

# Android Application Security I

*Mobile Security 2026*

Florian Draschbacher  
[florian.draschbacher@tugraz.at](mailto:florian.draschbacher@tugraz.at)

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

# Reminder: Practicals

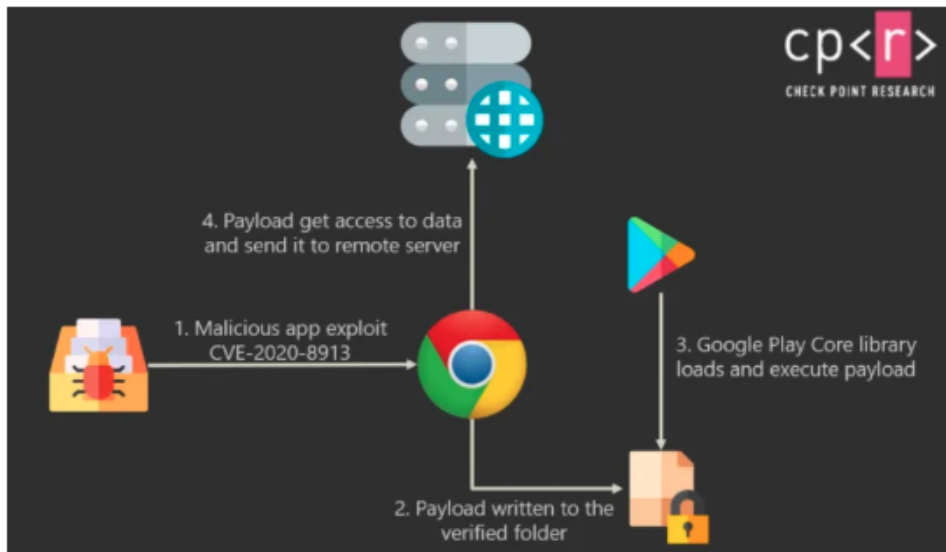
- Assignment 1 deadline: **Today, 23:59!**
  - Q&A in the practicals time slot

# 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](https://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

🛡 SECURITY IS AN EXPENSIVE HASSLE

# Samsung's Android app-signing key has leaked, is being used to sign malware

The cryptographic key proves an update is legit, assuming your OEM doesn't lose it.

RON AMADED – DEC 2, 2022 10:13 PM 146



Source: [arstechnica.com](https://arstechnica.com)

## What?

Attackers stole platform signing keys

## Details?

- Google discovered malware signed with platform keys
- Hackers had compromised them for multiple vendors
  - Samsung, LG, Mediatek, ...
- Apps signed using these keys ran as system user

## Result

- Malware ran with system privileges
- Could use virtually all permissions

# Introduction

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

- Application security teased 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

2021/2023 Whitepaper by Android Security Engineers Source: <https://dl.acm.org/doi/pdf/10.1145/3448609>

- Sensitive operations must be allowed by all 3 stakeholders
  - User
  - Developer / Application
  - Platform
- **Sensitive operation:** E.g. Sharing data from app A to app B
  - **App A developer consent:** Offering share functionality in app
  - **User consent:** Selecting target app in share dialog
  - **Platform consent:** Arbitrating and limiting access to specific data
  - **App B developer consent:** Accepting shared data

# 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

Google  
Play

Unknown  
Sources  
Warning

Install  
Confirmation

Verify Apps  
Consent

Verify Apps  
Warning

Runtime Security  
Checks

Sandbox &  
Permissions

**And soon:**

- Developer verification
- Advanced sideloading flow

# Android Applications

# Android App Development

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

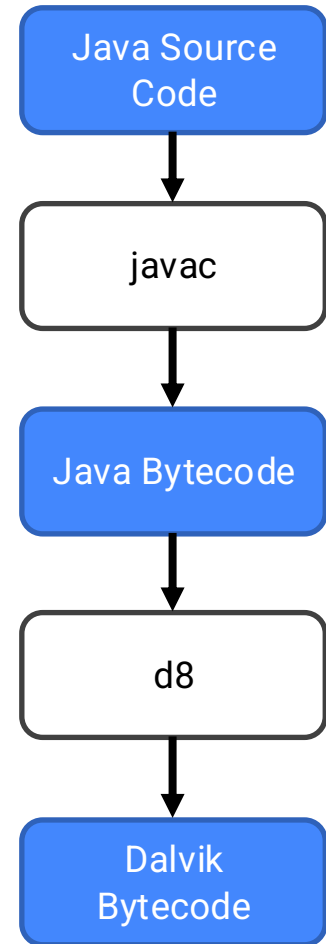
\* Or other languages that compile to Java Bytecode (such as Kotlin)

## 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+**

# 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!
  - Allows loading & executing any other app'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:** Real 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 may be modified without breaking signature

