ISEC TU Graz

Android Application Security II

Mobile Security 2026

Florian Draschbacher
florian.draschbacher@tugraz.at

Android App Development

- Logic implemented in Java / Kotlin / C / C++
  - "Contract between app and OS": AndroidManifest.xml
- Apps are embedded in and driven by the platform framework
  - Many different entry points
  - Lots of callbacks
- Java APIs for basic functionality
  - Data Types, File System APIs, Networking, Crypto, …
- Android APIs for OS integration
  - IPC, HW access

ISEC TU Graz

Android API Architecture

- Many APIs are Stubs for RPC interfaces to system services
  - Run inside the system_server process (runs as system user)

- For example:
  LocationManager is the RPC interface for LocationManagerService

- Example call flow:
  1. Call LocationManager.getLastKnownLocation()
  2. Binder is used to forward call to system_server process
  3. LocationManagerService ensures that caller holds LOCATION permission
  4. Result is returned through Binder

Demo: https://cs.android.com

Inter-Process Communication

- At the lowest (kernel) level, IPC is implemented through **Binder**
- Arguments need to be serialised for passing them to other processes
  - Affected classes need to implement Parcelable interface

```
public class MyParcelable implements Parcelable {
    private int mData;

    public int describeContents() { return 0; }

    public void writeToParcel(Parcel out, int flags) { out.writeInt(mData); }

    public static final Parcelable.Creator<MyParcelable> CREATOR = new Parcelable.Creator<MyParcelable>() {
        public MyParcelable createFromParcel(Parcel in) { return new MyParcelable(in); }

        public MyParcelable[] newArray(int size) { return new MyParcelable[size]; }
    };

    private MyParcelable(Parcel in) { mData = in.readInt(); }
}
```

Android Versions / API Levels

- The Android application framework evolved over time
  - New APIs, deprecated APIs, changed permissions, policies, UI design, …
  - Fragmentation: Backwards compatibility is important

- New Android version: New API (=SDK) Level
  - Accessible through Build.VERSION_CODE

- Every app references 3 different API versions:
  - Minimum SDK Version: App requires at least this API version
  - Target SDK Version: App operates as if it was running on this Android version
  - Compile SDK Version: All classes/methods known in this version may be used

Target SDK Level

- Apps can set Target SDK Level to bypass API policies introduced in later version

Example:

- Runtime permissions were added in Android 6.0 (SDK/API Level 23)
- Only affected applications targetting API Level 23!
- Apps could set lower targetSdkVersion to bypass user prompts

- **Google Play** only allows upload of apps targetting most recent SDK Level!
  - Since 2023
  - Used to be ~2 releases behind the most recent SDK Level

Key Framework Components

An excerpt from some AndroidManifest.xml

```
…
 <application>
    <activity android:name=".MainActivity” android:exported="true” >
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

    <service android:name=".SomeService” android:exported="false” />

    <receiver android:name=".MyBroadcastReceiver” android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.AIRPLANE_MODE" />
        </intent-filter>
    </receiver>

    <provider android:name=".FileProvider” android:authorities="com.demo.fileprovider” android:exported="false">
        <meta-data android:name="android.support.FILE_PROVIDER_PATHS"
          android:resource="@xml/file_paths" />
    </provider>

  </application>
 ….
```

Intent

- In many IPC transactions, an Intent carries arguments
  - Specifying the **component** that should be launched
    - Explicitly (package and class name)
    - Implicitly (action that should be supported)
  - **Data**: A URI or file path to a remote or local file
  - **Extras**: Key-Value pairs of arbitrary data

- The system is responsible for
  - Resolving the Intent: What component should be used
  - Instantiating and starting the target component

Context

- The base class for most Android app components

- Offers helper functions for
  - Reading app resources and assets
    - Context.getResource(), Context.openAssetStream()
  - Obtaining IPC handles for system services
    - Context.getSystemService()
  - Accessing the app-private folder
    - Context.getFilesDir()
  - Launching or registering app components
    - Context.startService(), Context.registerReceiver()
  - ...

