Android Application Security I

Mobile Security 2024

Florian Draschbacher
florian.draschbacher@iaik.tugraz.at

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

Reminder: Practicals

- Decide on a topic for Assignment 2 **until today 23:59**!
  - Send me an email with your topic and team members

- Assignment 1 deadline: **April 19th!**
  - Any questions?

Introduction

We learned about the **low-level security of Android** last time.

- Application security teasered through app sandbox

This week: Platform perspective on application security

- What role do applications play on the system?
- What makes an application?
- How can we ensure apps are not malicious?
- How are apps distributed?
- How can apps be analysed and modified?
- How can apps prevent being modified?

Android Platform Security Model

2021 Whitepaper by Android Security Engineers Source: https://dl.acm.org/doi/pdf/10.1145/3448609

- Last updated in 2023

- Sensitive operations must be allowed by all 3 stakeholders
  - User
  - Developer / Application
  - Platform

- **Sensitive operation:** E.g. access to app-private files
- **Developer consent:** APK signature

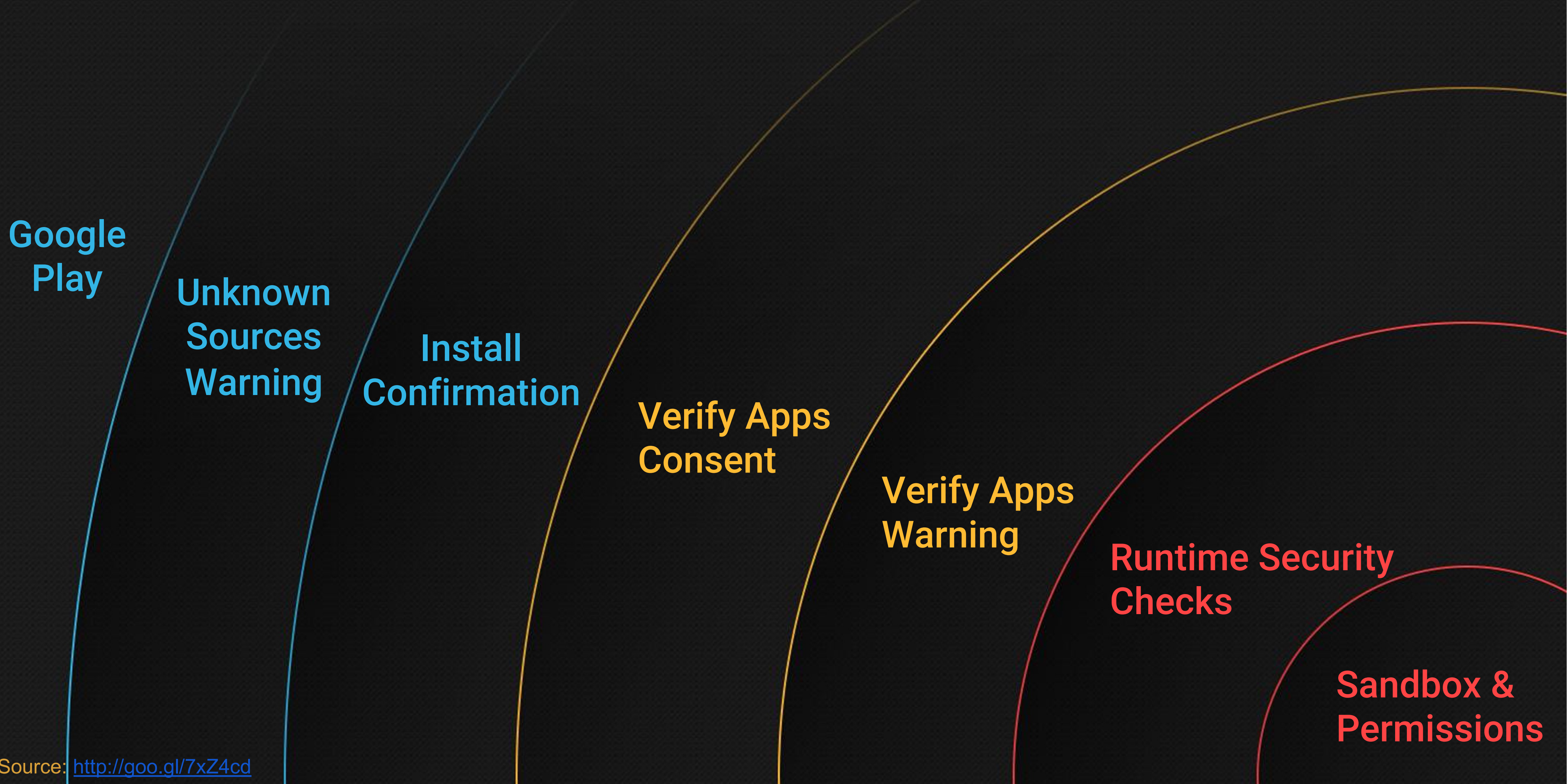
Android Applications

- Not all applications on Android are user-installed
  - Devices ship with a considerable number of apps preinstalled

