IAIK TU Graz

Android Application Security

Mobile Security 2022

Florian Draschbacher
florian.draschbacher@iaik.tugraz.at

Slides based on those by **Johannes Feichtner**

Outline

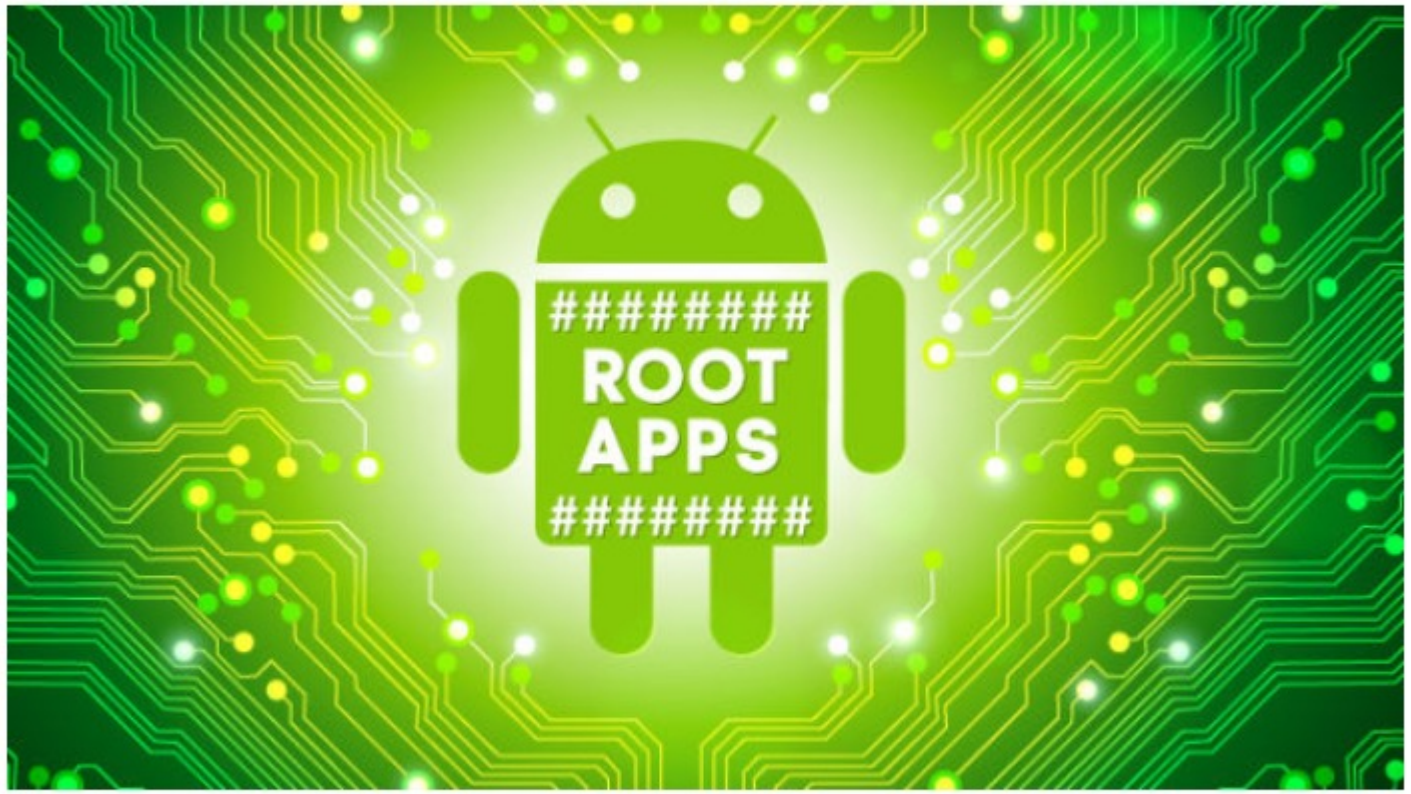
- Android application format
- App Distribution
- Permissions
- Data Storage
- Application Security
- Attacks and Malware
- Reverse-Engineering & Analysis