Activity

- ~"A UI screen and its logic controller"
- All (visible) apps have a Main Activity
  - May be started from the app launcher
- Launched through `Context.startActivity()`
  - Intent specifies target activity & arguments
- Must be declared in **AndroidManifest.xml**

Picture: developer.android.com / Apache 2.0

ISEC TU Graz

Intent Filters

- If Activity should be launchable **by other apps**:
  - Mark as exported in AndroidManifest.xml
- If Activity should support launching through **implicit** Intent:
  - Mark with intent-filter in AndroidManifest.xml

Explicity set exported=false otherwise!

```
<activity android:name=".MainActivity” android:exported="true">
   <intent-filter>
      <action android:name="android.intent.action.MAIN" />
      <category android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
   <intent-filter>
      <action android:name="android.intent.action.SEND"/>
      <category android:name="android.intent.category.DEFAULT"/>
      <data android:mimeType="text/plain"/>
   </intent-filter>
</activity>
```

BroadcastReceiver

- ~"Subscribe to system-wide events"
- Broadcasts are Intents published through `Context.sendBroadcast()`
  - Sent by system components or apps to communicate certain events
  - Their *action* carries information about the specific event
- BroadcastReceivers allow subscribing to specific events
  - IntentFilter specifies desired action
- Broadcast senders or receivers may be protected using a permission
  - Sender may restrict receivers to only those holding given permission
  - Receiver may only accept broadcasts send by apps holding given permission

BroadcastReceiver

- May be registered at runtime or statically through AndroidManifest.xml
  - Statically registered: Don’t receive implicit broadcasts
  - Dynamically registered: Only works while app is running

```
<receiver android:name=".MyBroadcastReceiver" android:exported=”true">
   <intent-filter>
      <action android:name="android.intent.action.AIRPLANE_MODE" />
   </intent-filter>
</receiver>
```

```
BroadcastReceiver receiver = new MyBroadcastReceiver();
IntentFilter filter = new IntentFilter("android.intent.action.AIRPLANE_MODE");
ContextCompat.registerReceiver(context, receiver, filter, ContextCompat.RECEIVER_EXPORTED);
```

Don’t export BroadcastReceivers for app-internal broadcasts!

ISEC TU Graz

Service

- ~"Keep app running while no Activity is shown"

- Foreground Service: Visible to user through notification
- Background Service: Almost impossible nowadays
  - Battery drain and security issues

- Must be declared in **AndroidManifest.xml**
  - Binding to it (attaching to IPC interface) may be restricted using permission
  - If bindable or launchable from other apps: Set `exported=true`

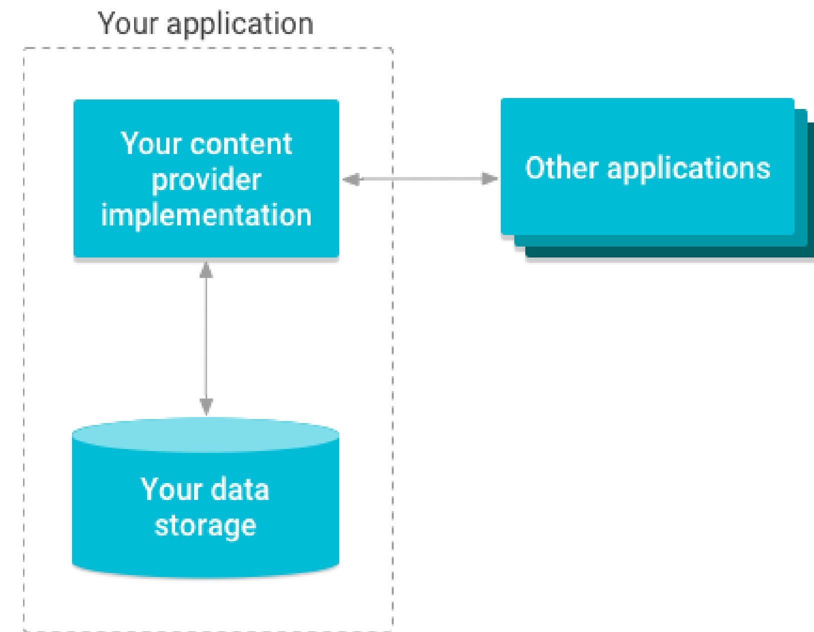
- Started using `Context.startForegroundService()`