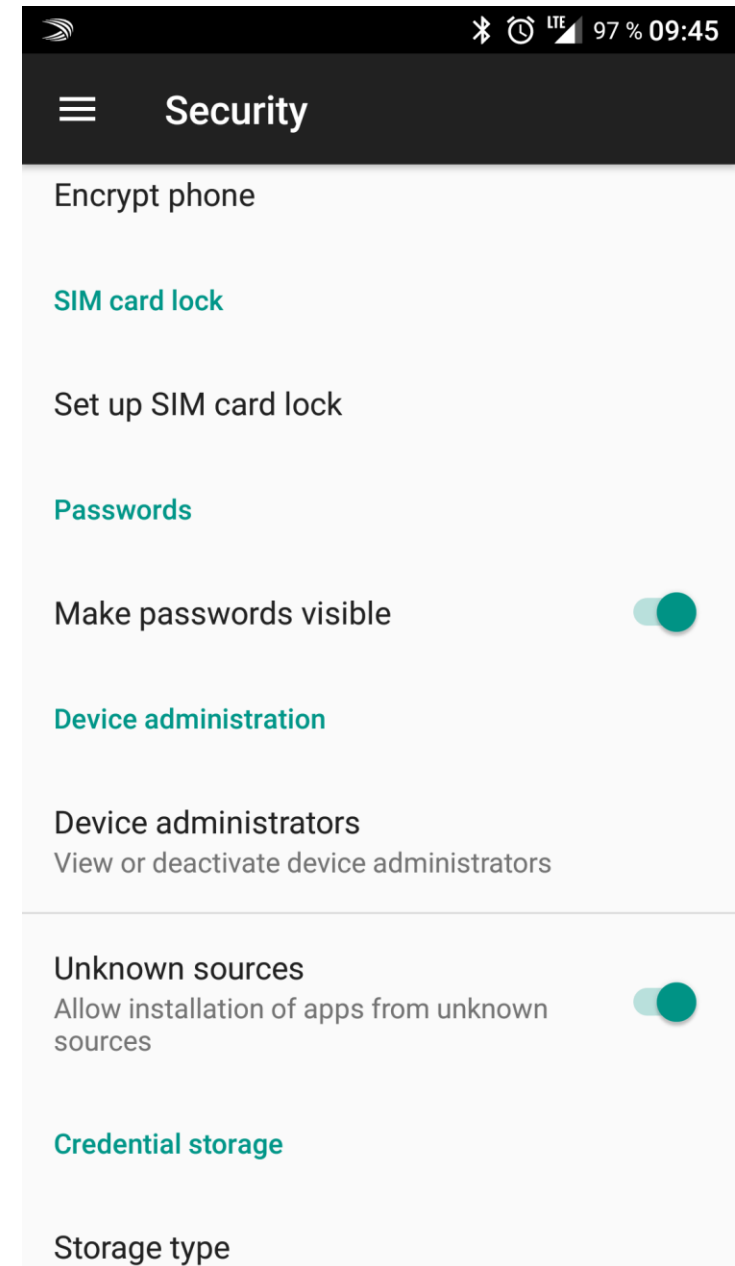
- Signature scheme v1 only signs file entries in the ZIP
- DEX code can be embedded in the ZIP file
  - ZIP file: Trailer at end points to file entries
  - DEX file: Header at start points to following data chunks
  - So: A file can be a valid DEX file and ZIP file at the same time!
- Android runtime supports running APK or DEX files
  - File type confusion can be exploited

# App Distribution

# App Sources

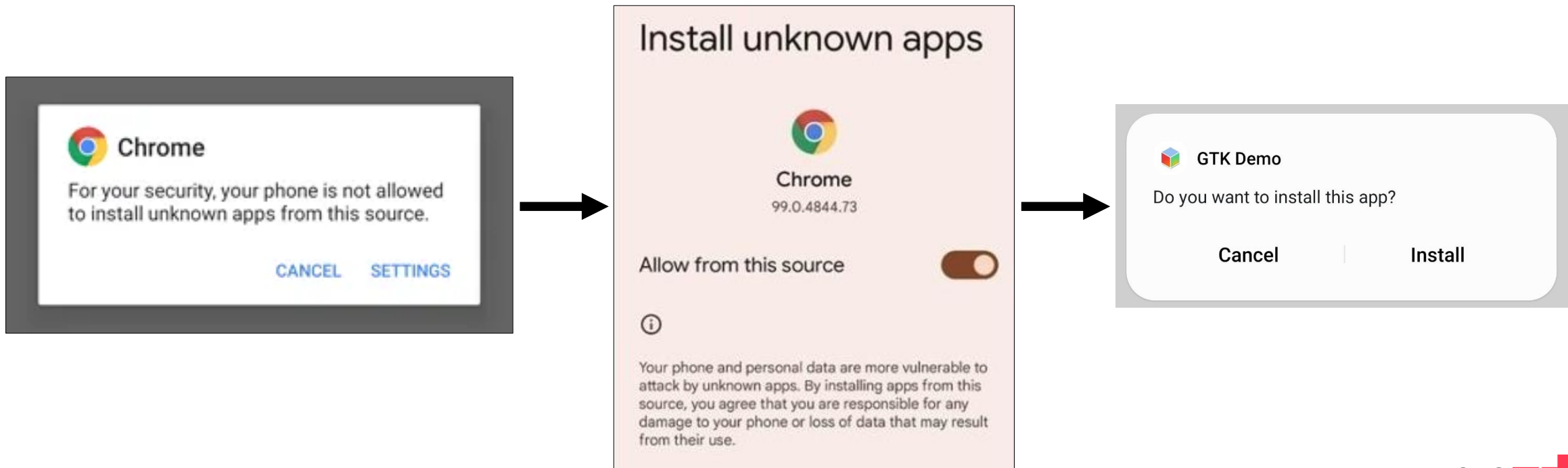
## Android allows installation of apps from

