Software Security Course

Mobile App Security

Dimitrios A. Glynos
{ daglyn at unipi.gr }

Department of Informatics
University of Piraeus

Part I

Mobile App Security

Mobile Apps

- Applications that can be installed by users on a mobile device
  - “smart” phone
  - tablet
  - vehicle “infotainment” system
  - ...
- Usually downloaded from a controlled “market place”
- Packaged as a software “bundle”
- Apps may run on a personal or business device and may thus handle ***personal*** or other ***sensitive*** data

Mobile App Characteristics

- **HTML5 App**: Installable app written in HTML5 / JavaScript / CSS, running on top of a standard "web view"[1]
  - Compare this to a **Responsive HTML5 Web Application**, that simply runs on the mobile device browser
- **PWA**[2]: A website that can register an "offline" version on the device Homescreen
- **Native App**: Mobile app developed on top of foundation libraries (e.g. on top of Android Java libraries)
  - Native Apps may introduce system level components (native libraries etc.)
- **Hybrid App**: An HTML5 installable app that also utilizes system components (using a JavaScript-to-native-code bridge[3])
- Note: App Clips or Instant Apps allow users to experience (limited) capabilities of an app without installing the full app

[1] A component provided by the platform mimicking the rendering of a browser tab

[2] Progressive Web App

[3] see WKScriptMessageHandler on iOS, and addJavascriptInterface or WebMessagePort.postMessage on Android

Vulnerability Landscape

- Buffer overflows in native components
  - custom C/C++ library
  - system library
  - browser vulnerabilities
- Business logic errors / wrong implementation of security controls
  - CVE-2013-4787 duplicate filename in apk
- Privacy issues
  - communication with server is susceptible to MITM attacks
  - user tracking
  - insecure storage of sensitive data

Risk

- Modification of app state
- Leakage of sensitive information
- Complete device compromise

Threat Agents affecting an app

- An actor performing an attack on a Wi-Fi network
- An actor that has pushed a malicious app to a store
- An actor that has convinced a victim user to download a malicious resource
- An actor that has compromised another app on the victim's device
- An actor that has sent a malicious message to a victim user through the cellular network
- An actor that has compromised a service on the internet
- An actor that lies in the vicinity of a victim device
  - NFC attacks
- An actor that has short-term access to the device
  - An 'evil maid' having brief access to a locked device
- An actor that has longer-term access to the device
  - A thief

Computing on a mobile device

- The OS and mobile device frameworks offer a number of security controls for applications to use
- Developers of critical applications consider mobile platforms as hostile execution environments
- “rooted” / “jailbroken” devices
    - Devices where the firmware has been modified by users in order to gain administrative capabilities
    - Basic security controls like the execution of only signed binaries have been disabled
    - Some software vendors consider these setups as insecure and do not allow further execution
    - Others cannot overlook this growing customer base
- Proactive Application Binary Protection
    - Obfuscation
    - Static and Dynamic Tamper protection

Security Tests for Mobile Apps

- Black box application security tests to app and related web service(s)
  - Most of the time these require access to a rooted / jailbroken device, so as to carry out in-depth inspection of app artifacts and behaviour
- Code reviews
- Bundle audits

Part II

Android

Android

- OS for smart phones based on the Linux kernel
- Developed by Google
- Based on standards set by the Open Handset Alliance[4]
- Most popular OS for smart phones
  - In the first quarter of 2024, Android devices accounted for 71% of the mobile device market share (source: statcounter)
- Although Android is open source it is bundled with binary drivers and closed source applications (e.g. Google Mobile Services)

[4] An initiative to align with the multiple Android device makers and chipset vendors

Secure Boot and Firmware Upgrading

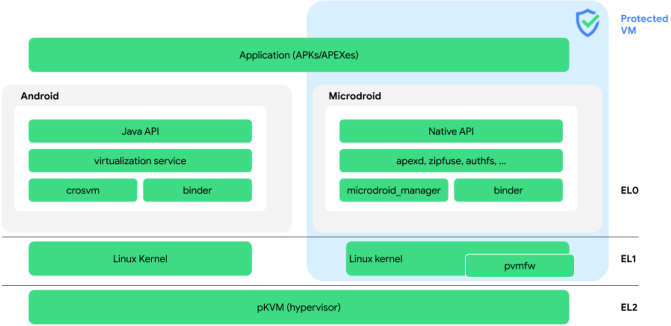
- An Android OS installation typically consists of Google (and contributed) Android code, Device maker code and Chipset vendor code
- Android *Project Treble* wishes to separate the Chipset vendor code from the rest of the OS code, to make it easier for a Google release to be pushed to the Device maker (and thus to the End user)
- Since Android 8, Android provides to vendors reference Android Verified Boot code
- Android Verified Boot aims to achieve the following:
  - Verifying that the signed firmware that is loaded is one that the Device maker considers as authentic.
  - Verifying that the pushed firmware version is not an old one (protection via RPMB[5] hardware).
- Some Device makers (incl. Google) allow users to flash[6] the bootloader (and thus any firmware to the device)

