IAIK TU Graz

Android Application Security

Mobile Security 2021

Johannes Feichtner
johannes.feichtner@iaik.tugraz.at

Outline

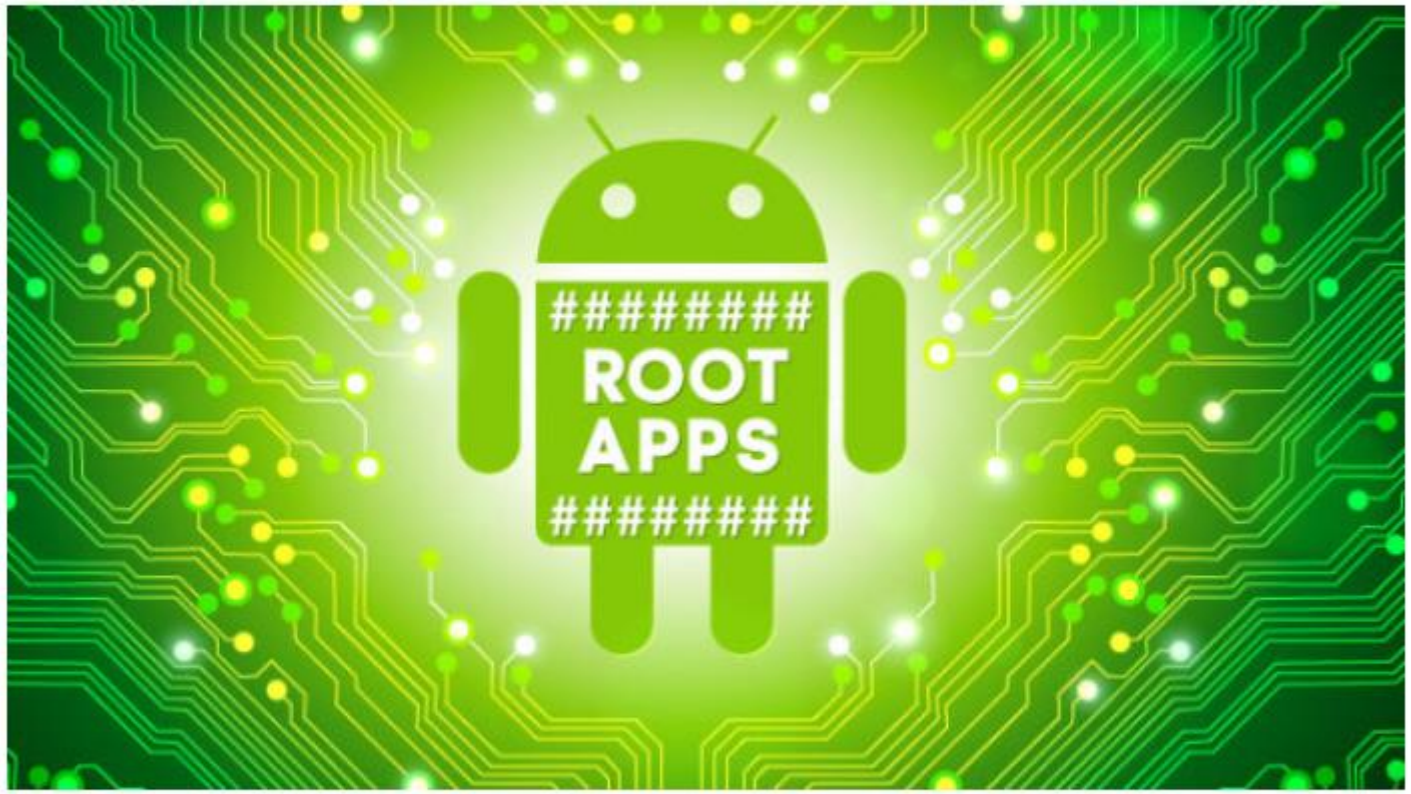
- What happens on app installation?
- What is an Android app actually?
- Permissions?

New type of auto-rooting Android adware is nearly impossible to remove

20,000 samples found impersonating apps from Twitter, Facebook, and others.

by **Dan Goodin** - Nov 4, 2015 11:15pm CET

130



UCR Today

Researchers have uncovered a new type of Android adware that's virtually impossible to uninstall. The adware exposes phones to potentially dangerous root exploits and masquerades as one of thousands of different apps from providers such as Twitter, Facebook, and even Okta, a two-factor authentication service.

Source: http://goo.gl/bRWWGw

What?