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

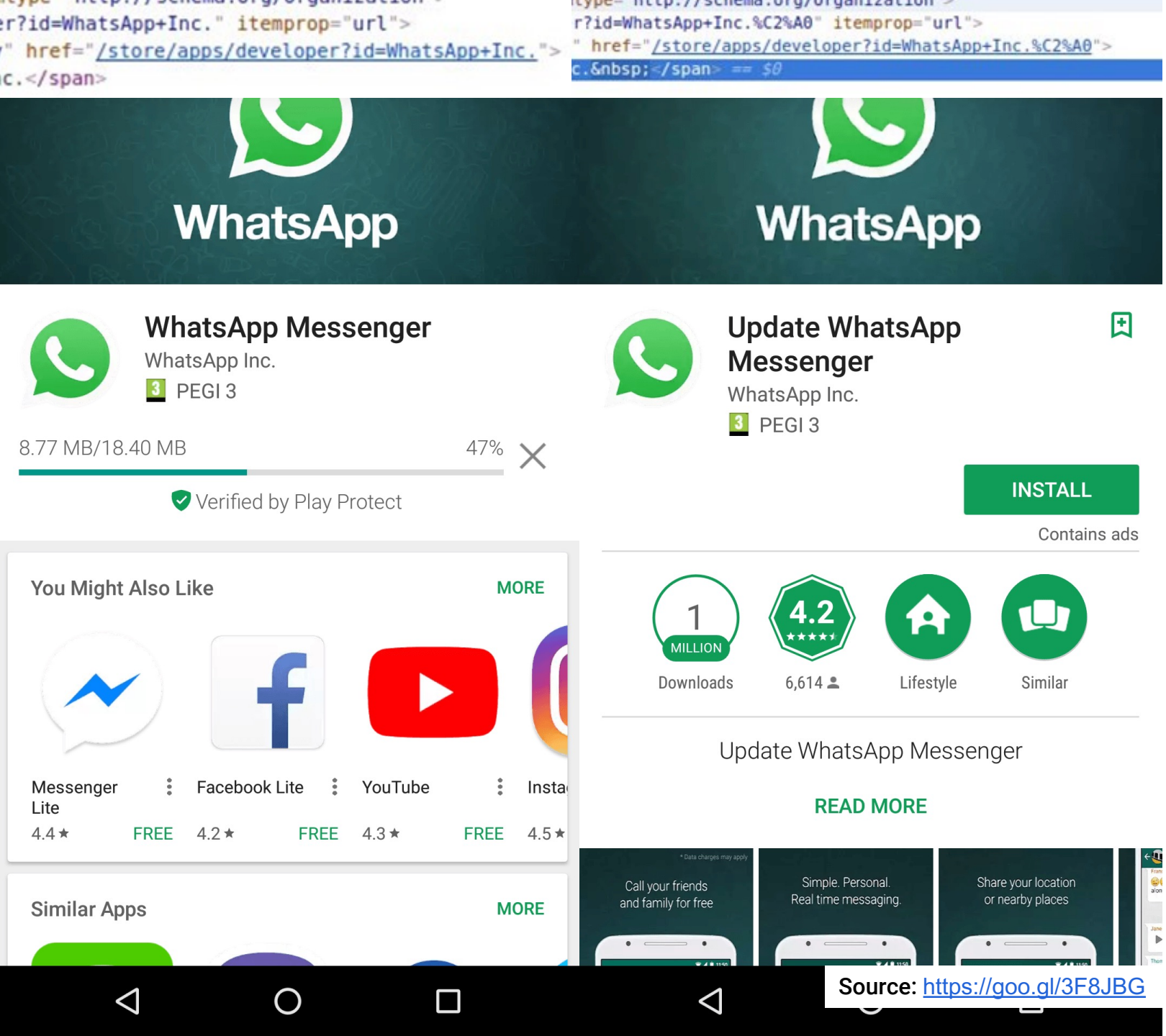
How?

- 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?