Services and IPC

- Services may offer functionality for call by other processes
  - RPC implemented through Binder
- Interface defined in Android Interface Definition Language (AIDL)
  - Proxy (hiding away low-level marshaling and RPC) auto-generated
- Apps may call Context.bindService() to obtain service's Binder handle
  - Allows invoking functions of the service's RPC interface

ContentProvider

- ~"Selectively grant other apps access to database or files"
- Every data item is addressed through a `content://` URI
- Some implemented by the system
  - Others by third-party applications
- **Optionally protected by permissions**
  - **Separate permissions for read/write**



Picture: developer.android.com / Apache 2.0

Source: developer.android.com

Data Storage

Data Storage on Android

File Types

Pictures, Videos, Documents, Music, Databases, …

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

Today:

- Private folder(s) mostly staid the same
  - Though additionally encrypted on Android 10+
- Full public file system access no longer possible for most apps
  - There is `MANAGE_EXTERNAL_STORAGE` (but limited use for Google Play distribution)
- All public file access routed through system `ContentProviders`
  - Fine-grained per-path access control

ContentProviders for Data Storage

- **App-Specific Files**
  - **FileProvider**: Implemented by apps to expose their files to other apps
- **Media**: Pictures, Audio, Videos
  - **MediaProvider:** 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/Provider without requiring extra permission
- Permission still needed to access items created by other apps
  - Access either granted by another app with access to the file
  - Or by a user file/folder picker

- 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

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/Coding Agents!

More Interesting APIs

Reflection

- Android apps may use Java reflection
  - Dynamically accessing classes, methods or fields through their names
- This sometimes allowed or facilitated apps to bypass API restrictions
  - E.g. On early Android versions, Wifi AP could be started despite no official API
- Starting with Android 9: Restrictions on non-SDK interfaces
  - Sine then: More and more APIs hidden away from apps
- There still are ways for bypassing these restrictions!
  - Android 9 & 10: Use Double-Reflection / Meta-Reflection
  - All versions: E.g. use JNI / Java Unsafe API to manipulate ART runtime structs

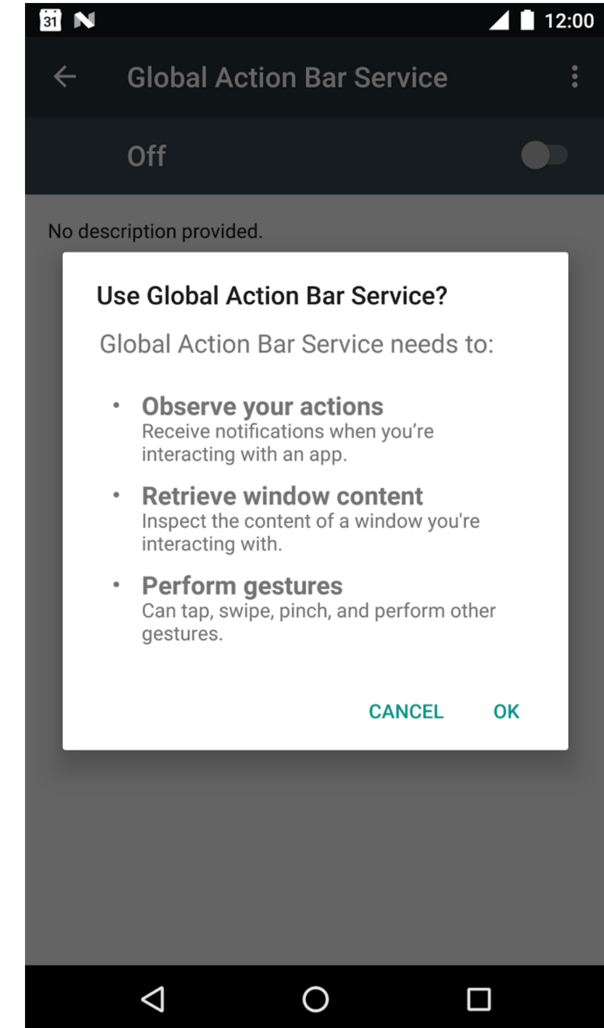
AccessibilityService

~"Service for helping impaired users navigate their device"
- Screen readers, Voice control, …

- Must be explicitly enabled by the user
  - Multiple services may be enabled in parallel

- Can access UI of other apps and inject input events
  - Very powerful role on the device!