[5] Replay Protected Memory Block

[6] aka *unlocked bootloader*

Android Virtualization Framework

- Android 13 introduced the Android Virtualization Framework
- Part of Android's kernel code, the Protected Kernel-based Virtual Machine (pKVM), is executed at boot time as a *hypervisor* at a higher privilege level (EL2 ARM exception level) than the Android kernel (EL1)
- Android may now execute sensitive workflows, such as upgrade-time system rebuilds, in a protected guest VM.



Confidentiality Controls

- Instead of full disk encryption, Android supports file-based encryption
  - Files are encrypted with AES-256-XTS[7]
  - Credential Encrypted storage, is the default app storage and is only available after first unlock.
  - Device Encrypted storage, is an app storage that is available just after boot (before device unlock).
- Android supports the use of a TEE[8] or SE[9]
  - The TEE (or SE) handles cryptographic material, and makes sure sensitive data (like fingerprint data) are not exposed to the untrusted world of the main processor context.
  - Trusty is a reference implementation of a TEE OS and TEE services.
- Apps may request to generate / maintain keys in a KeyStore (framework component) which utilizes the hardware-backed KeyMaster (service).

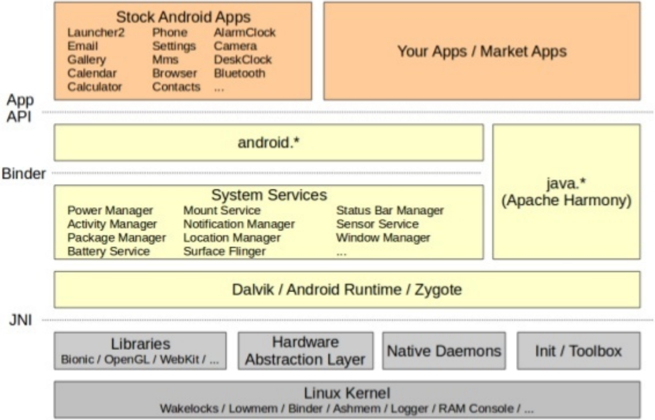
---

[7] or *Adiantum* if no hardware acceleration is possible

[8] Trusted Execution Environment

[9] Secure Element

Android Architecture



Source: `http://www.slideshare.net/opersys/inside-androids-ui`

APK file

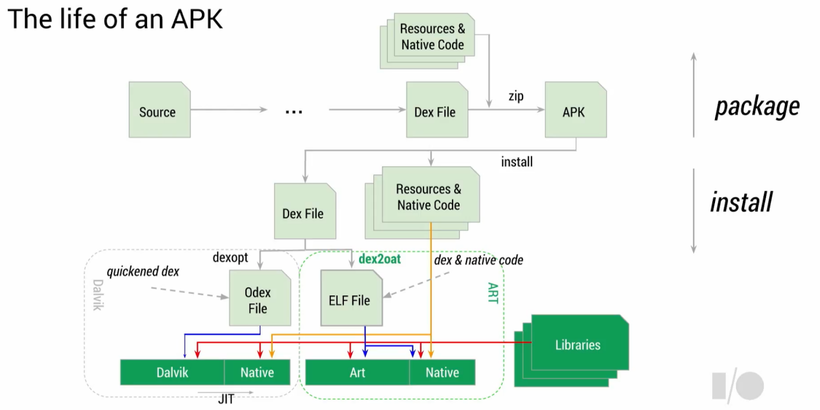
- Container of application resources (application “bundle”)
- Usually downloaded through Google’s market place (Google Play)
- A signed (by developer) and compressed archive of files

```
$ jarsigner -verbose -verify foo.apk
...
sm    367112 Tue Oct 01 10:38:02 EEST 2013 assets/fonts/arial.ttf
sm    292616 Tue Oct 01 10:38:02 EEST 2013 classes.dex
sm    139340 Fri Sep 20 16:09:54 EEST 2013 lib/armeabi/libjpeg.so
sm     13024 Tue Oct 01 10:38:02 EEST 2013 AndroidManifest.xml
sm     80292 Tue Oct 01 10:37:46 EEST 2013 resources.arsc
sm      4247 Fri Sep 20 16:10:28 EEST 2013 res/drawable/aa.png
        9728 Tue Oct 01 10:38:02 EEST 2013 META-INF/MANIFEST.MF
        9781 Tue Oct 01 10:38:02 EEST 2013 META-INF/CERT.SF
         863 Tue Oct 01 10:38:02 EEST 2013 META-INF/CERT.RSA
jar verified.
```