- Four different privilege levels of Android applications:

1. **System apps**: Signed with firmware keys by device manufacturer
2. **Privileged apps**: Preinstalled to `/system/priv-app/` directory
3. **Preinstalled apps**: Preinstalled to `/system/app/`
4. **User apps**: Not preinstalled and not signed with firmware keys

Multiple Layers of Defense



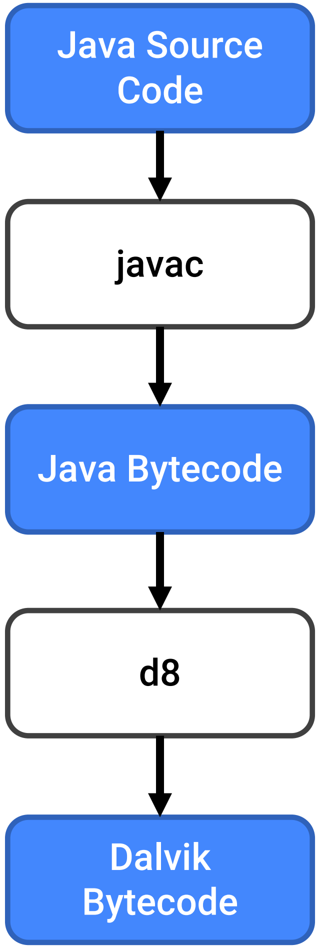
Source: http://goo.gl/7xZ4cd

Android Applications

Android App Development

Android apps are developed in Java* and compiled to Dalvik Bytecode

* Or other languages that compile to Java Bytecode (such as Kotlin)



Advantages:

- Apps compatible with all CPU architectures
- Use existing tools and libraries
- Convenient high-level language
  - Garbage collection
  - Memory safety

Disadvantages:

- Slower than native code

Android Runtime

Responsible for executing Dalvik bytecode (DEX) on device

- ART Runtime:
  - **Interpretation**: Quick start of newly installed apps
  - **Ahead-Of-Time** compilation
  - **Just-In-Time** compilation

- Parts of apps may also be compiled from C/C++ to native machine code
  - Java Native Interface (JNI)

Android App Structure

Applications are packaged into APK files during build

ZIP file containing

File / Folder	Purpose
`assets/`	Raw asset files, e.g. textures for games. Identified by filename
`AndroidManifest.xml`	Meta data about app: Required permissions, app components, …
`classes.dex`	All classes in Dalvik bytecode
`lib/`	Compiled native code (C/C++) as shared-objects (.so) Platform-specific versions, e.g. ARM („armeabi“), ARMv7, x86, MIPS
`res/`	App resources, e.g. GUI layouts in XML format, graphics, colors, …
`resources.arsc`	Index of resources + compressed string resources

Application Signing

APK files are signed by the application developer

- Self-signed X.509 certificate
- Package update requires same certificate

- Three different signing schemes
  - v1: Signed individual uncompressed files, but not ZIP metadata
  - v2: Signature over complete compressed data **Android 7.0+**
  - v3: Extends v2 with support for key rotation **Android 9+**
  - v4: Signature in separate file, supports verification during app download **Android 11+**

Source: source.android.com

Signing Dilemma

Application Signing != Code Signing

- Android supports code loading at runtime
  - Useful for shared frameworks, testing, dynamic addon loading
  - Can also be loaded from Internet!
  - By loading & executing any other application‘s code (`createPackageContext` API)

Problems

- Malicious app can evade detection by application analysts
- Code injection attacks on benign apps may affect millions of users!

Signing Dilemma

What if…

- Code is loaded from external domains via HTTP
  - MITM! → Possible for attackers to modify / replace downloaded code
- Code is loaded and stored on device‘s file system
  - E.g. Directories on external storage (SD card)
  - Other apps may tamper additional code before loading
- Applications forge package names
  - Package name not displayed during installation
  - First-come, first serve → malicious app could be installed prior to legitimate one!

Conclusion: <u>Real</u> code signing (as on iOS) would

- …mitigate many exploits & attack surfaces
- …ease static application analysis significantly!

Application Signing: v1 vulnerability (Janus)

APKs signed with v1 signature scheme may be modified without breaking signature

- Signature scheme v1 only signs file entries in the ZIP

- DEX code can be embedded in the ZIP file
  - ZIP file: Trailer at end points to file entries
  - DEX file: Header at start points to following data chunks
  - A file can be a valid DEX file and ZIP file at the same time

- Android runtime supports running APK or DEX files
  - File type confusion can be exploited