- Google Play is very strict on which apps allowed to use AccessibilityService API
  - If not for accessibility: Disclose exact use, manual review



VpnService

~"Service for rerouting device's Internet traffic"

– Receives IP packets of all other processes

- Must be explicitly allowed by the user
  – Indicator in status bar, only one active VpnService allowed
- May collect information about user
  – HTTPS: Accessed hosts
  – HTTP: Full request + response
- Google Play is restricting use of VpnServices
  – Similar process as for AccessibilityService

Device Administration API

- DeviceAdmin Apps: May enforce security policies on device
  - E.g. password strength, disable camera, remote wipe / lock
- Must be explicitly enabled by user
  - Once enabled: Must be disabled before app can be uninstalled
- Even more powerful role: DeviceOwner
  - Must be explicitly enabled through ADB or Android Enterprise
  - Can only disable (and therefore uninstall) itself
  - May
    - Install apps without user consent
    - Reboot the device
    - Install trusted CA certificates
    - ...

Vulnerabilities and Attacks

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`)

```java
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

```xml
<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)

```java
public class AttackerActivity extends Activity {
    @Override
    protected void onCreate(@Nullable Bundle savedState) {
        Intent intent = new Intent();
        intent.setComponent(new ComponentName(”at.victim",
                                              ".VictimActivity"));
        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]()

Crypto API 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]

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

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

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]()

Google Play Core Library Vulnerability

- Registered an unprotected BroadcastReceiver
  - Data parsed from received Intents: Data URI for download, Split ID
- Loaded data from URI and put into
  - /data/data/com.app.abc/unverified-splits/{split_id}
- Unverified splits are verified and moved to verified-splits folder