Android App Runtime

- Android Java code is compiled to class files which are then translated to DEX files (bytecode suitable for the Dalvik register-based VM)
- Dalvik's allocation, garbage collection and JIT compilation times were hurting performance
- In version 5, Android moved from a Dalvik VM-based runtime to ART[10]
- In ART, DEX files are Ahead-of-Time compiled to ELF64 OAT shared libraries (with eager object pre-initialization)
- The ART runtime is now a mixture of loaded native code, a VM to interpret DEX code, and a JIT mechanism to compile parts of DEX based on usage profiling

[10] Android RunTime



Source: Google I/O 2014 presentation "The ART runtime"

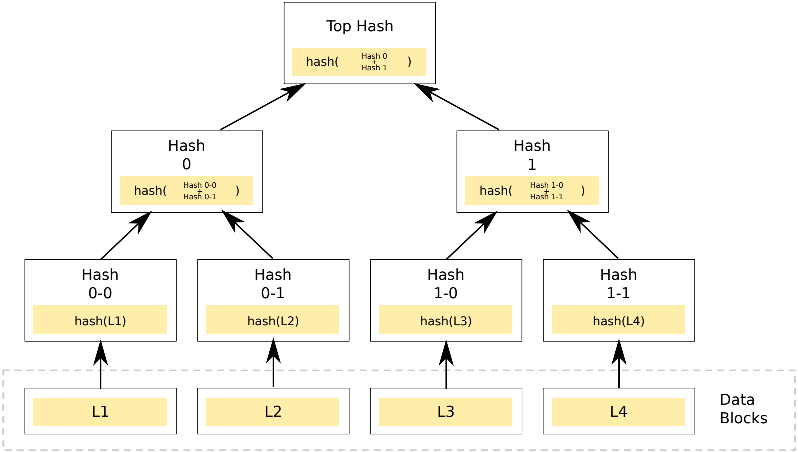
APK signing

- Google **does not** maintain a Certificate Authority to verify Developer signing certificates
- In recent years, Google recommends to developers to have Google **manage their signing key**
- Google also recommends uploading the original APK artifact for publishing, using (another managed) **upload key**
- As of 2024, the APK v4 signature scheme is the default one used
  - v4 supports the (optional) use of a Merkle Tree[11] to efficiently hash progressive APK downloads
  - v3 signatures allowed for signing key rotation
  - v2 signatures verified[12] the whole of the APK zip archive
  - v1 signatures verified the file contents of the zip archive, but not the zip directory.. (Java's default JAR signing)

[11] Merkle Tree data stored in separate signature file apk-name.apk.idsig

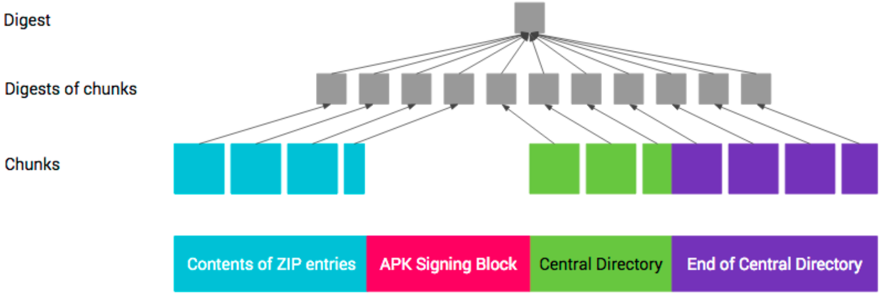
[12] Introduced a *signing block* within the APK zip structure