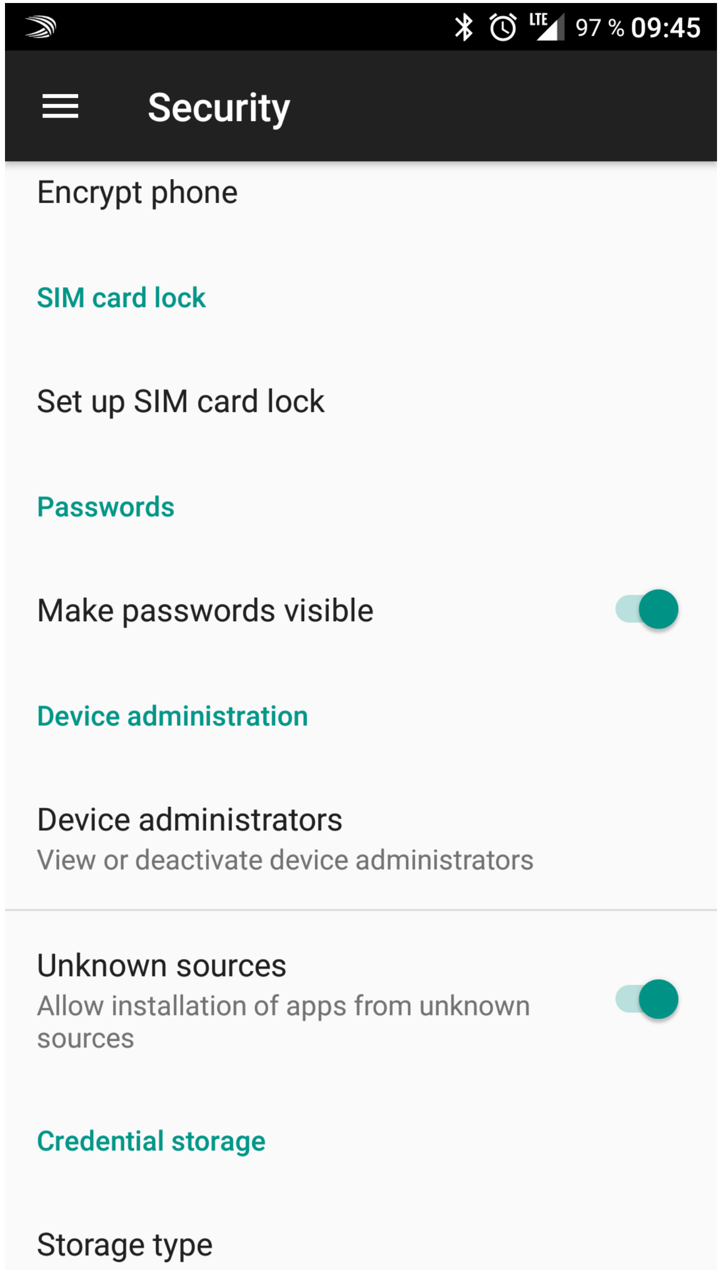
Source: guardsquare.com

App Distribution

App Sources

Android allows installation of apps from

- Google Play
  - Trusted by default
  - Requires license from Google
- Third-party app stores
  - Amazon, F-Droid, Samsung, ….
  - Popular in regions where Google Play is unavailable (China)
  - Requires explicit permission to install apps
    - Or preinstalled by vendor as privileged app
- From file system
  - If app available as .apk file



IAIK TU Graz

Google Play

- Pre-installed on (almost) all Android devices
- User needs Google account
  - App retrieval limited by customer age and geographic location
- Developer needs Google account
  - Personal data validated and exposed publicly
  - 20$ one-time fee (+30% on all sales / 15% for small developers)

Security mechanisms

- Automated and manual app reviews

Google Bouncer (2012)

In a nutshell…

- Dynamic & static runtime analysis of every uploaded app
- Emulated Android environment based on `qemu`
- Runs for 5 minutes
- Uses Google‘s infrastructure / IP addresses for external network access

Analysis

1. Explore app by emulating UI input, clicking, etc.
2. Check for known malware
   - Malware signatures, heuristics, similarities, source / developer, third-party reports
   - If flagged malicious → Manual analysis by human being
   - If confirmed malicious → Goodbye Google account ☺

Source: googlemobile.blogspot.com

2012: Playing with the Bouncer

- Remote connect-back shell by J. Oberheide and C. Miller
  - https://www.youtube.com/watch?v=ZEIED2ZLEbQ
- Construct strings at runtime
  - If Bouncer statically detects `/system/bin/ls:` never executed dynamically
- There are various ways to evade detection
  - Only load malicious code after 5 minutes
  - ...

Conclusion: Automated app analysis is never perfect!