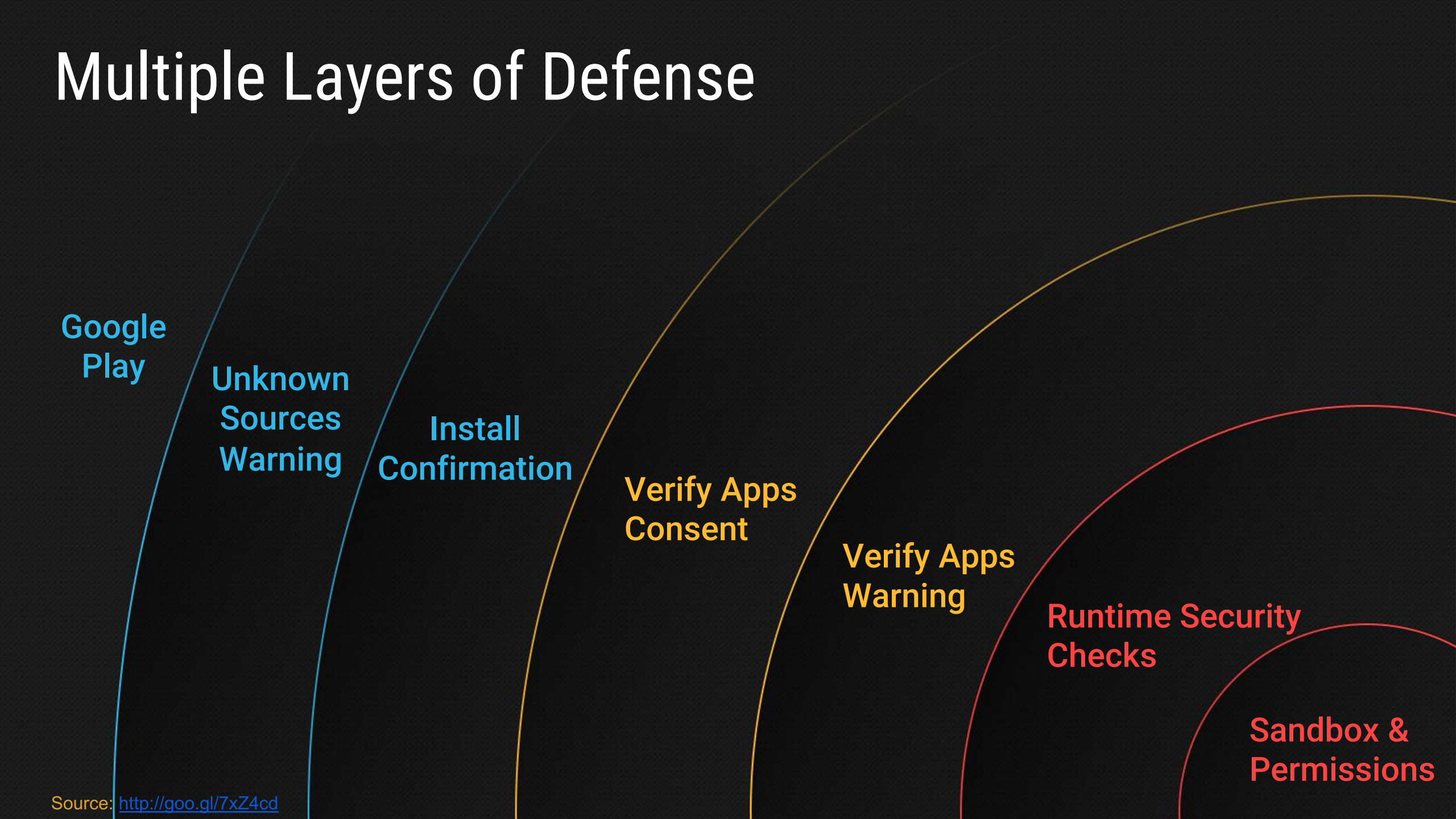
- Author added non-visible Unicode character to vendor name
- 1 to 5 Mio. downloads

Problem

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

IAIK TU Graz

Multiple Layers of Defense



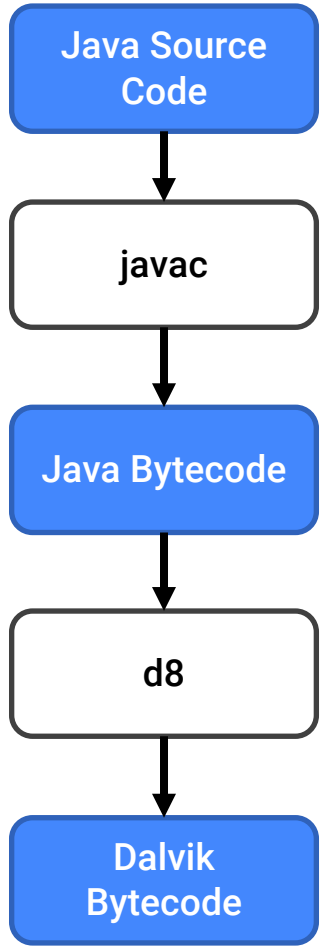
Source: http://goo.gl/7xZ4cd

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

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: 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 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