On Merkle Trees



Source: Wikipedia article on Merkle Tree

Merkle Tree and APK Signing block



Source: Android documentation on APK signing

App, System and Device Integrity Checks

- Critical applications need to know the application, operating system and device integrity (e.g. unlocked bootloader) status
  - If the application has been tampered with, then the application's web API must not accept critical transactions
  - If the OS has been tampered with, then the application must not rely on its controls to handle critical data
  - If the device configuration is not at a secure state, then any attestation provided by the OS may be false
- Google offers the Play Integrity API to receive such an attestation regarding the device, system and app integrity
  - Google is planning to also roll out an install-time integrity test to select (opt-in) partners in Android 15
- Heuristics for such controls have traditionally been provided by Binary Protection Suites

Application Sandboxing

- Applications of different authors run on the same device
- Android needs to contain their execution to protect other apps (and the system)
- Containment is implemented through Application Sandboxing
- Application Sandboxing in Android is implemented in three levels
  - UNIX File Permissions
  - SELinux Mandatory Access Control
  - SECCOMP sandbox

UNIX File Permissions

- No /etc/passwd or /etc/group
- Hard-coded UserIDs and GroupIDs (`#define AID_SYSTEM 1000`)
- Each application receives new UserID dynamically upon installation
- Capabilities (members of AID_INET_ADMIN are allowed to configure network interfaces)
- “Sandboxing” through tight file permissions, employing the principle of least privilege
  - In Android 6.0 the default permissions of an app home directory changed to 0700 (from 0751).
  - Since Android 10, files in the SD Card now have app-ownership permissions[13]

[13]As most devices today come with an embedded SD card which need not be FAT32 mountable by other devices.

Example UNIX File Permissions

```
$ adb shell
shell@android:/ $ ls -al
...
drwxrwx--x radio    radio                 2014-06-30 14:43 modemfs
drwxr-xr-x root     root                  2014-06-14 11:47 system
dr-xr-x--- system   sdcard_r              2012-01-01 08:17 storage
...
shell@android:/ $ cat /proc/mounts
...
/dev/block/mmcblk0p3 /system ext4 ro,noatime,user_xattr,acl,barrier=1,
data=ordered 0 0
/dev/block/mmcblk0p5 /data ext4 rw,nosuid,nodev,noatime,user_xattr,acl,
barrier=1,journal_async_commit,data=ordered,noauto_da_alloc,discard 0 0
...
$ ps
...
root      949   2     0      0     ffffffff 00000000 S binder
gps       1648  1     10928  964   ffffffff 00000000 S /system/bin/gpsd
u0_a20    15599 1643  487112 45000 ffffffff 00000000 S
com.sec.android.app.clockpackage
```

SELinux on Android

- SELinux is a Mandatory Access Control system developed by the NSA
- In Mandatory Access Control systems, the kernel keeps a policy of how processes may interact with resources (typically, files) and this cannot be changed[14] during system runtime
  - Compare this to the typical UNIX DAC[15] system where file permissions may change through `chmod(1)`
- SELinux associates security labels with “subjects” and “objects”
- Android separates processes in more than 60 SELinux *security domains*
  - A particular domain has access to specific resources and all further access is denied
  - Android 5 separated system resources (for system services) from app resources through different security domains
  - Android 6 separated app resources across physical users
  - Android 9 introduced per-app security domains

---

[14] SELinux *enforcing* mode

[15] Discretionary Access Control

SELinux example on Android

```
$ adb shell
### Show SELinux context of the /dev/wlan device file
$ ls -alZ /dev/wlan
crw-rw-rw- 1 system system u:object_r:wlan_device:s0 486, 0 1972-12-01 22:38 /dev/wlan
### Show the point where this was designated in the SELinux policy
$ grep -r wlan_device /vendor/etc/selinux/vendor_file_contexts
/dev/wlan u:object_r:wlan_device:s0
### What is allowed to use the wlan_device resource
$ grep wlan_device /vendor/etc/selinux/vendor_sepolicy.cil
...
(allow hal_wifi_ext wlan_device (chr_file (ioctl read write getattr lock append map open
watch watch_reads)))
### Which process holds (transitions to) the hal_wifi_ext attribute?
$ grep -r hal_wifi_ext /vendor/etc/selinux/vendor_sepolicy.cil |grep typetransition
(typetransition init_33_0 hal_wifi_ext_exec process hal_wifi_ext)
### Which executable file(s) take the hal_wifi_ext_exec attribute during execution?
$ grep hal_wifi_ext_exec /vendor/etc/selinux/vendor_file_contexts
/vendor/bin/hw/vendor\.google\.wifi_ext@1\.0-service-vendor u:object_r:hal_wifi_ext_exec:s0
/vendor/bin/hw/vendor\.google\.wifi_ext@1\.0-service-vendor-lazy
u:object_r:hal_wifi_ext_exec:s0
```