20.000 trojanized apps with various local root exploits: Memexploit, Framaroot, ExynosAbuse

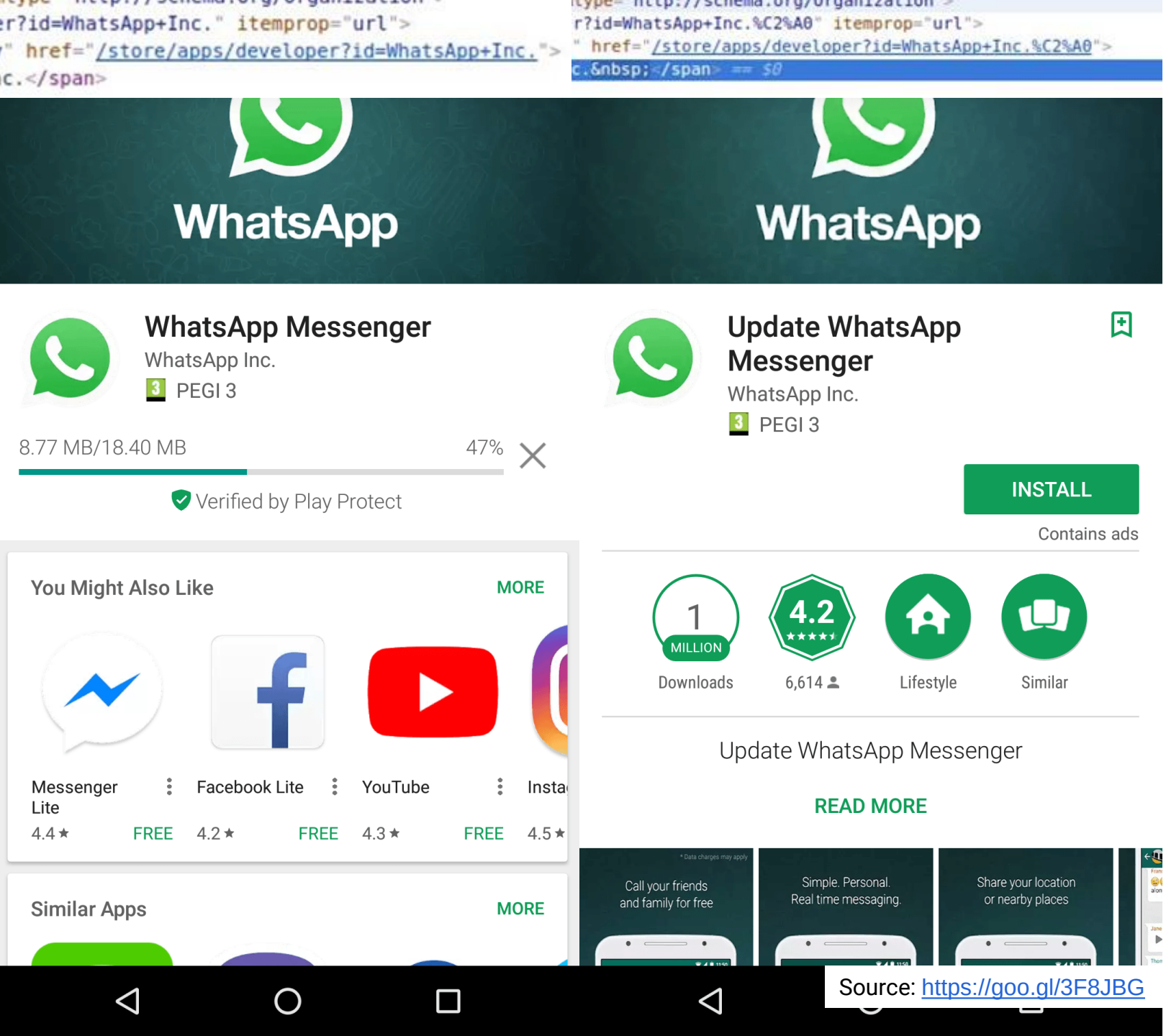
How?

- Repackaged > 1000 popular apps
- Distributed on 3rd party markets

Result

System applications with root

→ Super-permissions to break out of sandbox



Source: https://goo.gl/3F8JBG

What?

PlayStore listed fake WhatsApp Messenger

How?