- Google Play
  - Trusted by default
  - Requires license from Google
- Third-party app stores
  - F-Droid, Samsung Galaxy Store, Huawei AppGallery, ....
  - 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



# App Sideloading

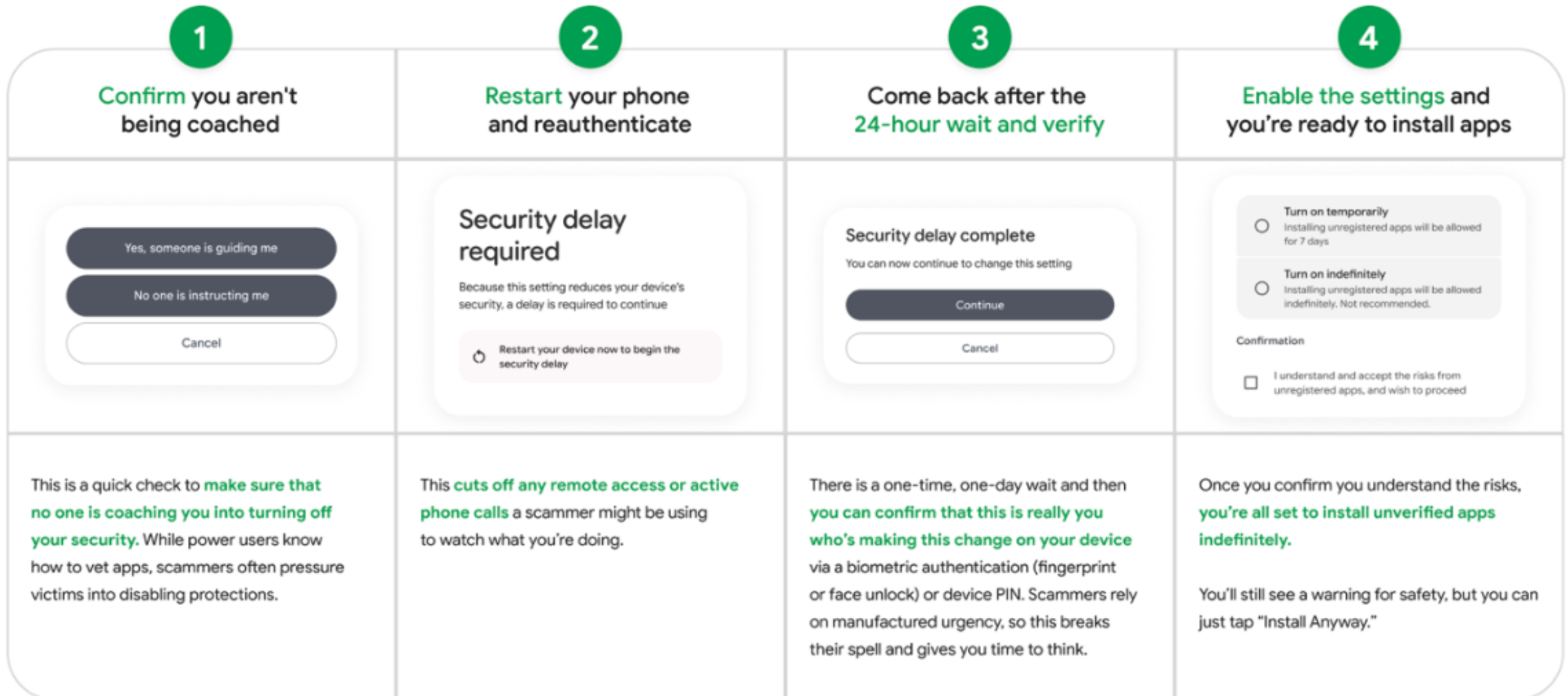
- Currently, it is possible to install APK files from arbitrary sources
  - Apps can request APK installation through API
  - Users need to approve API use per invoking app
  - Additional confirmation needed per-APK



# Developer Verification: 3 Options (coming soon)

- Full distribution
  - Paid account, identity verification, Register package names
  - Unlimited apps and users, via Google Play or other channels
- Limited distribution
  - Free account, (likely) identity verification, Register package names
  - Apps can be installed on up to 20 devices, secure handshake (QR code or link)
- Unverified developers
  - No account or registration needed
  - Advanced sideloading flow for installing apps from unverified developers