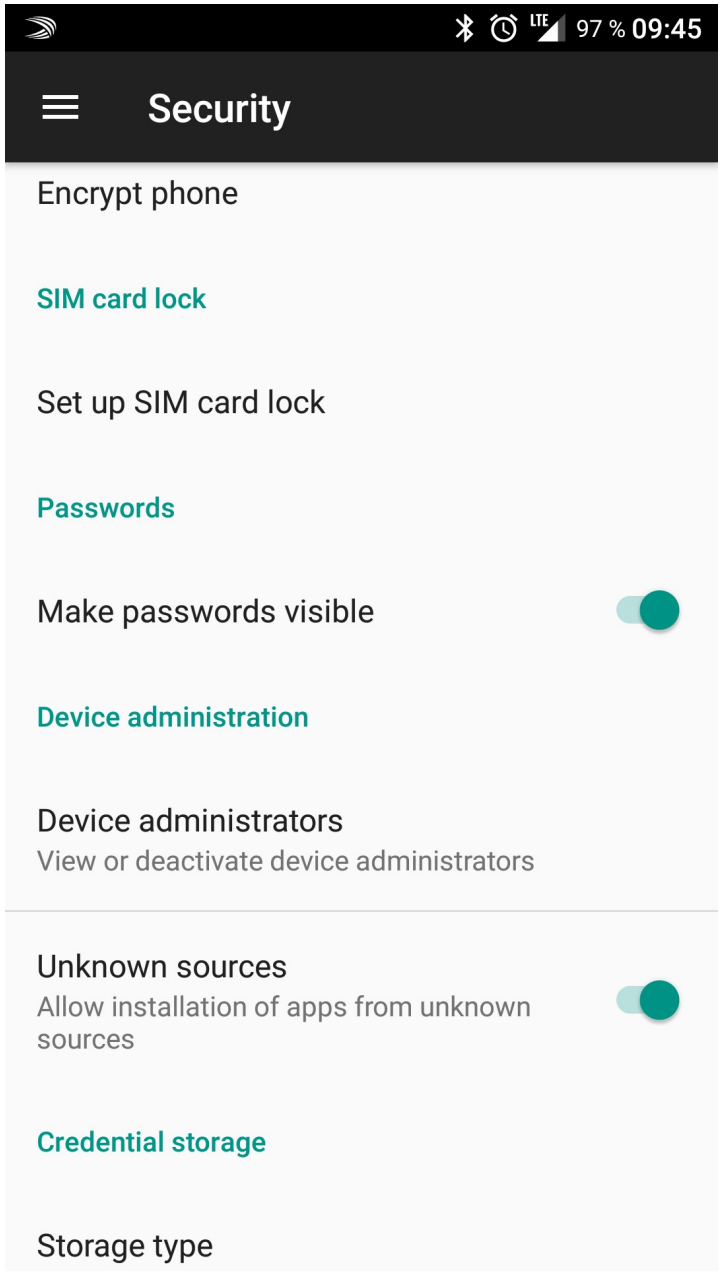
Source: guardsquare.com

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
- From file system
  - If app available as .apk file
  - Can be downloaded from anywhere



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!

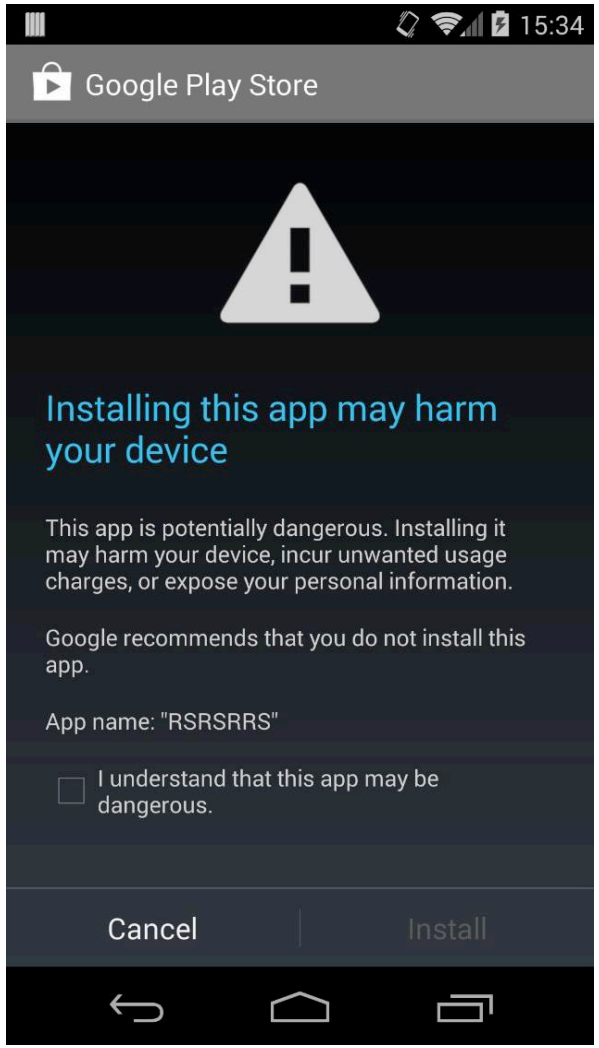
Source: J. Oberheide, C. Miller: "Dissecting the Android Bouncer"