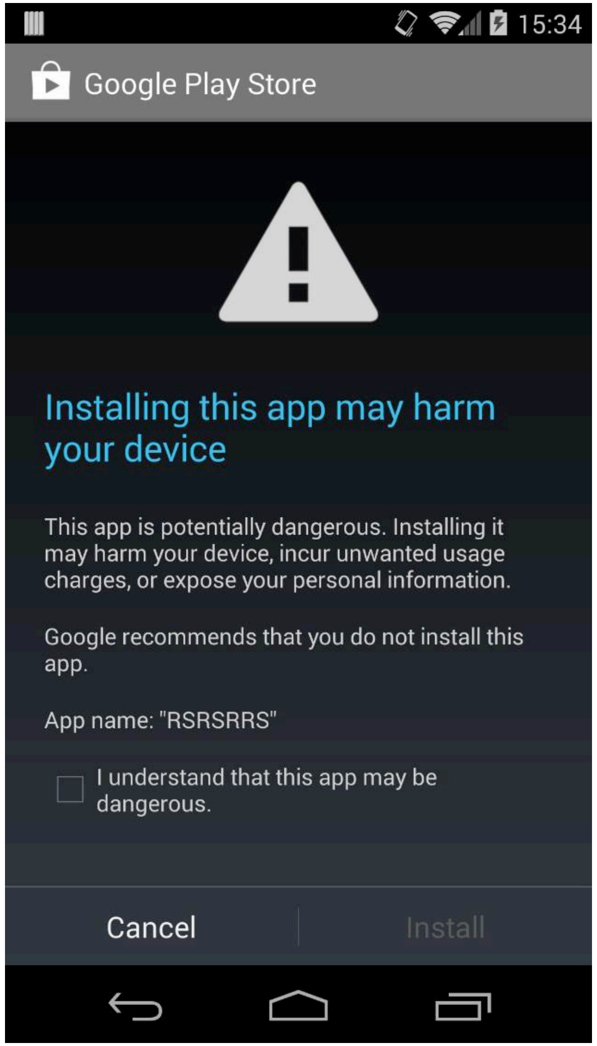
Source: J. Oberheide, C. Miller: "Dissecting the Android Bouncer"

Verify Apps (2012)

Can be disabled by user!

App scans extend to user side

- Apps are verified / categorized prior to install
  - Remote database with malware signatures
- Sends log data, related URLs and device info to Google
- Warn or block potentially harmful apps (PHA)



Source: androidauthority.com

Verify Apps (2014 – 2017)

- Constantly scans installed apps instead of just at installation
  - React to threats that only became known after installation

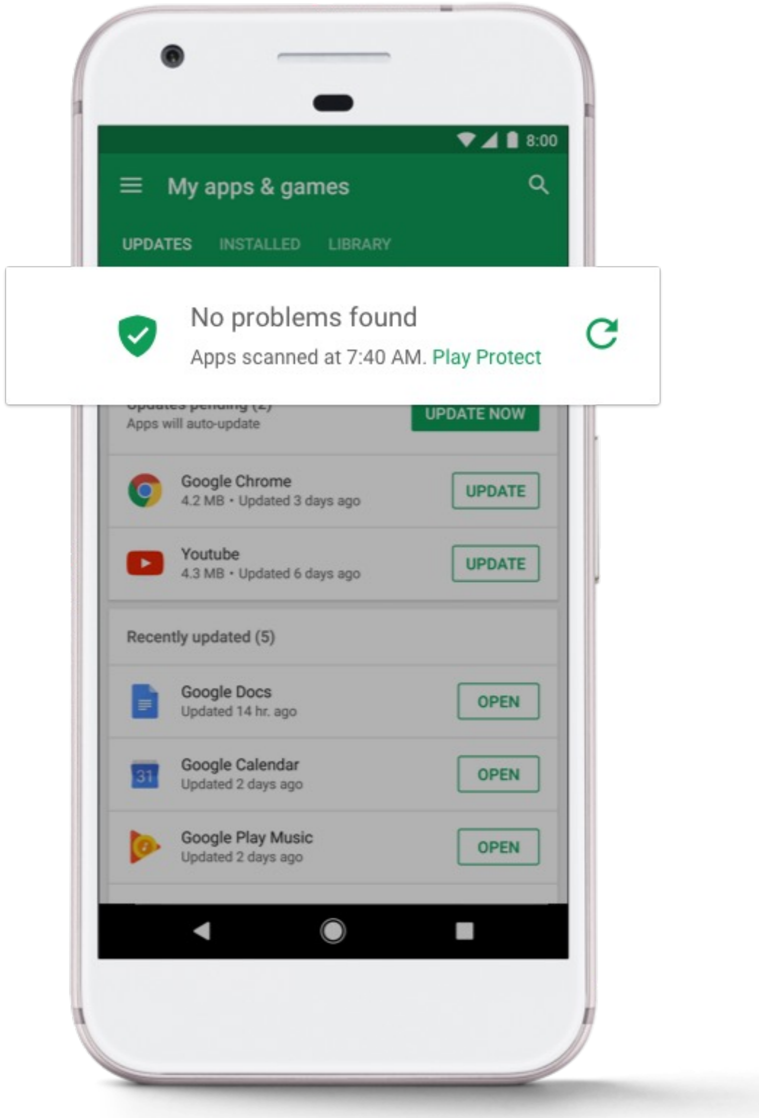
- Monitor device state
  - Dead or Insecure: A device stopped checking up with Verify Apps server
  - Likely either because malware disabled VA or device had to be reset
  - Both indicate a previously installed app was malicious
  - DOI app: Many devices DOI after installing this app

- The introduction of machine learning into Google’s app analysis

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

Google Play Protect (2017-)

- Google Bouncer and Verify Apps were rebranded
- „Advanced similarity detection“
  - Google claims to use machine learning algorithms
  - No implementation details documented
- Unknown apps can be sent to Google servers
  - For further analysis
- 2021: Separate app
  - No longer integrated into Play Store
  - Still depends on Google Play Services



Can still be disabled by user!