# Advanced Sideloading Flow



# 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 😊

# 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
  - ...

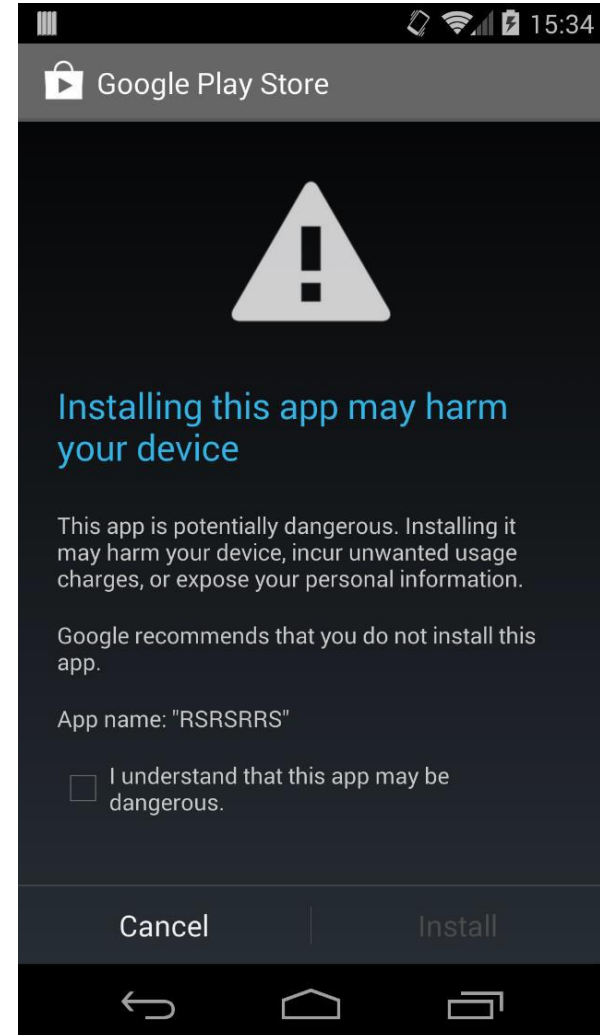
**Conclusion:** Automated app analysis is never perfect!

# Verify Apps (2012)

## App scans extend to user side

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

Can be disabled by user!

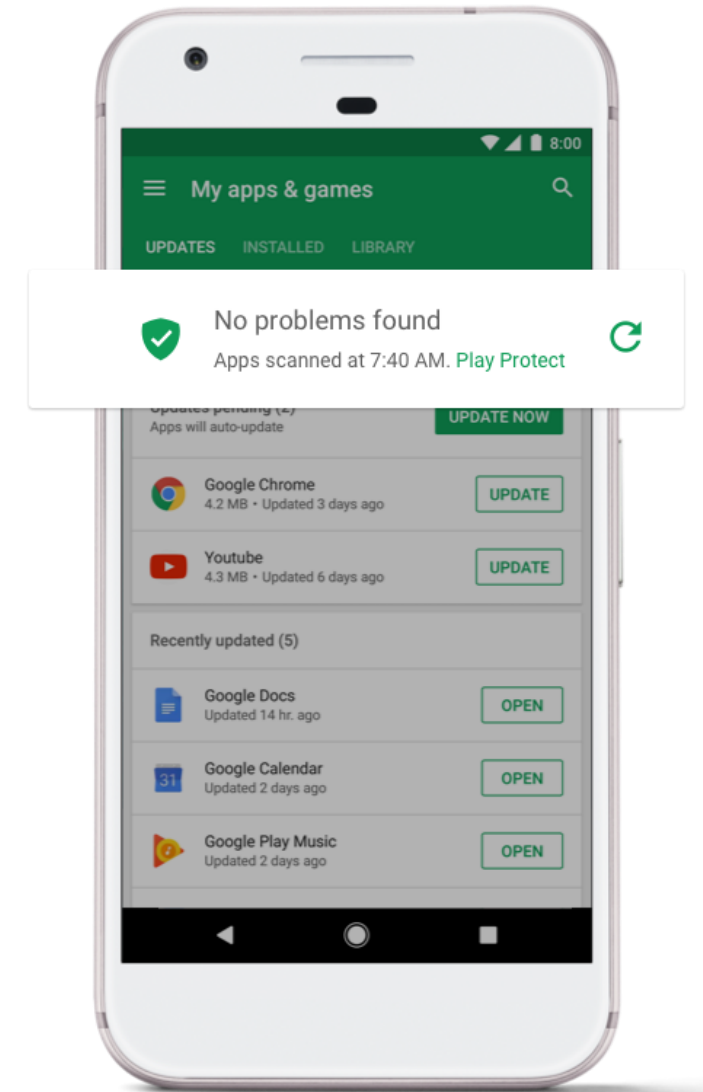


# Verify Apps (2014 – 2017)

- 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

# 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



Can still be disabled by user!