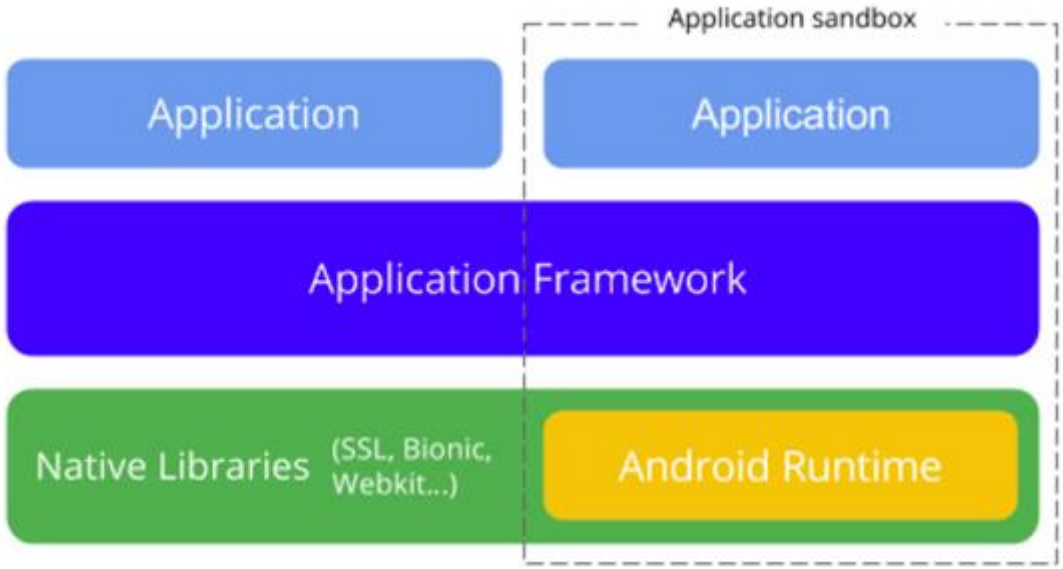
- Author added non-visible Unicode character to vendor name
- 1 to 5 Mio. downloads

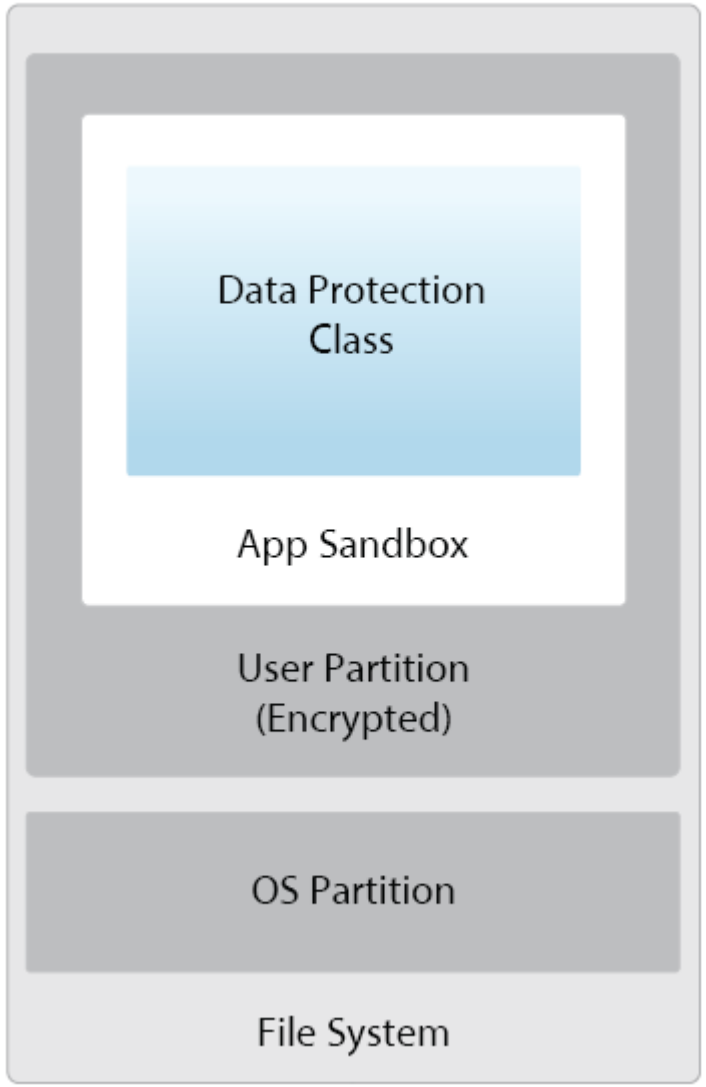
Problem

- Ad-loaded wrapper app to download `whatsapp.apk`
- Barely visible in app list: blank icon, no text