Sources: security.googleblog.com, android.com, 9to5google.com

IAIK TU Graz

Pirated Applications

- (Paid) APK files can be
  - Extracted from Android devices
  - Modified and resigned
  - Redistributed on the Internet

- Pirated applications
  - Paid applications for free, removed license checks, …
  - Commonly augmented with malicious components

- Android is prone to **"Repackaging Attacks"**
  - Not possible on (unjailbroken) iOS!

Android Application Bundle (AAB)

- Developers used to submit apps to Google Play as signed APKs
- **Problem:** Universal APKs contain resources needed only for other devices
  - Example: x86 native libraries wastes space on ARM device

- **Solution:** Android Application Bundle
  - Developers submit app to Google Play as signed AAB
  - Contains all compiled code and resources
  - Google Play generates and signs optimised APKs for specific devices
  - **Mandatory** for new apps since 2021

Code Transparency

- Problem:
  - APK Signature is fundamental to Android Platform Security Model
  - With AAB, Google now has control over APK signature!

- Solution: Code Transparency
  - Developers sign compiled code stored in AAB file
    - JWT file copied to all generated APK files
  - Developers can download APK generated by Google Play
    - Ensure the Code Transparency is still valid

Reverse-Engineering & Analysis

Decompiling DEX Code

- DEX code can be disassembled to SMALI IR using *apktool*
  - Process is reversible -> Repackaging with added instrumentation code

```
.super Ljava/lang/Object;

.method public static main([Ljava/lang/String;)V
    .registers 2
    sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;
    const-string v1, "Hello World!"
    invoke-virtual {v0, v1}, Ljava/io/PrintStream;-
>println(Ljava/lang/String;)V
    return-void
.end method
```

```
public static void main(String[] args) {
    System.out.println("Hello World!");
}
```

- Alternatively, partly decompile the code to Java using *JADX*
  - Usually not reversible (some needed information lost through compilation)
  - Easier to analyse

IAIK TU Graz

Debugger

- Inspect and modify internal state
- Follow and manipulate control flow
- Android OS only allows attaching debugger to apps marked as debuggable
  - Usually automatically added by Android Studio for debug builds
- Manifest can be patched to make production builds debuggable!
  - Changes signature though

Native Code Analysis

- Applications may implement some logic in native libraries
  - Faster performance
  - Use existing C/C++ libraries
- Machine code harder to reverse-engineer than DEX code
  - Non-exported symbols stripped
  - Control flow difficult to reconstruct
- Tools:
  - Ghidra (Open Source)
  - HexRays IDA Pro (Commercial $$$)

Runtime Manipulation

Apps are executed through the ART runtime ➔ opportunity for manipulation

- ART keeps method tables for every class
  - Can overwrite pointers to exchange method implementations
  - If method JIT/AOT-compiled: Some assembler vodoo required
- Xposed Framework: Embed manipulation primitives in Zygote process
  - Make every app process (forked from Zygote) load Xposed modules
- Frida: Either inject into running process (root) or into APK file
  - Dynamically manipulate app through Javascript console

App Attestation

Application Repackaging

- APKs signature only protects against malicious **update**

- An attacker may simply
  - Obtain the legitimate APK of a banking app from Google Play
  - Modify it to redirect all new transactions to the attacker’s account
  - Sign resulting APK with a new developer key
  - Redistribute it
    - Upload to Google Play
    - Find a way to replace app for existing users (social engineering, …)
  - Profit

Static Centralised Solution

- Why not have app stores ensure package name uniqueness at upload?
  - Package name can easily be changed
- Similarity check for apps uploaded to app stores
  - Might be fooled
- Also: APKs may be distributed through other channels
  - 3rd party app stores
  - Internet

Dynamic Repackage Proofing

- The developer of the banking app needs a way to check APK integrity

- What about checking the APK signature at runtime?

```
public static boolean checkSignature(Context context, String legitimate) {
    PackageInfo packageInfo = context.getPackageManager().getPackageInfo(getPackageName(),
        PackageManager.GET_SIGNATURES);

    for (Signature signature : packageInfo.signatures) {
        String sha1 = getSHA1(signature.toByteArray());
        return legitimate.equals(sha1);
    }
}

...

if (!checkSignature(context, APP_SIGNATURE)) {
    System.exit(-1);
}
```

Problem solved?