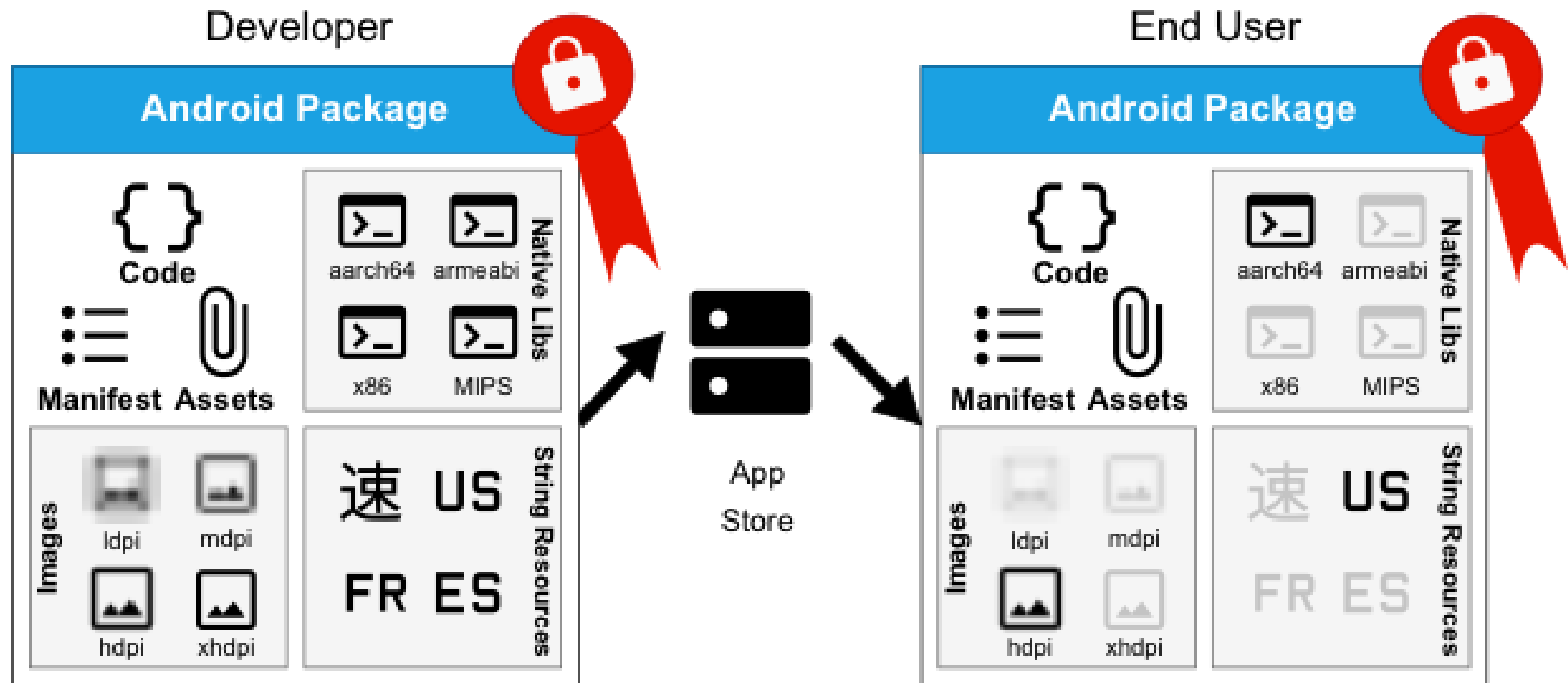
# Pirated Applications

- (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)

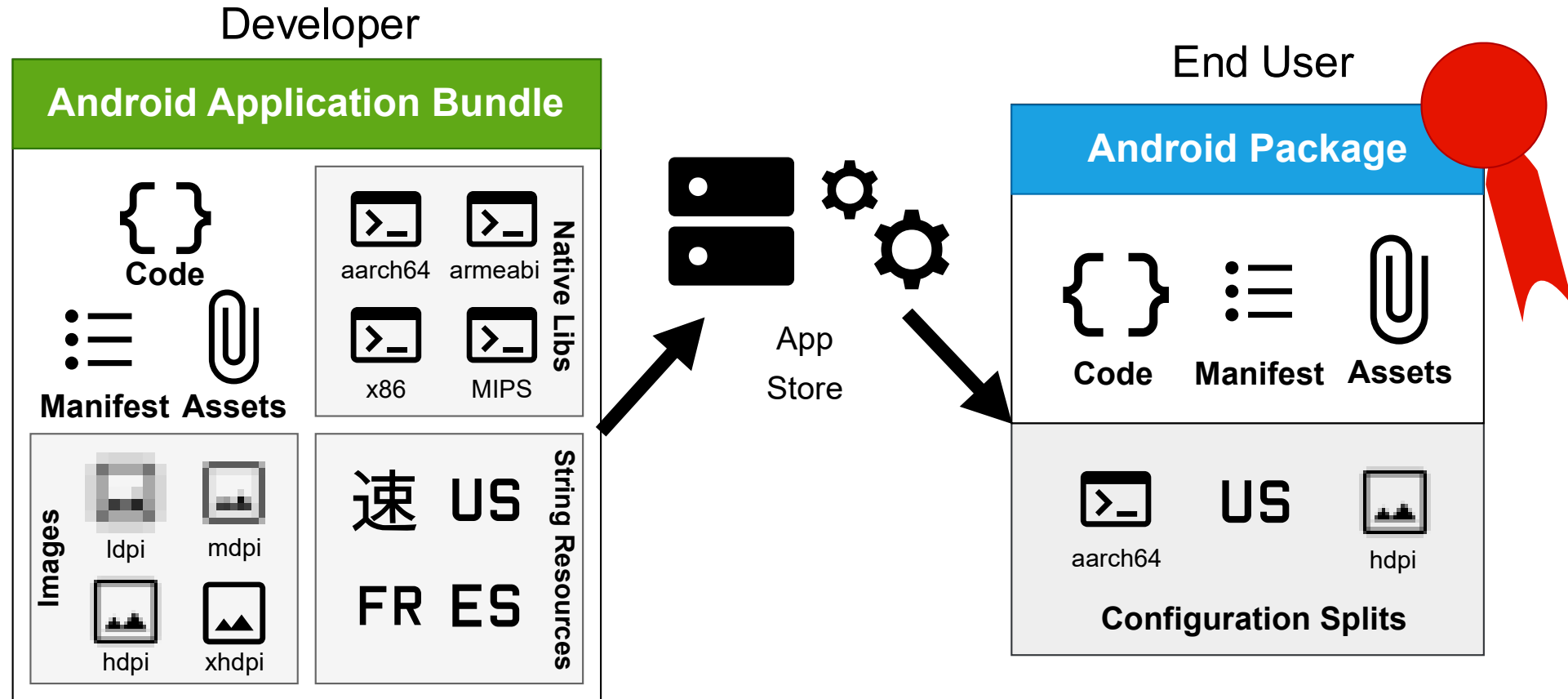
- For installation on an Android device, an app needs to be packaged as APK file
- Developers used to submit apps to Google Play as signed APKs
- **Problem:** Universal APKs contain resources needed only for other devices
  - Example: x86 native libraries wastes space on ARM device
- **Solution:** Android 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