IAIK TU Graz

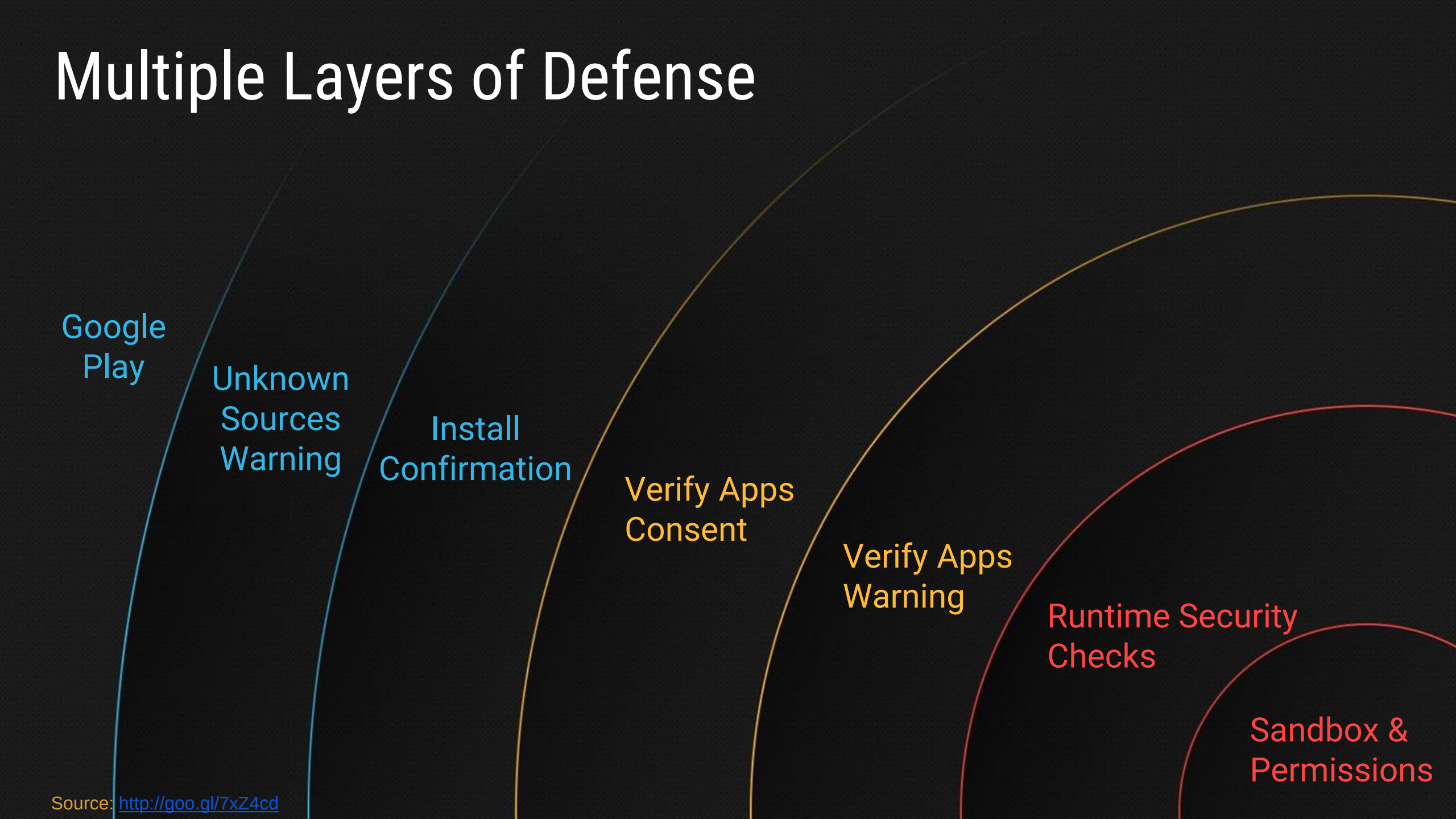
Application Security





Android Application Security

Multiple Layers of Defense



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

App Installation Process

1. Google Play or „Unknown Sources“ warning (requiring user confirmation)
2. Install confirmation shows user requested permissions
3. Verify app: check against DB of malware before installation since Android 4.2
   →Can be disabled by user!
4. Application sandbox and runtime checks

Weakest link in chain: **The user!**

Note: Google‘s defense layer protects Android, not your data!

Google PlayStore

- 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
  - Must not deploy app elsewhere → „non-compete clause“ in Distribution Agreement

Security mechanisms

- Control instrument for app distribution (review, stop dist., remove app)
- Google Bouncer: In-house malware detection system
- Applications have to be self-signed
  - No modified app can be installed or updated

Google Bouncer

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 bugs
   - Malware signatures, heuristics, similarities, source / developer, third-party reports
   - If flagged malicious → Manual analysis by human being
   - If deemed malicious → Goodbye Google account ☺

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
  - E.g. app with call to `/system/bin/ls` never executed dynamically