Dynamic Repackage Proofing: Problems

- Checking the APK Signature in-process at runtime is not enough

- The malicious party may simply remove the signature check

```
public static boolean checkSignature(Context context, String legitimate) {
    return true;
}
```

- Can we fix this?

Improved Dynamic Repackage Proofing

- We need to prevent attackers from
  - Locating signature checks
  - Removing / bypassing signature checks

- Possibilities:
  - Implement check in native code
  - Encrypt DEX code in APK and decrypt at runtime
    - Bind decryption key derivation to untampered app state

- Problem finally solved?

Improved Dynamic Repackaging Proofing: Flaws

- All these solutions are effectively **security by obscurity**
  - Only increase the effort required for reverse-engineering and tampering

- Full reverse-engineering not even necessary:
  - Manipulate in-process ART runtime instance
  - Hook framework methods called for determining APK and runtime integrity
    - Spoof result of `PackageManager.getPackageInfo()`
    - Build a fake environment for victim code

- For a long time, this was as secure as it gets

App Attestation

- **Problem:** In-process dynamic APK signature checks may be circumvented
- **Solution:** Incorporate out-of-process checks

- Idea: System service attests APK signature to the app process
  - App may request information about its APK signature
  - Response is signed with asymmetric key
  - App may forward attestation to its backend server

- **Problem:** What if this system service is compromised?
- **Solution:** Incorporate TEE and also check OS integrity

App Attestation

- The definitive solution for ensuring integrity of app (APK) and system (OS)
  - Builds upon infrastructure for key attestation

General procedure

1. Backend server generates random challenge and sends it to app
2. App requests TEE to generate signed attestation including
   - Random Challenge
   - APK signature
   - Verified Boot state
3. App forwards attestation to backend server
4. Server aborts communication if attestation invalid

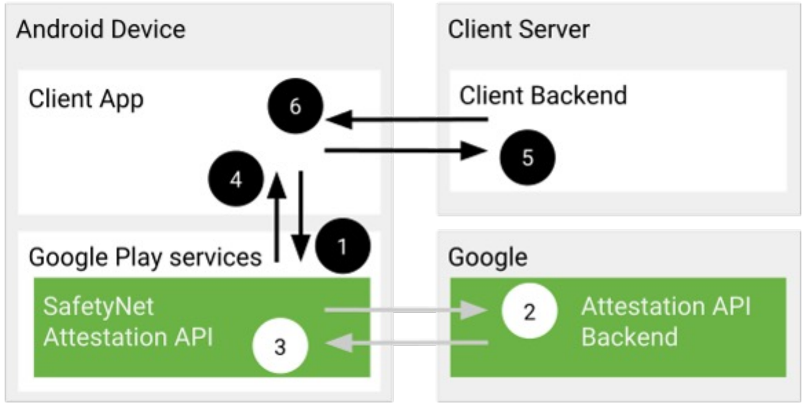
Source: Prünster et al.: Fides: Unleashing the Full Potential of Remote Attestation. ICETE 2019

SafetyNet Attestation API

- Google's (deprecated) official implementation of app attestation

Workflow:

1. App calls SafetyNet Attestation API with nonce
2. Snet Service checks environment
   - Requests attestation from Google servers
3. Google sends signed attestation to Snet Service
4. Snet Service returns result to app
5. App forwards signed attestation to backend server
6. Server validates response



Source: developer.android.com

SafetyNet Attestation API

SafetyNet was deprecated in 2022, probably due to problems in practice

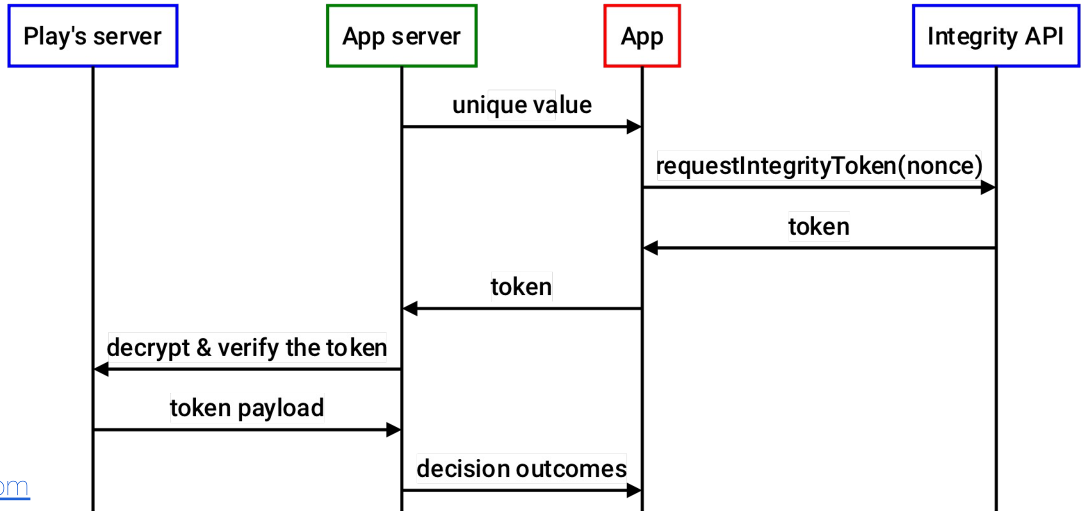
- Only used by small fraction of apps
  - 62 out of 163773 (0,04%) analysed apps in 2021
  - Requires Google API key
  - Requires server component
- Many apps didn't use the API properly
  - 32 (52%) validated the attestation locally (may simply be bypassed)
  - All others still did not correctly handle different error cases