# < 2018: Direct APK Distribution



 Key exclusively held by developer

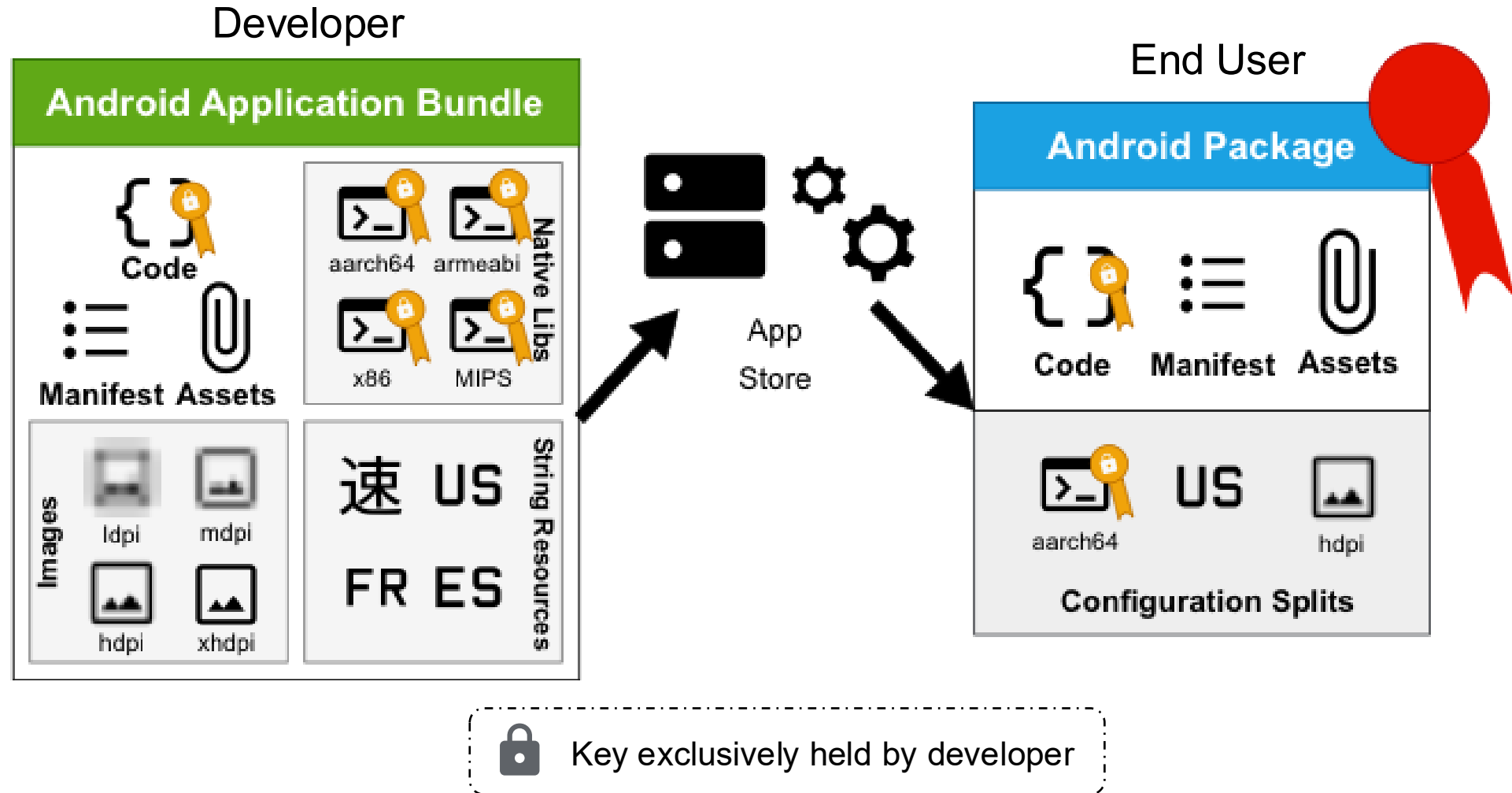
# 2018: Android Application Bundle



# Code Transparency

- Problem:
  - APK Signature is fundamental to Android Platform Security Model
  - With AAB, the app store operator 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
- Still far from perfect
  - Optional, so can simply be stripped
  - Android Manifest and Resources not covered → Changing execution flow possible
  - No infrastructure in OS, users needs to manually check