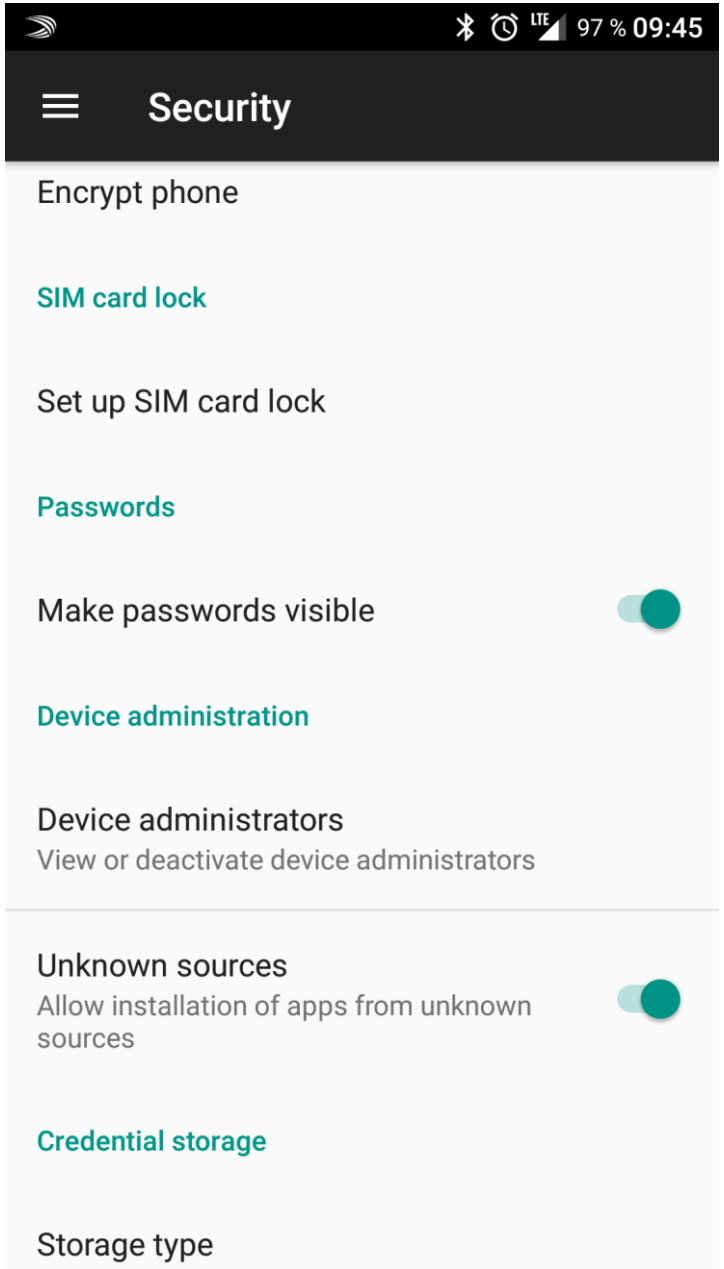
- Detect emulation through API calls (http://goo.gl/eAPIHz)
  - `TelephonyManager.getDeviceId() == 0` → emulator!
  - `Build.HARDWARE == „goldfish“` → emulator!

Conclusion: Dynamic app analysis is never perfect!

Unknown Sources

First visible layer of defense on device

- By default, no apps from 3rd party stores
  - Amazon, F-Droid, Samsung
  - Security checks?
- From file system
  - If app available as .apk file
  - Can be downloaded from anywhere



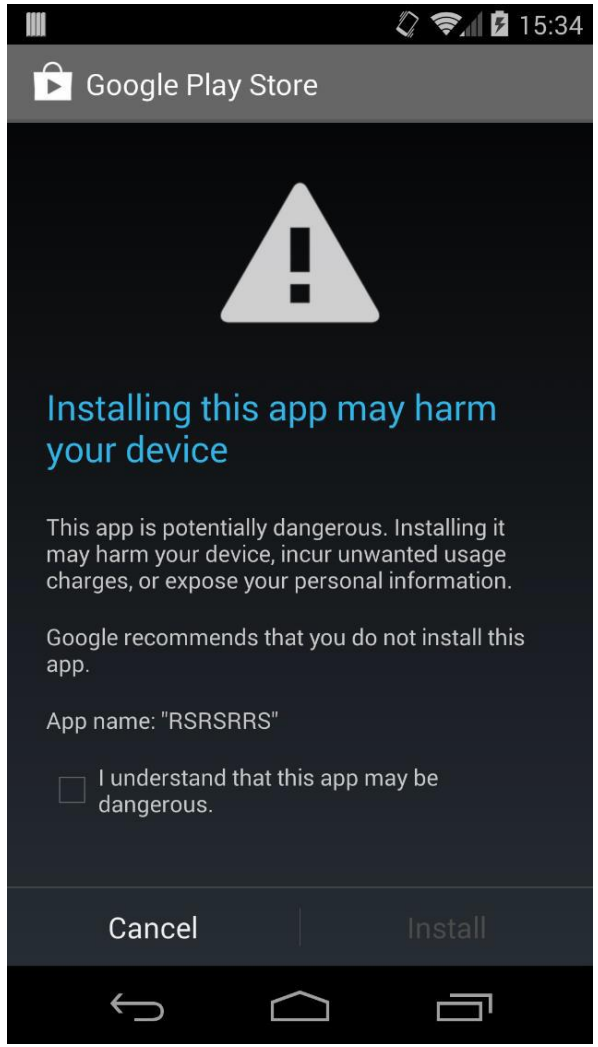
Verify Apps

Can be disabled by user!