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

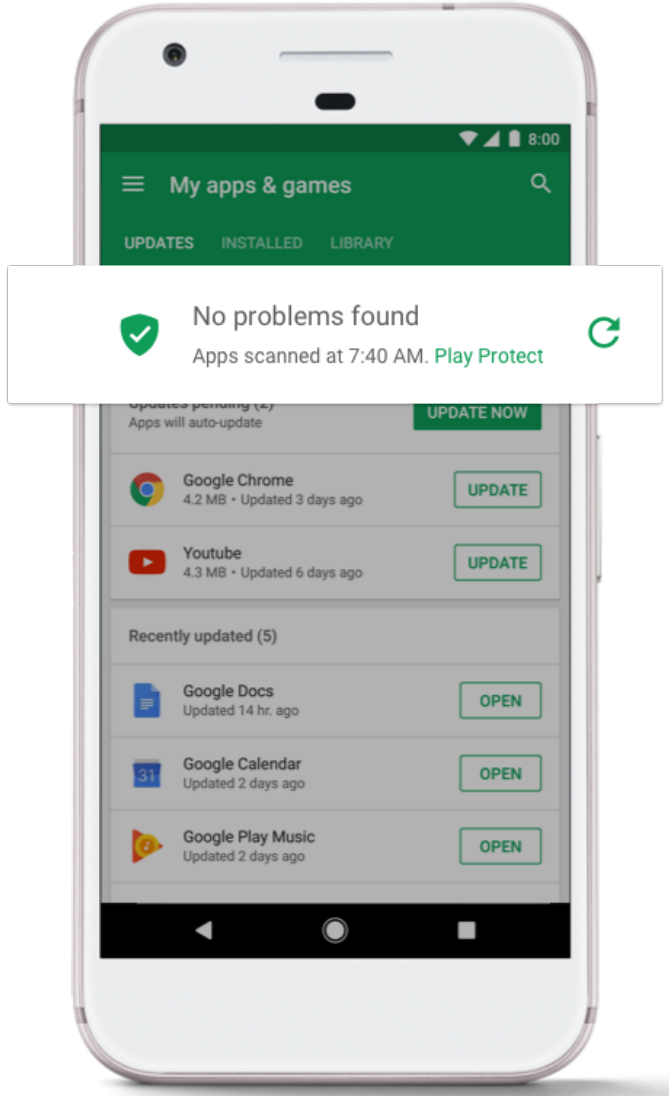
- Monitor device state
  - Dead or Insecure: A device stopped checking up with Verify Apps server
  - Likely either because malware disabled VA or device had to be reset
  - Both indicate a previously installed app was malicious
  - DOI app: Many devices DOI after installing this app

- 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

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!

