
    - AppArmor (simpler, path-based, used on Novell and Ubuntu servers)

Details: Standard UNIX filesystem ACLs

- *read*, *write*, *execute* bits for *owning user*, *group*, and *all others*
- Additional bits available, e.g. *setuid* and *setgid*
  - binary called with privileges of owner, not privileges of caller
  - typical combination is a file owned by root with setuid bit set
- setuid binaries used on many UNIX/Linux systems to allow normal users to perform administrative tasks (e.g. `passwd`) or for arbitrary code elevation (e.g. `su`, `sudo`)
- For controlled root access, simple and effective method on Android:
  - no changes to any existing system binaries
  - only need one additional binary installed with setuid bit set
  - typically `/system/(x)bin/su` as on standard UNIX systems, but with Android-specific GUI to ask user for permission on root access

Android logging on running (live) Android system

- Kernel log similar to other Linux systems, typically low-level and hardware information
  - `dmesg`
- Android applications log, configurable in verbosity of messages (log level)
  - `logcat`
  - `logcat –b radio` for GSM logs
- Information on running services, used accounts, memory use, IPC (intents, broadcasts), process information, etc.
  - `dumpsys`
  - Note: date/time format is in milliseconds since 1970-01-01, divide by 1000 and convert with date –d @<timestamp>
- Combination of above information: `bugreport`

Data at rest on Android devices: nonvolatile memory

- Generally stored on flash: NAND, eMMC or (Micro-)SD
- Android system and applications store a variety of relevant data items:
  - SMS/MMS
  - Call logs
  - Voice mail
  - Email
  - Web history, form data, cookies, web cache, etc.
  - Google search history
  - List of YouTube videos that were seen
  - Pictures, videos, audio recordings, etc.
  - Geo-location information
  - Account information and sometimes plain text passwords
  - etc.

Standard directory and file structure

- `/cache`: backup, firmware updates, …
- `/data`: partition for user data, generally the most important forensic data source
- `/data/app`: installed apps
- `/data/app-private`: installed apps that use copy protection
- `/data/backup`: recent directory for cloud backup API
- `/data/data`: app data
- `/data/{user|user_de}/<user id>/`: user-specific app data on devices with FBE
- `/data/local/tmp`: writable by user SHELL, used during app installation
- `/data/misc`: Bluetooth, WiFi, DHCP, VPN, etc. configuration files for Linux
- `/data/system`: many system configuration files, e.g. `accounts.db`
  - on devices with FBE, distinguish between `system_de` and `system_ce`, and between user accounts
- `/system/app` and `/system/priv-app`: apps bundled with the device (by standard Android, manufacturer, and network operator)

Standard directory and file structure for apps

- `/data/data/<application FQDN>/` as main directory with access rights only for this specific installed applications (and potentially others signed with the same private key)
- Subdirectories:
  - `lib/`: libraries required to run the application (native code called via JNI)
  - `files/`: custom files managed by the application
  - `databases/`: SQLite databases and journal files
  - `shared_prefs/`: XML format files storing primitive data types as preferences
  - `cache/`: custom files cached by the application, typically after downloading from web resources (e.g. browser, maps, etc.)
- **Plus arbitrary files in arbitrary directories on SD card**
  - **Hint**: use https://github.com/jduck/canhazaxs to discover “interesting” files accessible by some UID and/or GIDs on a running Android device
  - Note: shared file access limited starting with Android 10/11

Databases in Android: SQLite

- SQLite3 used as embedded SQL database
  - database files are self-contained and platform-independent
  - libraries available for many platforms and programming languages
  - SQL for data manipulation and query
- Android provides API for using SQLite databases in applications and uses them for system level storage, e.g.:
  - WebKit based browser stores icons, cache, geo location information, local website storage, etc.
  - mail client stores account details
  - Android system apps store WiFi passwords, etc.
  - Dropbox client stores all files, sync state, etc.