Source: [Ibrahim et al.: SafetyNOT: On the usage of the SafetyNet Attestation API in Android. MobiSys 2021]

Google Play Protect

Official successor to SafetyNet Attestation API

- Attestation result is now an encrypted token
  - Backend may use Google server to decrypt and validate attestation
- Makes it harder to validate locally
  - Though still possible to obtain decryption key for local use



Source: [developer.android.com]

IAIK TU Graz

Permissions

Android Permissions

The Android OS controls access to certain resources through **Permissions**

- Identified by unique name
  - E.g. `android.permission.INTERNET`
- Developers specify needed permissions in AndroidManifest.xml
  - Some granted at install, others require runtime user consent
- Enforced at different levels
  - Kernel, e.g. `INTERNET` permission
  - Native service level, e.g. `READ_EXTERNAL_STORAGE` for SD card access

Defining A Permission

- Permissions are defined in the AndroidManifest.xml of the platform or an app

```
<!-- Used for runtime permissions related to contacts and profiles on this device. -->
<permission-group android:name="android.permission-group.CONTACTS"
    android:icon="@drawable/perm_group_contacts"
    android:label="@string/permgrouplab_contacts"
    android:description="@string/permgroupdesc_contacts"
    android:priority="100" />
<!-- Allows an application to read the user's contacts data.
<p>Protection level: dangerous -->
<permission android:name="android.permission.READ_CONTACTS"
    android:permissionGroup="android.permission-group.UNDEFINED"
    android:label="@string/permlab_readContacts"
    android:description="@string/permdesc_readContacts"
    android:protectionLevel="dangerous" />
```

- Every permission is assigned to a protectionLevel

Permission Protection Levels

Normal permissions

Automatically granted at install, no user confirmation needed

For ex.: `BLUETOOTH, CHANGE_NETWORK_STATE, INTERNET, NFC, INSTALL_SHORTCUT`

Dangerous permissions

Require explicit user approval at install or runtime

`CALENDAR, CAMERA, CONTACTS, LOCATION, MICROPHONE, PHONE, SENSORS, SMS, STORAGE`

These permissions are grouped, e.g. `PHONE = { READ_PHONE_STATE, CALL_PHONE, … }`

→ You always grant entire group, e.g. allow reading phone ID + making calls!

Special permissions

Require manual activation through system settings

`SYSTEM_ALERT_WINDOW, WRITE_SETTINGS, REQUEST_INSTALL_PACKAGES`

More Protection Levels

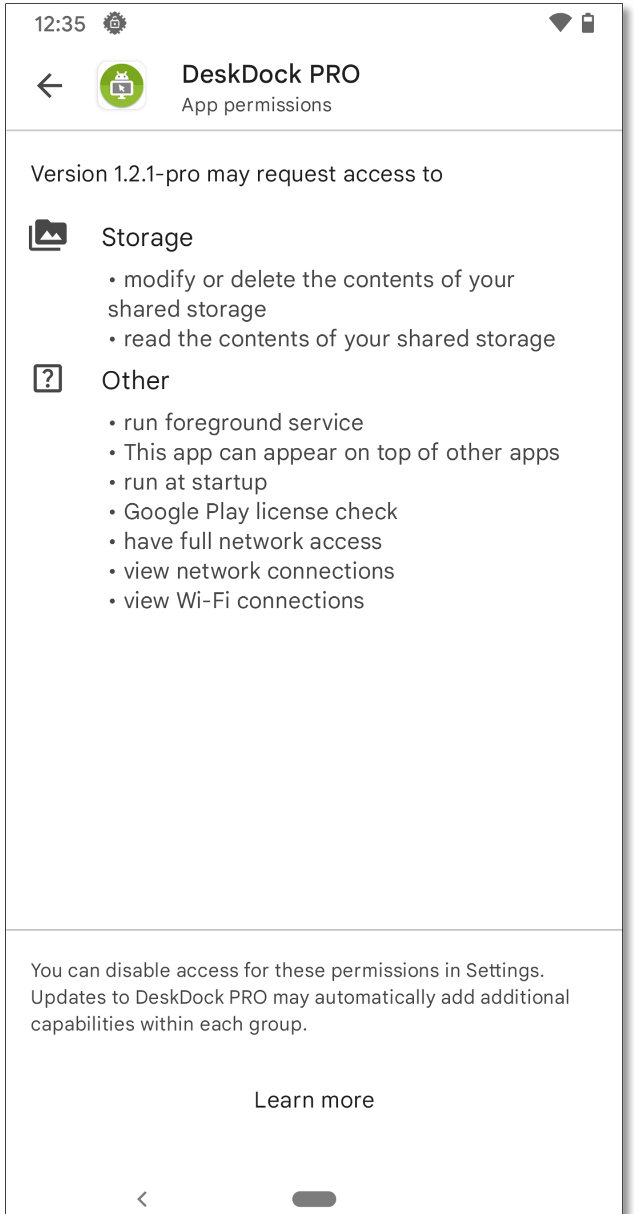
- **Signature**
  May only be granted to an app signed with the same key as the defining app
  - Used to define system-only permissions