Second visible layer…

- Apps are verified / categorized prior to install
  - Remote database with malware signatures
  - Verification agents
    - With Google Play: Since Android 2.3
    - For others: Since android 4.2

- Warn or block potentially harmful apps
  - Backdoors
  - Fraudware
  - Hostile downloaders
  - Phishing apps
  - Privilege Escalation apps
  - Rooting apps
  - Spyware
  - Trojans / Trojanized apps



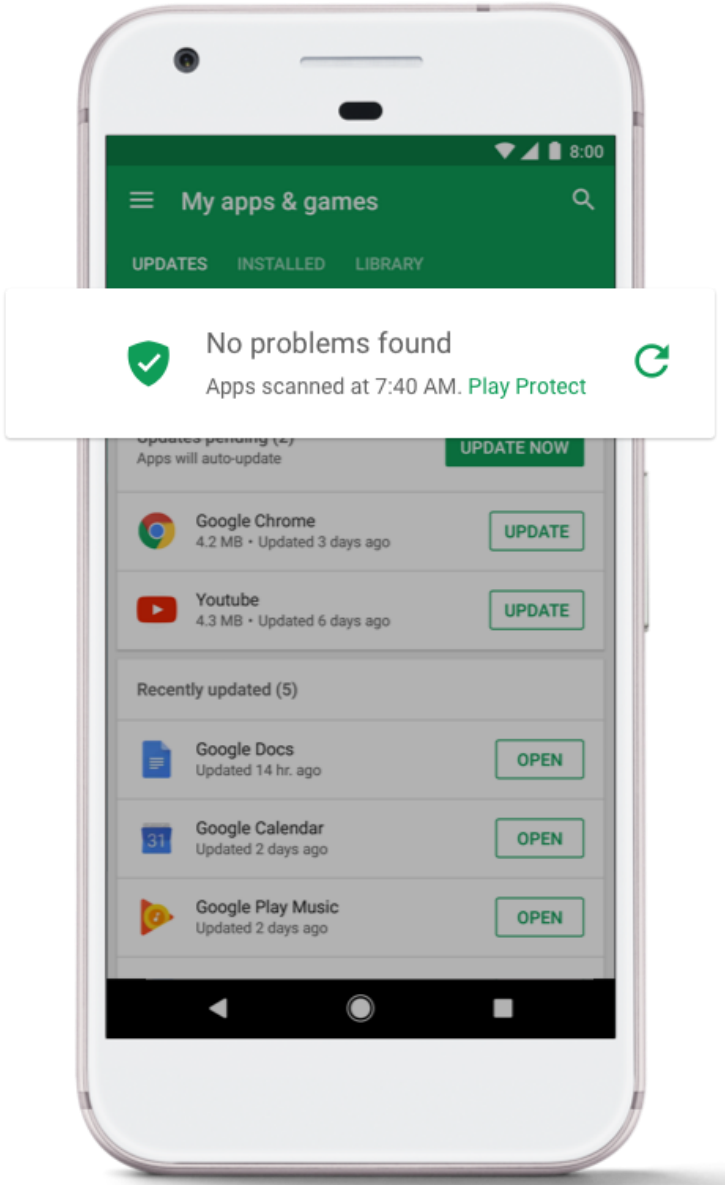
Google Play Protect

Extended verification since Android 8

- Malware Scanning
  - PlayStore service scans and reports apps on device
  - Now also for unknown / side-loaded apps

- SafetyNet Verify Apps („Attestation“) API
  - „Let developers understand if a device is tampered“
  - App can request to be run in certain environment, e.g. not-rooted, custom ROM, API hooking, etc.
    - Send compatibility check request to Attestation API

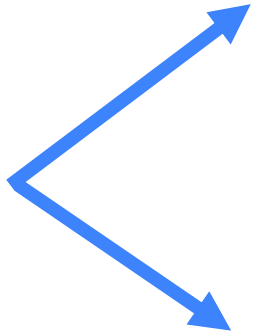
  - Can refuse to run if known „bad“ app or setting is found



IAIK TU Graz

Android App Structure

```
com.example.app.apk
 - assets/
 - AndroidManifest.xml
 - classes.dex
 - resources.arsc
 - lib/
    - armeabi-v7a/
        - libapp.so
 - META-INF/
    - CERT.RSA
    - CERT.SF
    - MANIFEST.MF
 - res/
    - drawable/
    - layout/
    - xml/
```