seccomp-bpf on Android

- SECCOMP is a Linux kernel facility that limits the system calls that are available to a **process**
- `seccomp-bpf` uses the Berkeley Packet Filter language to implement the system call filtering, achieving O(log n) complexity (due to the use of a binary search tree)
- Android applies a seccomp-bpf filter at the *Zygote*, the creator of all app processes
  - On Android 8 `seccomp-bpf` blocks 17 out of 271 Linux kernel system calls on the ARM64 architecture[16]

[16] https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html

Example rules of Android's seccomp-bpf

Disabled system calls from https://android.googlesource.com/platform/bionic/+/8dc9f46a3f1a47cddfbb22c89a939239378f42f8/libc/SECCOMP_BLOCKLIST_APP.TXT

```
int setgid:setgid32(gid_t)                     lp32
int setgid:setgid(gid_t)                       lp64
int setuid:setuid32(uid_t)                     lp32
int setuid:setuid(uid_t)                       lp64
int setregid:setregid32(gid_t, gid_t)          lp32
int adjtimex(struct timex*)                    all
int clock_adjtime(clockid_t, struct timex*)    all
int clock_settime(clockid_t, const struct timespec*)  all
int settimeofday(const struct timeval*, const struct timezone*)   all
int chroot(const char*)                        all
int mount(const char*, const char*, const char*, unsigned long,
          const vid*)                          all
int umount2(const char*, int)                  all
int swapon(const char*, int)                   all
int swapoff(const char*)                       all
...
```

App Permissions

- App Permissions are described in a per-app `AndroidManifest.xml`
- Permission android.permission.CAMERA grants the app access to camera functionalities
- `android.permission.CAMERA` maps to UNIX group “camera”
- Other examples:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.CALL_PHONE" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

Permissions for Intent Filters

- Intent filters for Activities, Broadcast Receivers, Services
  - *An Intent is a messaging object you can use to request an action from another app component.*
  - *An Activity is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map.*
  - *Broadcast receivers are implicit event receivers*
  - *A Service is an application component that can perform long-running operations in the background and does not provide a user interface.*
  - Developers may apply intent filters to Activities, Broadcast Receivers and Services using Android or *custom* Permissions. Example:

```
<permission android:name="org.foo.permission.UNPACK_FILE"
 android:protectionLevel="signature" />
...
<activity android:name=".InstallWidgetActivity"
android:permission="org.foo.permission.UNPACK_FILE"/>
...
```

Permissions for Content Providers

- *Content providers manage access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security.*
- Permissions may be applied to content providers. Example:

```
<provider android:name="org.foo.SeriesProvider"
android:writePermission="org.foo.permission.WRITE"
android:authorities="org.foo.data" />
```

Types of Permissions

- Install-time Permissions
  - Normal permissions - low risk permissions
  - Signature permissions - making sure only package with same signature (or OEM package) may perform the action
- Run-time permissions
  - Dangerous permissions - user is challenged to accept these dangerous permissions
  - Special permissions - for OEM or privileged apps (e.g. drawing over other apps), enabled through Settings

Binder

- **Binder** is an Inter-Process Communication (IPC) mechanism for Android apps and services
- The binder kernel module exposes three devices that allow for message passing over shared memory
  - /dev/binder - for framework/app communication
  - /dev/hwbinder - for framework/vendor hardware-related communication
  - /dev/vndbinder - for vendor/vendor communication
- Binder facilitates the transfer of intent data to Activities, content from Content Providers etc.