SQLite example: Dropbox

```
rene@iss:/tmp/com.dropbox.android/databases$ sqlite3 db.db
SQLite version 3.7.7 2011-06-23 19:49:22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables
android_metadata  dropbox           pending_upload
sqlite> .mode line
sqlite> select * from dropbox where _id = 11;
              _id = 11
            _data =
         modified = Tue, 23 Aug 2011 11:17:19 +0000
            bytes = 1268
         revision = 1493
             hash =
             icon = page_white_text
           is_dir = 0
             path = /Getting Started.rtf
       canon_path = /getting started.rtf
             root = dropbox
             size = 1.2KB
        mime_type = application/rtf
   local_modified =
      local_bytes =
      sync_status =
………
```

Example application data: SMS and MMS

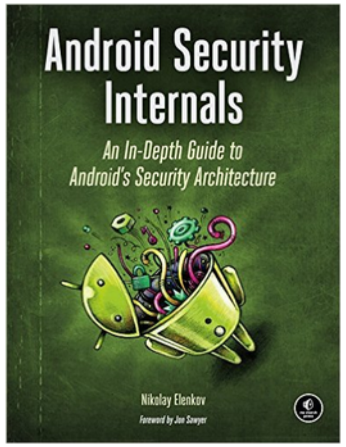
- Database at `/data/data/com.android.providers.telephony/databases/mmssms.db`
- In table `sms`, full SMS log with sender/receiver, date/time and content
- Note: newer Android Messages client may start encrypting E2EE messages at-rest

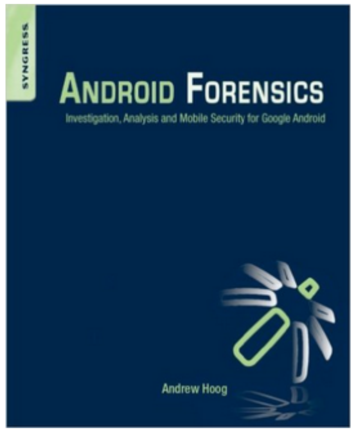
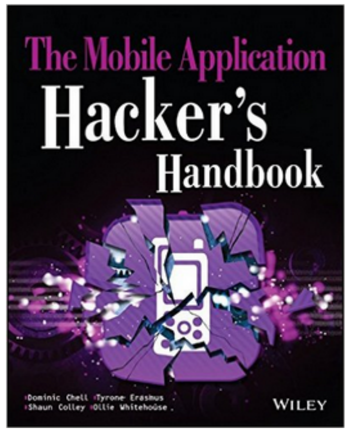
Example application data: Contacts

- Database at `/data/data/com.android.providers.contacts/databases/contacts2.db`
- In table `contacts`, all contact details
  - supports different applications in using the same database, e.g. Xing, Facebook, Skype, etc.
- In table `calls`, full call log

Reading list

- Nikolay Elenkov: “**Android Security Internals: An In-Depth Guide to Android's Security Architecture**”, No Starch Press, November 2014
- Joshua J. Drake, Pau Oliva Fora, Zach Lanier, Collin Mulliner, Stephen A. Ridley, and Georg Wicherski: “**Android Hacker's Handbook**”, Wiley, March 2014
- Andrew Hoog: “**Android Forensics: Investigation, Analysis and Mobile Security for Google Android**”, Syngress Media, July 2011
- Shaun Colley, Dominic Chell, Ollie Whitehouse, and Tyrone Erasmus: “**The Mobile Application Hacker's Handbook**”, Wiley, February 2015



Online resources

- https://usmile.at/symposium/program
  - *Nikolay Elenkov: Android's security architecture*
    https://usmile.at/sites/default/files/androidsecuritysymposium/presentations/Elenkov_AndroidSecurityArchitecture.pdf
  - *Pau Oliva Fora: Assessing Android applications using command-line fu*
    https://usmile.at/sites/default/files/androidsecuritysymposium/presentations/OlivaFora_AssessingAndroidApplicationsUsingCommandLineFu.pdf
- Google official docs
  - https://source.android.com/devices/tech/security/enhancements/
  - https://source.android.com/devices/tech/security/index.html
- Yanick Fratantonio’s class materials on Android app reverse engineering: https://mobisec.reyammer.io
- http://nelenkov.blogspot.co.at/
  - https://plus.google.com/communities/118124907618051049043
  - http://forum.xda-developers.com/general/security
- http://www.droidsec.org/wiki/
- https://www.nowsecure.com/blog/
- https://labs.mwrinfosecurity.com/publications/

JOHANNES KEPLER
UNIVERSITY LINZ

Thanks for your attention!

Further questions:
Email: rm@ins.jku.at
Twitter: @rene_mobile

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at