Code and resources (common)

```
/data/app/com.example.app/
 - lib/arm/libapp.so
 - oat/arm/base.odex
 - base.apk
```

Data (per user)

```
/data/user/0/com.example.app/
 - files/
 - databases/
 - shared_prefs/

/data/user/1/com.example.app/
 - …
```

Android App Structure

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
`META-INF/`	
`MANIFEST.MF`	Enumeration of all files in app package + SHA-1 checksums
`CERT.SF`	Signature file. Digest of manifest file + individual digests per app file
`CERT.RSA`	Digital signature over CERT.SF + developer‘s signing certificate
`res/`	App resources, e.g. GUI layouts in XML format, graphics, colors, …
`resources.arsc`	Resource meta data (binary format). Listing of all uses resources

Package Directories

```
# ls -l /data/user/0/

drwxr-x--x bluetooth bluetooth com.android.bluetooth
drwxr-x--x system    system    com.android.keychain
drwxr-x--x u0_a4     u0_a4u    com.android.providers.calendar
drwxr-x--x system    system    com.android.providers.settings
drwxr-x--x radio     radio     com.android.providers.telephony
drwxr-x--x u0_a5     u0_a5u    com.android.providers.userdictionary
drwxr-x--x u0_a27    u0_a27u   com.android.proxyhandler
```

- Updating system apps → `/system` partition usually not writable!
  - `/system/app/` → `/data/app/`

- "Forward locking" = copy protection of apps. Default: world-readable .apk files
  - World-readable resources (`/data/app/`) and code separate (`/data/app-private/`)
  - Mainly for paid apps (DRM)

Android Permissions

Permission = Ability to perform particular operation

- Assignment
  - Typically at install time (`AndroidManifest.xml`)

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

  - Also at runtime since Android 6.0

- Enforced at different levels
  - Kernel, e.g. `INTERNET` permission
  - Native service level, e.g. `READ_EXTERNAL_STORAGE` for SD card access
  - Framework level
    - Dynamic: Check for permission in app while executing
    - Static: Intents, Content Providers

Permissions Groups

Normal permissions

Automatically granted, no user confirmation needed

For ex.: `BLUETOOTH`, `CHANGE_NETWORK_STATE`, `DISABLE_KEYGUARD`, `FLASHLIGHT`, `INTERNET`, `NFC`, `USE_FINGERPRINT`, `SET_ALARM`, `INSTALL_SHORTCUT`, `VIBRATE`

Dangerous permissions

Require explicit user approval at install or runtime

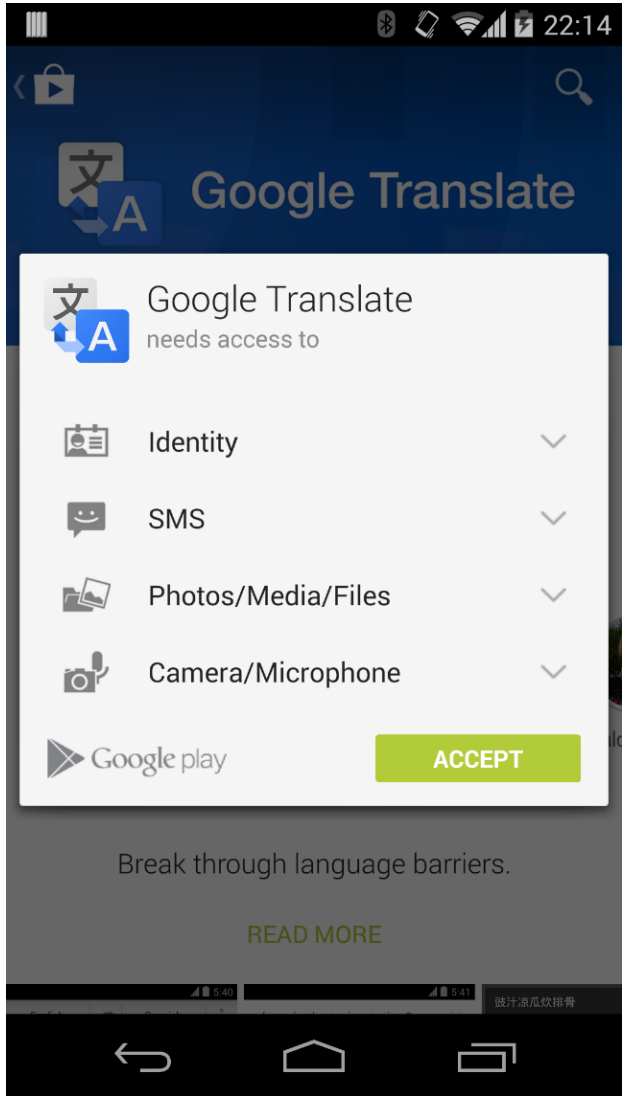
`CALENDAR`, `CAMERA`, `CONTACTS`, `LOCATION`, `MICROPHONE`, `PHONE`, `SENSORS`, `SMS`, `STORAGE`