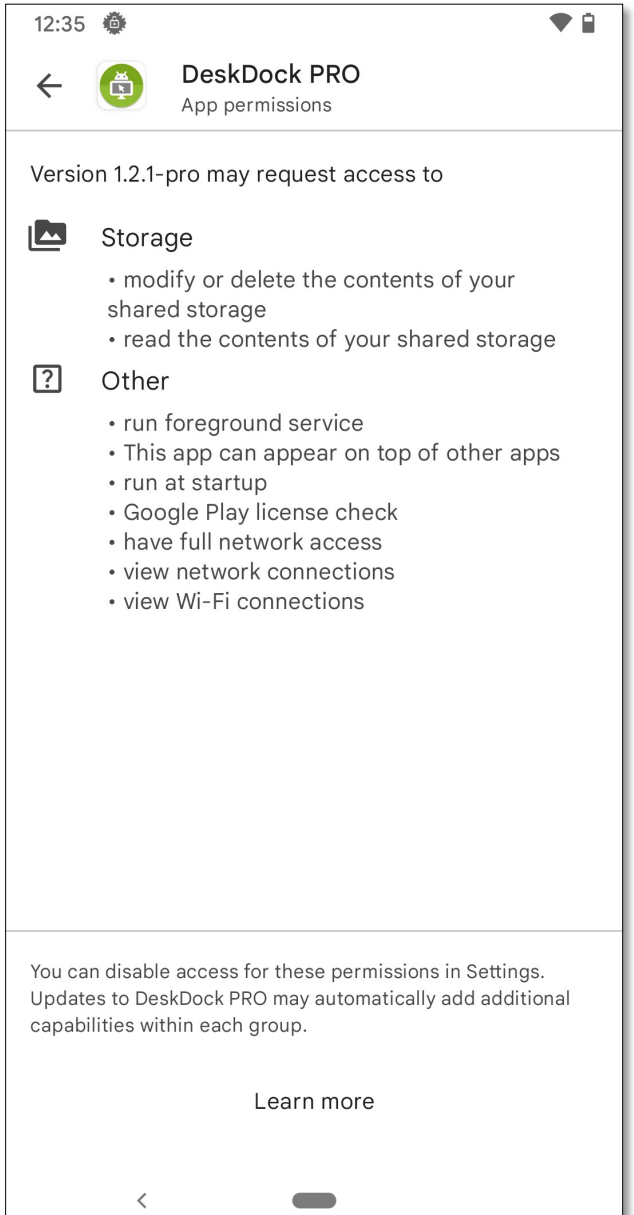
Permissions

Android Permissions

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

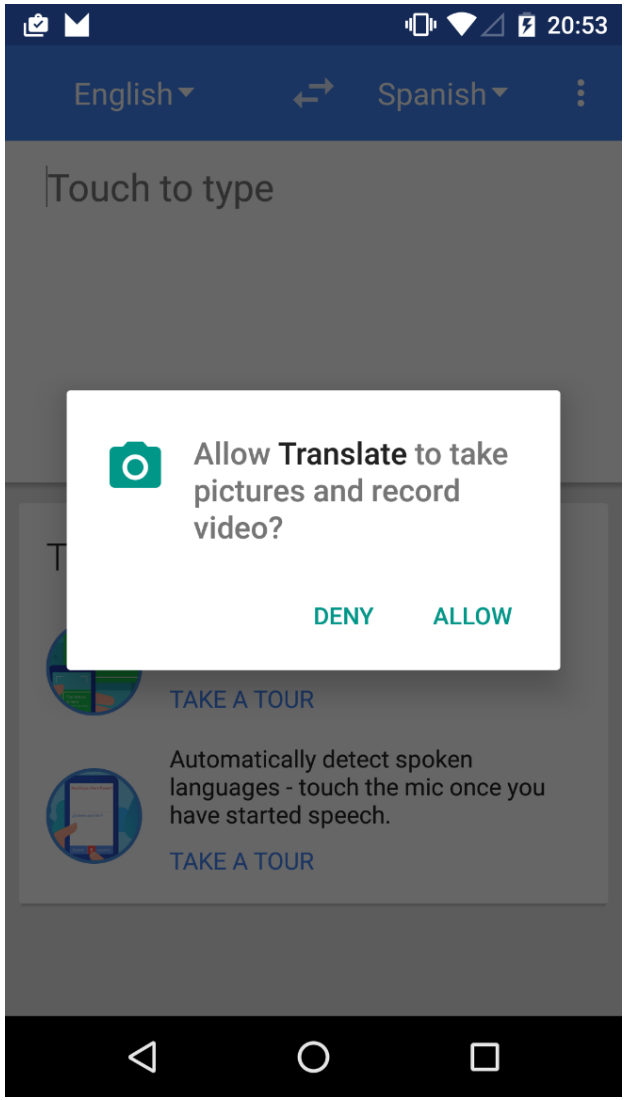
- OS defines a set of permissions, each with unique name
  - E.g. `android.permission.INTERNET`
  - Not all can be granted to third-party apps
- Developers specify needed permissions in AndroidManifest.xml
  - Some granted at install, others require runtime user consent
- Enforced at different levels
  - Kernel, e.g. `INTERNET` permission
  - Native service level, e.g. `READ_EXTERNAL_STORAGE` for SD card access

Install-Time 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

Runtime Permissions



- Need to prompt for dangerous permissions
- 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
- Managable by device owner
  - Useful for MDM

Permissions Groups

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`

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

Data Storage

Data Storage on Android

File Scopes

App-Specific Files

- Private to the application
- Sharing must be initiated by the app

Public Files

- Not linked to a particular app
- Media, Documents, Downloads, …

File Locations

Internal Storage

- Always available
- Very limited capacity

External Storage

- Might be removable (SD, USB)