- Files from verified-splits folder loaded into classpath

Problem: split_id not validated!

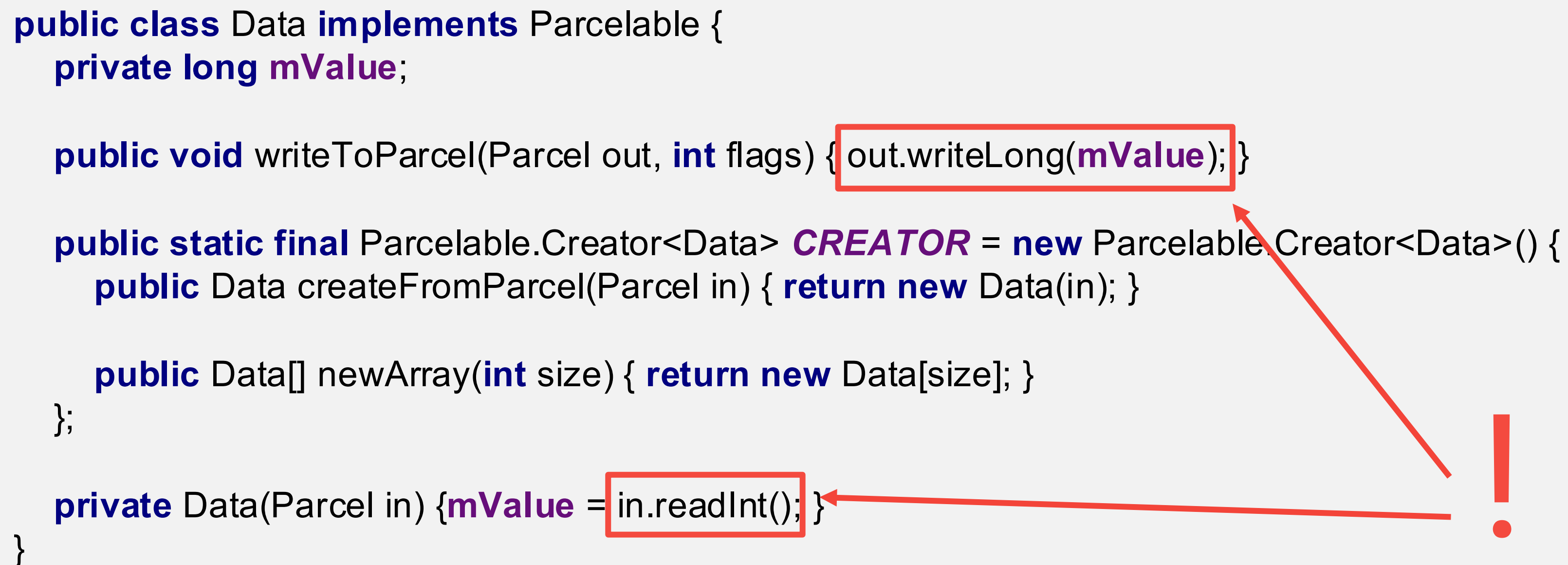
- Path traversal: Set split_id to “../verified_splits/config.test”
- Allows code execution in the context of victim app!

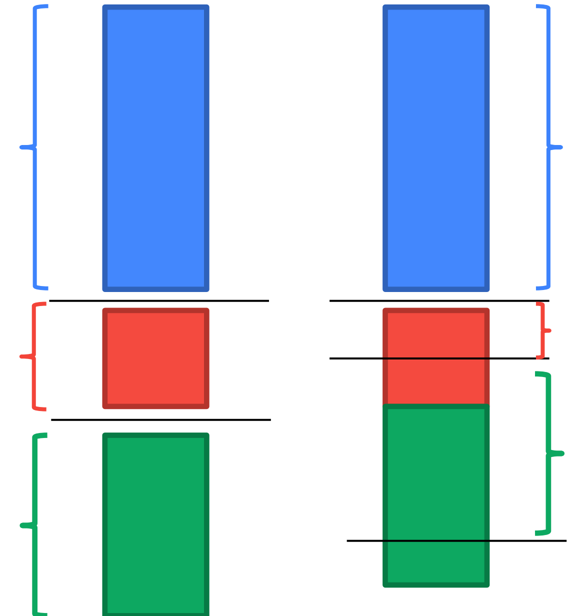
Source: oversecured.com

Parcelable Mismatch

- For every IPC transaction (through Binder), arguments have to be serialised
  - Arguments need to implement Parcelable interface from earlier today

- What if parsing the serialised Parcelable does not yield the original instance?
  - i.e. !new Data(writeToParcel(myData, 0)).equals(myData)
  - Data misalignment, subsequent data will be read from wrong offset!





Parcelable Mismatch

- If transaction contains Parcelables originating from system & malicious app:
  - Data controlled by malicious app may spill into data originating from system
- E.g. Delivering broadcast to victim app
  - Attacker App ➔ System ➔ Victim App
  - Attacker App Parcelable: Intent
  - System Parcelable: ActivityInfo
  - May be exploited for code execution!

CVE-2017-0806	GateKeeperResponse	CVE-2018-9474	MediaPlayer.TrackInfo
CVE-2017-0664	AccessibilityNodeInfo	CVE-2018-9431	OSUInfo
CVE-2017-13288	PeriodicAdvertisingReport	CVE-2018-9522	StatsLogEventWrapper
CVE-2017-13289	ParcelableRttResults	CVE-2018-9523	Parcel.writeMapInternal()
CVE-2017-13286	OutputConfiguration	CVE-2021-0748	ParsingPackageImpl
CVE-2017-13287	VerifyCredentialResponse	CVE-2021-0928	OutputConfiguration
CVE-2017-13315	DcParamObject	CVE-2021-0685	ParsedIntentInfo
CVE-2017-13310	ViewPager's SavedState	CVE-2021-0921	ParsingPackageImpl
CVE-2017-13312	ParcelableCasData	CVE-2021-0970	GpsNavigationMessage
CVE-2017-13311	ProcessStats	CVE-2021-39676	AndroidFuture
CVE-2018-9471	NanoAppFilter	CVE-2022-20135	GateKeeperResponse

- Full writeup: https://github.com/michalbednarski/ReparcelBug2
- Parcelable was responsible for a series of critical Android vulnerabilities
  - Situation improved with systemic changes in Android 12

Picture Source: Ke et al.: Android parcels: the bad, the good, the better. BlackHat Europe 2022

ISEC TU Graz

Outlook

- 24.04.2026
  - iOS Platform Security
- 08.05.2026
  - iOS Application Security