Problem due to grouping

E.g. `PHONE = { READ_PHONE_STATE, CALL_PHONE, … }`

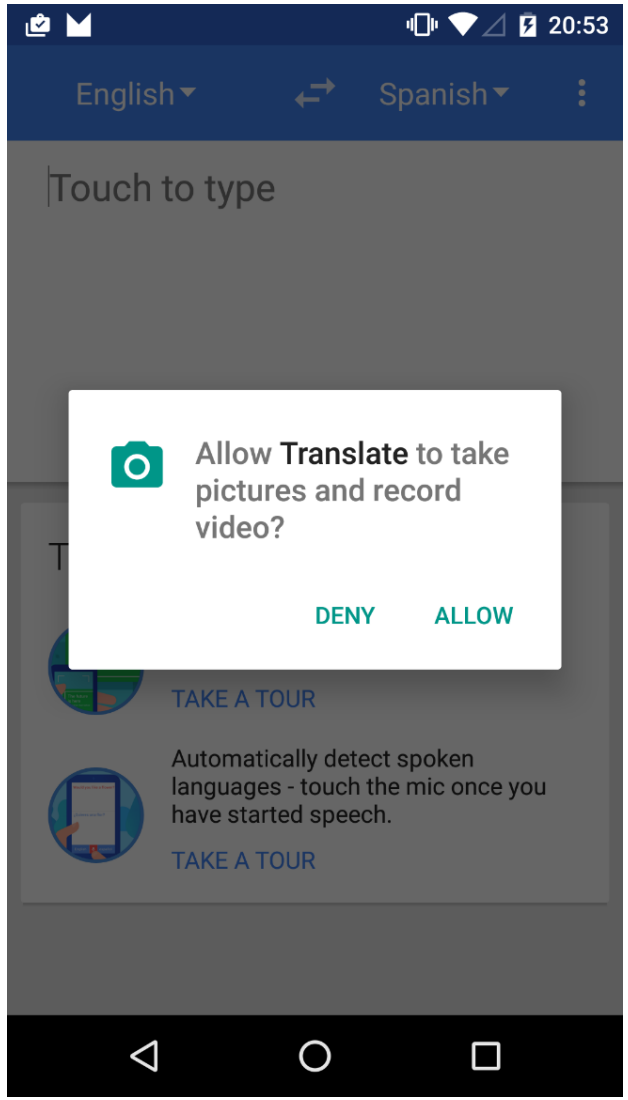
→ You always grant entire group, e.g. allow reading phone ID **+** making calls!

Install-Time Permissions



- All permissions granted at install time
- With Android 4.2+
  Only dangerous permissions require confirmation
- No runtime checks required
- Once granted, cannot be revoked
- Fine-grained
- Granted for all users on device

Runtime Permissions



- Need to prompt for dangerous permissions
- Can be revoked at any time
- Granted / revoked with entire group
  - Accept „PHONE“ → Grant reading phone ID + calling
- Managed individually per app and user
- Managable by device owner
  - Useful for MDM

Application Signing

For all .apk files

- Self-signed X.509 certificate
- Not using PKI → no certificate chain of trust!
- Individual signature for each file included in APK

  →Attacker cannot simply exchange file in app package!
- Signing certificate == Package & developer identity
- Package update requires same certificate

For update packages (OTAs)

- Modified ZIP format
- Signature in ZIP comment over whole file
- Verified by OS and recovery

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!
- Using various class loaders (APK, JAR, pure dex files, optimized dex files)
- By loading & executing any other application‘s code (`createPackageContext` API)

Problems

- Malicious app can evade detection by Google Bouncer & app analysis
  - Some remedy provided with Google PlayProtect (since Android 8)
- 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
  - 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 application analysis significantly!

Case Study: Lipizzan

- Developed by „Equus Technologies“ (Israel)
- 20 apps in Play Store, installed on 100 devices
  - „Backup, Cleaner, Recorder, Notepad, …“

Source: https://goo.gl/K7Ea3a

Two-stage approach

1. Clean app in PlayStore (stage 1)
   - After install, „License Verification“ loads stage 2
   - Check device properties (platform, version, etc.) and abort criteria
2. If all clear, stage 2 uses root exploits to gain system permissions

Result

Attacker has full control over device and sensors via C&C servers

Case Study: WhatsApp

Outlook

- 06.05.2021
  - Static and Dynamic Application Analysis

- 20.05.2021
  - Mobile Network Security