# 2021: Code Transparency for AAB



# Reverse-Engineering & Analysis

# Recall: APK Structure

The actual file format for installing Android apps

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

# Recall: APK Structure

The actual file format for installing Android apps

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

# 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 entirely reversible (some needed information lost through compilation)
  - Easier to analyse

# 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 \$\$\$)
  - Unidbg (Open Source; Allows emulating native libraries on desktop OSs)

# 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 voodoo required
- **Xposed Framework**: Embed manipulation primitives in Zygote process
  - Make every app process (forked from Zygote) load Xposed modules
  - No longer maintained, but the technique lives on in LSposed.
- **Frida**: Either inject into running process (root) or into APK file
  - Dynamically manipulate app through Javascript console

# App Attestation

# Application Repackaging

- APK 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
    - To users who had not installed the app before
    - 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

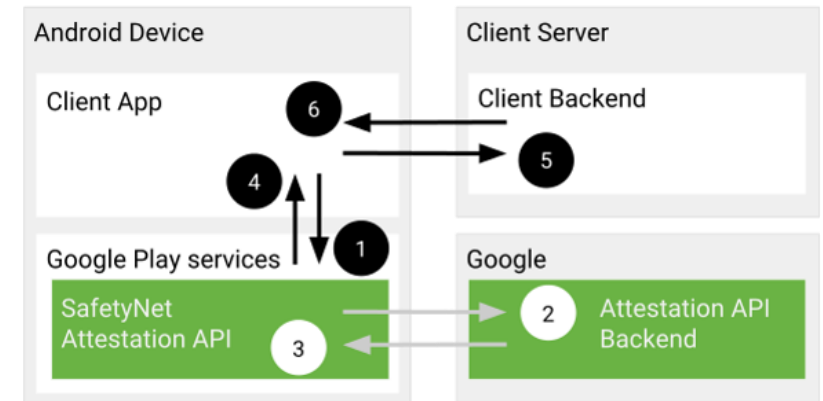
1. Backend server generates random challenge and sends it to app
2. App requests TEE to generate signed attestation including
  - Random Challenge
  - APK signature
  - Verified Boot state
3. App forwards attestation to backend server
4. Server aborts communication if attestation invalid

# 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](https://developer.android.com)

# SafetyNet Attestation API

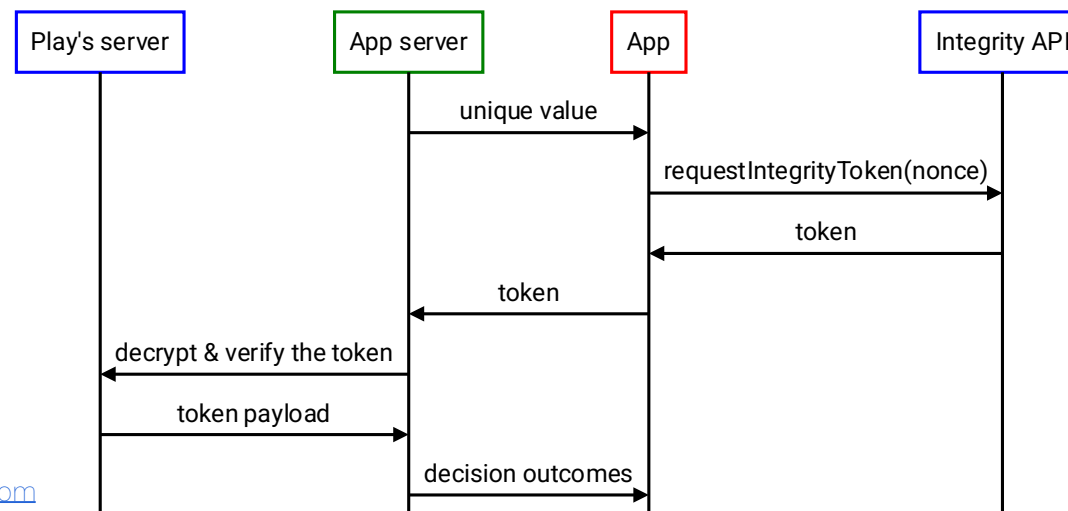
**SafetyNet was deprecated in 2022**, probably due to problems in practice

- 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

# 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](https://developer.android.com)

# 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
  - System service level, e.g. `ACCESS_FINE_LOCATION` for GPS location

# Consuming A Permission

Permission *use* is declared in the AndroidManifest.xml of an app

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools">

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

  <application>
    <activity
      android:name="com.isec.test.MainActivity"
      android:exported="true">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