Android Deep Links

- *Deep-links* are URIs that an app A (or website B) may present to have the user open an Activity in app C

```
<activity
  android:name="com.example.myapp.TestActivity" ...>
  ...
  <intent-filter>
     <action android:name="android.intent.action.VIEW" />
     <category android:name="android.intent.category.DEFAULT" />
     <category android:name="android.intent.category.BROWSABLE" />
     <data android:scheme="myapp" android:host="test" />
  </intent-filter>
</activity>
```

- In this example, clicking a “myapp://test” URI in app A (or website B) will start the `TestActivity` in app C

From Deep Links to verified Android App Links

- a single Deep Link URI scheme may be registered with multiple apps on a mobile device
  - e.g. a "`pdf://`" scheme for opening PDF files with Google Drive, Adobe Acrobat Reader etc.
- Android will by default let the user select which app will handle the scheme
- An *Android App Link* is a URI from a website B, that when triggered Android will **automatically select** a particular app C on the mobile device (based on information found in website B) to handle the Activity Intent
  - Requires JSON file[17] with app signer's certificate digest, hosted on site B
  - Requires the `autoVerify` attribute on the `intent-filter`

```
<activity ...>
   <intent-filter android:autoVerify="true">
   ...
   </intent-filter>
</activity>
```

[17] found under "https://website-b.domain.name/.well-known/assetlinks.json"

Part III

Android App Vulnerabilities

OWASP Mobile Top 10

- OWASP maintains a top 10 list of Mobile App Risks
  - M1: Improper Credential Usage
  - M2: Inadequate Supply Chain Security
  - M3: Insecure Authentication / Authorization
  - M4: Insufficient Input / Output Validation
  - M5: Insecure Communication
  - M6: Inadequate Privacy Controls
  - M7: Insufficient Binary Protections
  - M8: Security Misconfiguration
  - M9: Insecure Data Storage
  - M10: Insufficient Cryptography

OWASP MASVS

- OWASP also maintains the Mobile Application Security Verification Standard (MASVS) which covers the following areas
  - MASVS-Storage
  - MASVS-Cryptography
  - MASVS-Authentication (and Authorization)
  - MASVS-Network (Communication)
  - MASVS-Platform (Interaction)
  - MASVS-Code (Quality)
  - MASVS-Resilience
  - MASVS-Privacy

Trouble with permissions

- Incorrect permissions on event triggers
  - Any app may trigger a particular sensitive action (e.g. bring up an app's password dialog)
- Incorrect app file permissions
- App requires excessive permissions
- Incorrect system component permissions

Bad use of Android API

- Caching sensitive form data
- Enabling Javascript on a WebView
- Dangerous Javascript bridge to Java code

Communication channel issues

- Content delivered over HTTP (i.e. no SSL)
- No Certificate Pinning
- Bad certificate validation code

Hiding Data

- KeyStore can be used to manage cryptographic keys
- Sensitive assets should be (symmetrically) encrypted before they are stored on disk
- Sensitive DB data must be encrypted before stored
- Whole DB’s can also be encrypted through projects such as SQLCipher

Hiding Code

- Android Class files are transformed to DEX bytecode for Dalvik VM
- Tools like `dex2jar` transform DEX bytecode to JARs with class files
- Java code that has no obfuscation can be trivially reversed to something that resembles the original source code
- Obfuscated Java code requires some more work during reversing
- In all cases, however, DEX disassemblers (like `baksmali`) produce output which is easier to follow than, say, x86 / ARM assembly
- Many vendors choose to move sensitive code to native libraries for which there exist better obfuscation methods
- Tamper protection software suites are also applied to such critical applications

Part IV

Conclusions

Other Important Considerations

Some important concepts that we have not covered in this lecture

- Rooting / Jailbreaking a device, and maintaining a fleet of such devices for testing
- Removing certificate pinning during testing
- Bypassing binary protections during testing
- Testing services requiring an authentic binary and/or environment (e.g. Platform wallet services)
- Static patching and resigning
- Dynamic patching of app, framework component and/or native component

Conclusions

- Mobile apps bring a unique personalized experience to software applications
- The security features offered by mobile platforms (e.g. managed keystore etc.) have made some software vendors gradually switch from web app implementations to mobile app implementations
- In the same time, mobile apps bring new issues to the vulnerability landscape due to
  - the unique features offered by the mobile platforms
  - their exposure to potentially hostile networks and actors
  - their exposure to an untrusted execution environment that may potentially contain malicious 3rd party apps
- The app security model is continually changing and is expected to change even more in the next few years..

Further reading material

- Android Security Paper, 2023 edition
- Android Application Secure Design/Secure Coding Guidebook by JSSEC
- Android Internals::Developer's View
- Android Internals::Power User's View
- The Mobile Application Hacker's Handbook
- Android Security Internals (2014, N. Elenkov)
- Android Hacker's Handbook (2014, J. Drake et al)