Data Storage

On the first versions of Android, apps had

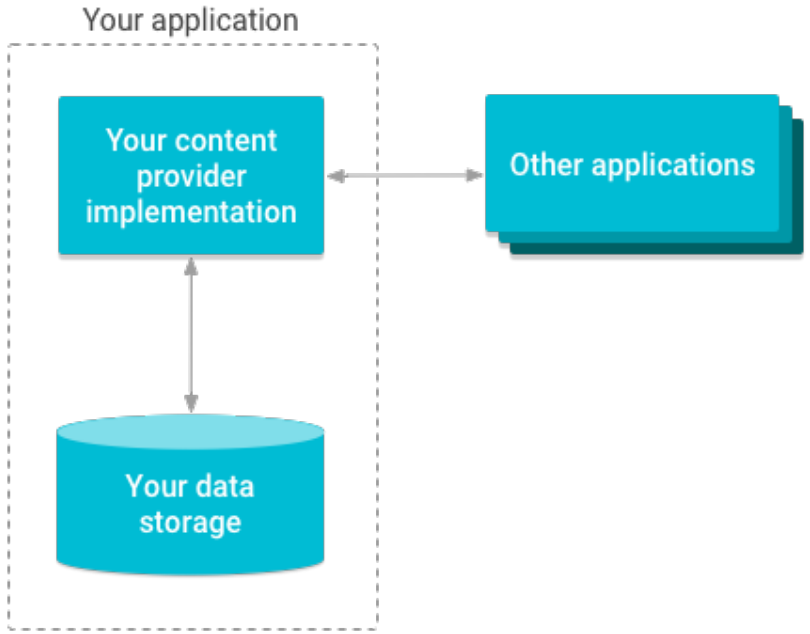
- Private folder(s) they could access without permissions
- Option to access (almost) full public file system by requesting permission
  - Simply use Java File APIs

Today:

- Private folder(s) mostly staid the same
  - Though additionally encrypted on Android 10+ to ward off root attackers
- Full public file system access no longer possible
- All public file access routed through system `ContentProviders`
  - Fine-grained per-path access control

ContentProvider

- Android-specific component for sharing data across processes
- Every data item is addressed through a `content://` URI
- Some implemented by the system
- Others by third-party applications
- Optionally protected by permissions



Picture: developer.android.com / Apache 2.0

Source: developer.android.com

ContentProviders for Data Storage

- **App-Specific Files**
  - **FileProvider:** Implemented by apps to expose their files to other apps
- **Media**: Pictures, Audio, Videos
  - **MediaStore:** Local centralised store, modifiable by apps
  - **CloudMediaProvider:** Read-only media from cloud (Android 13)
- **Documents**: Editable files (+ anything that's not media)
  - **DocumentProviders:** Central component of the **Storage Access Framework**
  - May be organised in a nested hierarchy

Sources: developer.android.com, medium.com/androiddevelopers

Storage Access Framework

Android 4.4+

An abstraction layer for file systems implemented on top of ContentProviders

- Several **DocumentsProviders** implement different data sources
  - Have a concept of nested document trees (~ folders)
  - External Storage
  - Media Store (videos, photos, audio)
  - Cloud Providers (Dropbox, Google Drive, …)
- Data source transparent to consuming applications
- User grants access to individual document or document trees

Source: developer.android.com

Scoped Storage

In Android 11, SAF was made **mandatory for accessing public files**

- Apps may write to MediaStore without requiring extra permission
- Permission still needed to access items created by other apps

- File API is transparently rerouted to MediaStore provider

- Exemption: ***All files access*** permission
  - Requires special approval for distribution through Google Play

Sources: developer.android.com, support.google.com

Application Security

Android Cryptography APIs

Java Cryptography Architecture: Consumer abstracted from Implementor

- **Cipher**: Encryption and Decryption
- **SecureRandom**: Random Number Generation
- **MessageDigest**: Calculating hash values
- **SecretKeyFactory**: Deriving keys from passwords
- …

Java Secure Socket Extension:

- **SSLSocket**: Provides TLS and SSL communication

HTTPS on Android

- Use Android's `HttpsURLConnection` class
  - By default: Secure `TrustManager` and `HostnameVerifier` (Details depend on Android version)
  - Possibility to use custom `TrustManager` and `HostnameVerifier`

- Use a third-party library such as OkHttp (built on top of `SSLSocket`)
  - Usually secure custom `TrustManager` and `HostnameVerifier`
  - Support self-signed certificates, certificate pinning, …

- Implement a custom HTTP stack on top of `SSLSocket`
  - Secure system-default `TrustManager`
  - HostnameVerifier up to developer!