# Defining A Permission

Permissions are also *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

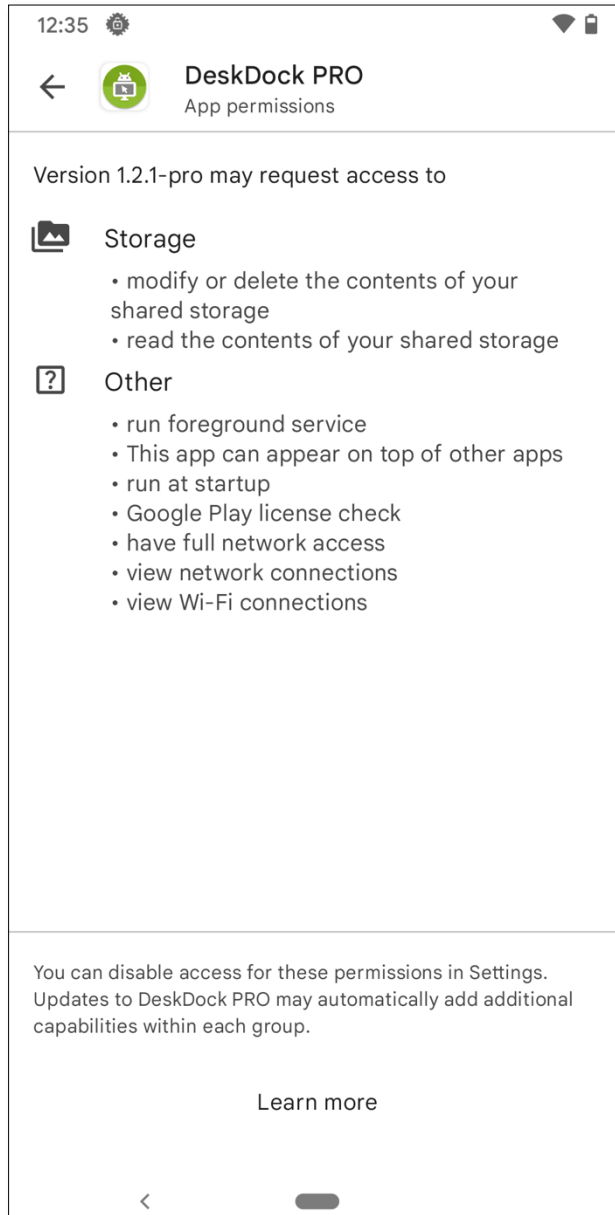
Require manual activation through system settings

SYSTEM\_ALERT\_WINDOW, WRITE\_SETTINGS, REQUEST\_INSTALL\_PACKAGES

# More Protection Levels

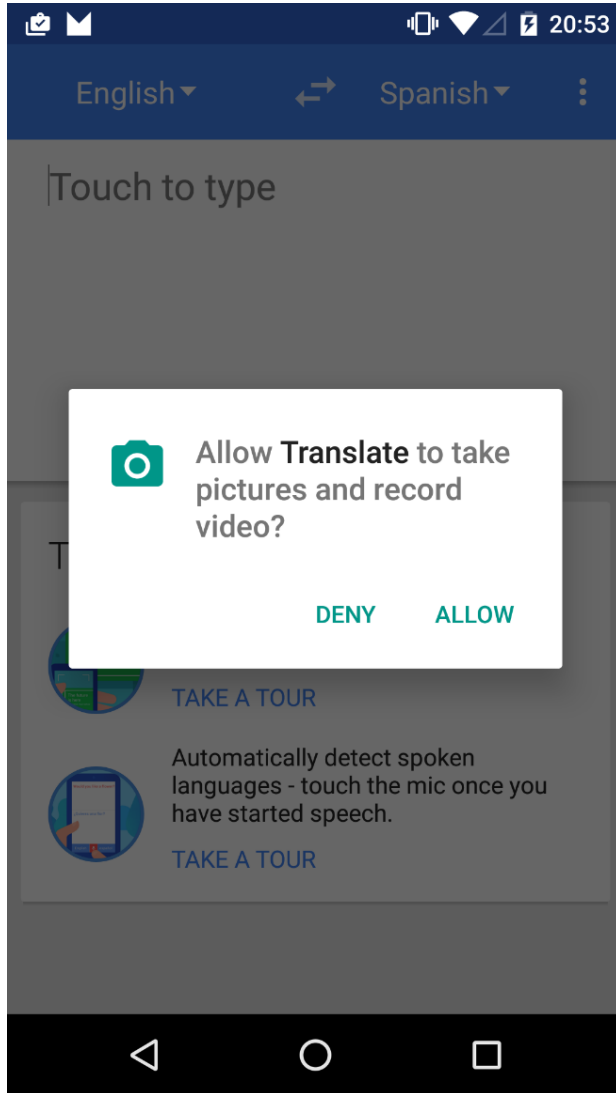
- **Signature**  
May only be granted to an app signed with the same key as the defining app
  - Used to define system-only permissions
- **Privileged**  
May only be granted to an app preinstalled to `/system/priv-app`
- **Development**  
May be granted to apps through the ADB shell
- Many more!

# 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
  - Managed individually per app and user
- Can be revoked by user at any time
  - Android 11: Auto-revocation for perms of unused apps
  - Android 13: App can revoke permission via API
- Granted / revoked with entire group
  - Accept „PHONE“ → Grant reading phone ID + calling
- Advanced options for some permissions
  - Only allow once, only allow in foreground, ...

# 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

# Outlook

- 17.04.2026
  - Android Application Security II
  
- 24.04.2026
  - iOS Platform Security