- **Privileged**
  May only be granted to an app preinstalled to /system/priv-app

- **Development**
  May be granted to apps through the ADB shell

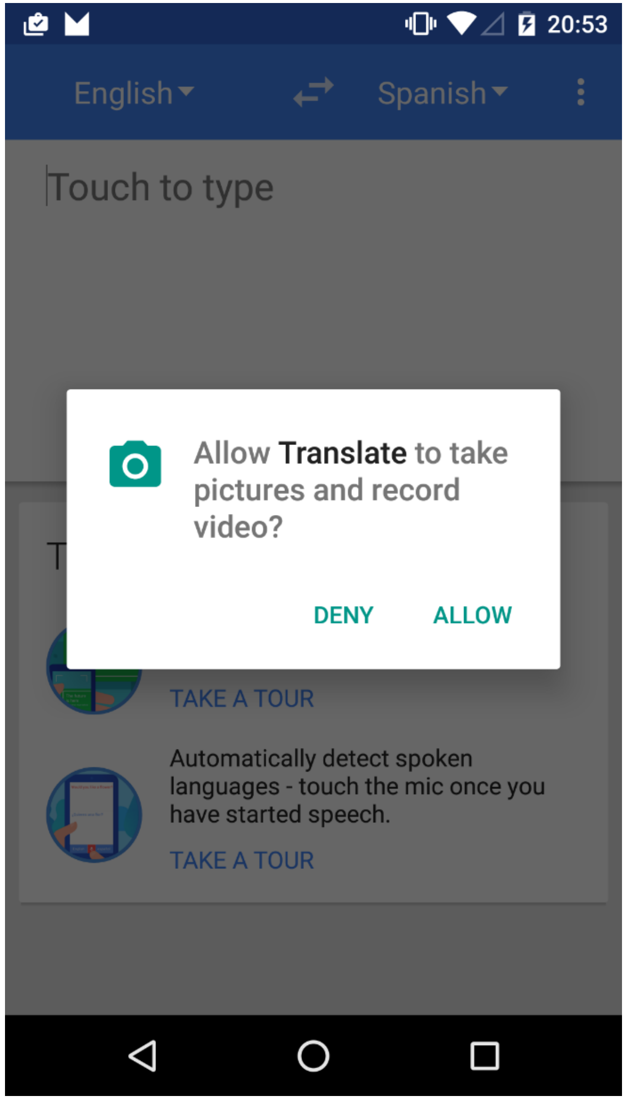
- Many more!

Normal Permissions



- Granted at install time
- Not even displayed to the user by default
  - Hidden away in Play Store app details
- No runtime checks required
- Once granted, cannot be revoked
- Fine-grained
- Granted for all users on device

Dangerous Permissions



- Need to be confirmed by the user at runtime
- Can be revoked by user at any time
  - Android 13: Revocation also by app
- Granted / revoked with entire group
  - Accept „PHONE“ → Grant reading phone ID + calling
- Managed individually per app and user

Custom Permissions

- Applications can define custom permissions
- Can be used for protecting access to app components
  - `ContentProviders`, `Services`
- Developers can specify protection level
  - **Signature:** Grant at install time only to apps signed with same certificate as the app that defined the permission
  - **Dangerous:** Show a dialog at runtime

Custom Permission Vulnerabilities (2021)

Stealthily obtain dangerous **system** permissions by misusing custom permissions

1. Install App A that defines a normal custom permission
2. Install App B that uses this custom permission
3. Uninstall App A and reinstall updated version

   Redefines custom permission as dangerous, assigns it to known permission group

```
<permission android:name="com.test.cp"
    android:protectionLevel="dangerous"
    android:permissionGroup="android.permission-group.PHONE"/>
```

4. App B now holds any permission in group `android.permission-group.PHONE`
   - Can now initiate phone calls (system permission `CALL_PHONE` is in `PHONE` group)
   - User was never asked

Source: Li et al.: [Android Custom Permissions Demystified: From Privilege Escalation to Design Shortcomings]

Outlook

- 17.04.2024
  - Android Application Security II

- 24.04.2024
  - iOS Platform Security