Network Security Configuration (Android 7)

- XML-based system for configuring self-signed certificates and pinning
- These use cases no longer require custom validation code
- Default NSC: Don’t trust user-installed CA certificates

However

- Even the NSC can be misconfigured
  - Trust user-installed CAs
- Some applications still use custom `TrustManagers` or `HostnameVerifiers`
  - Overrides the NSC system altogether
- NSC only works on Android 7 or later
  - Silently ignored when app is run on older OS

Source: developer.android.com

Crypto Misuse on Android

Apps commonly make mistakes in their use of cryptographic primitives

- **Cipher**: Using ECB mode, Re-using IV and key combination
- **SecureRandom**: Re-using seed value
- **MessageDigest**: Using MD5 algorithm
- **SecretKeyFactory**: Too low iteration count, salt re-use
- **SSLSocket**: Insecure TrustManager

2020 study found that > 99% of apps using crypto APIs make some mistake

Source: Piccolboni et al: CRYLOGGER: Detecting Crypto Misuses Dynamically

Avoiding Crypto API misuse

- Use **trusted** high-level **libraries** instead of re-inventing the wheel
  - Crypto: Google Tink
  - HTTPS: OkHttp
- Follow **best practices** from official developer documentation
- Do not trust random code snippets from StackOverflow

Attacks and Malware

UI Deception

- Android allows apps to display overlays on top of system UI
  - Requires special permission (increasingly harder to obtain on modern Android)
- Accessibility Service apps can explore app UIs and inject input events
  - Need to be explicitly enabled though system settings

This enabled

- Context-aware clickjacking
  - Overlay system UI to trick user e.g. into granting specific permission
- Inferring on-screen keyboard input
  - Through ingenious side-channel that exploits the mitigation against clickjacking

No longer possible on modern Android versions (overlays restricted)!

Source: Fratantonio et al.: Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop

Containerization

- Android apps may dynamically load code from external files
- It is possible to execute complete APKs in the context of another app

- Malicious app may pretend to be legitimate app
  - By executing the original legitimate app in a malicious container
  - Can intercept and extract all user data

- Malicious apps can evade detection by Play Store analysis
  - Loading malicious components as plugins at runtime

Sources: Shi et al: „VAHunt: Warding Off New Repackaged Android Malware in App-Virtualization's Clothing".
Luo et al: "Anti-Plugin: Don't let your app play as an Android plugin", blog.avast.com

Side Channels

Malicious apps may extract sensitive information using seemingly harmless permissions

- **Motion**: Extract passwords from device movements (Cai et al, 2011)
- **Sound:** Use speaker and microphone as sonar, infer unlock patterns (Cheng et al., 2019)
- **Power:** Fingerprint websites from device's power consumption (Quin et al, 2018)
- **Time:** Detect installed applications by timing API calls (Palfinger et al., 2020)
- **Data:** Fingerprint accessed websites from network traffic statistics (Spreitzer et al, 2018)
- **Electromagnetic emissions**: Extract screen content via SDR receiver (Liu et al, 2021)

Component Hijacking

- Benign applications may leak permissions to malicious apps
  - E.g. due to exporting components designed for app-internal use
- Example:

Victim App A (holds `android.permission.CALL_PHONE`)

```
public class VictimActivity extends Activity {
    @Override
    protected void onCreate(@Nullable Bundle savedState) {
        Intent intent = new Intent(Intent.ACTION_CALL,
                                   getIntent().getData());
        startActivity(intent);
    }
}
```

VulnerableActivity.java

```
<manifest package=”at.victim">
    <uses-permission android:name="android.permission.CALL_PHONE" />
    <application>
        <activity
            android:name=".VictimActivity"
            android:exported="true”/>
    </application>
</manifest>
```

AndroidManifest.xml

Attacker App B (holds no permission)

```
public class AttackerActivity extends Activity {
    @Override
    protected void onCreate(@Nullable Bundle savedState) {
        Intent intent = new Intent();
        intent.setComponent(new ComponentName(”at.victim",
                                              ".VActivity"));
        intent.setData(Uri.parse("tel://0800 123123"));
        startActivity(intent);
    }
}
```

➔ Attacker can initiate phone calls without holding the corresponding permission

Source: Zhang et al.: AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications

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

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

Outlook

- 06.05.2021
  - Static and Dynamic Application Analysis

- 20.05.2021
  - Mobile Network Security