CS 489 / 698
Software and Systems Security

Module 4

Mobile OS Security

Winter 2024

Module Outline

1. Overview of Android OS
2. Security Mechanisms
3. App Security
4. Advanced Topics: Permission Maps and Access Control Anomalies

Mobile devices

- Embedded
- Ubiquitous connectivity (wireless, cellular / 4G / 5G, NFC, …)
- Sensors: accelerometer, GPS, camera, …
- Computation: powerful CPUs (>1Ghz, multi-core)

- Two major OS: Android / iOS

Reference: https://gs.statcounter.com/os-market-share/mobile/worldwide

Mobile devices

7.3 Billion — **Is the Global Mobile Android Population**

>1 Billion — **Is the number of Android devices sold annually**

Smart Watches

Smart TVs

Smart Game Suites

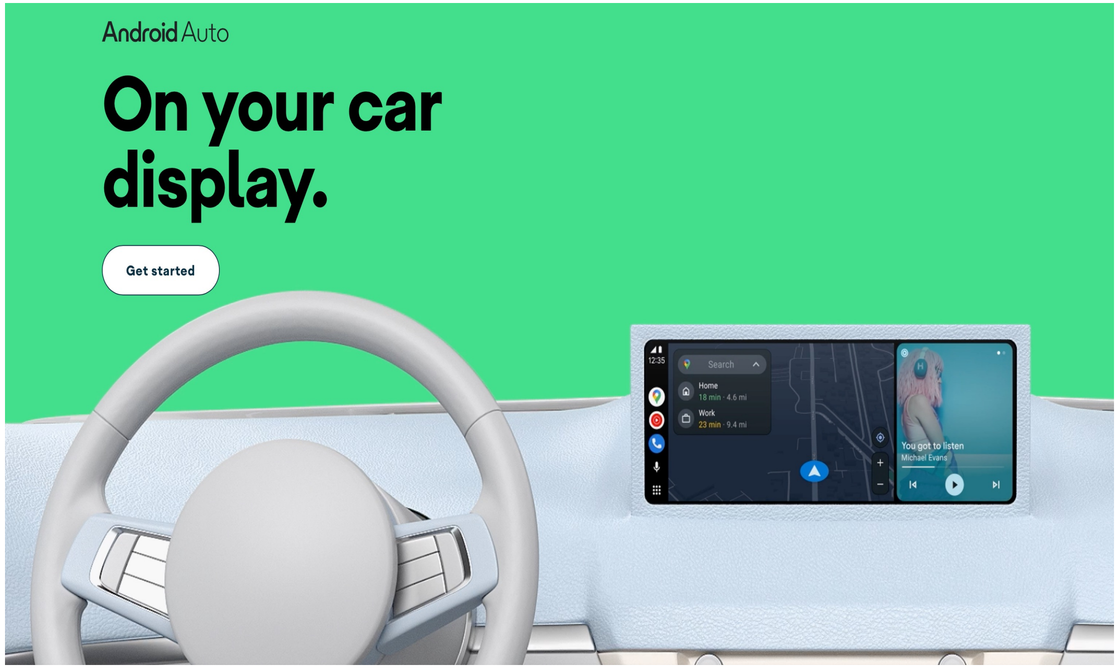
Smart Auto Guidance

Mobile devices (Android Auto and Android Automotive)

- Unlock your car with your phone,
- control infotainment system,
- call management, navigation

- Two flavors:

1. Android Auto: Connect your phone to your car display.
2. Android Automotive: OS runs directly on the in-vehicle hardware



Mobile Devices: Trends

- Increased reliance on mobile devices
  - Banking, work, personal data, communication
  - Data security and authentication is thus highly important
- Used for work
  - Bring your own device (BYOD)
  - Mobile Device Management used to protect enterprise
- Rely on different technologies
  - E.g., native development, web

What is Mobile Security?

- Or "What makes security different under the mobile platform?"
- Different communication channels
  - WiFi, NFC, cellular, Bluetooth, …
- Different actors
  - Broader range of users compared to traditional platforms
  - More prone to social attacks
- Different side channels
  - Examples: reflection, …

What is Mobile Security?

- Or "What makes security different under the mobile platform?"
- (Relatively) limited computing power / resources
  - Limited battery, memory, CPU, bandwidth
  - Cannot deploy traditional security solutions right out of box
- Portable
  - Non-conventional attack vectors, e.g., stealing, loss
  - Subject to short-range attacks (NFC, Bluetooth)
- Highly customized and fragmented
  - The OS is customized by different parties:
    - Hardware manufacturers, e.g., Qualcomm, MediaTek
    - Original Equipment Manufacturers (OEMs), e.g., Samsung, Xiaomi
    - Carriers, e.g., Bell, Telus, AT&T

What is Mobile Security?

- Or "What makes security different under the mobile platform?"
- Continuous and fast-paced evolution
  - Since its introduction in 2009, Android has released 25 major versions
  - Mobile users need to keep up with fast updates
- Wide range of software (mobile apps) than traditional platforms
  - "there is an app for it"
  - Preloaded (trusted) apps
  - (untrusted) third-party apps (to be installed)
- …

Mobile Threats:
What is stored on mobile devices?

- Depends on the type of mobile devices
- SmartTVs store: streaming services credentials, viewing history, …
- Smartphones store:
  - Contacts
  - Email, social network chats
  - Banking, financial apps data
  - Multimedia data
  - Location information and history
  - …

Mobile Threats:
What is stored on mobile devices?

- Depends on the type of mobile devices
- SmartTVs store: streaming services credentials, viewing history, etc
- Smartphones store:
  - Contacts
  - Multimedia data
  - Location information and history
  - …

What would happen if an "entity" accesses your mobile device?

Mobile Threats
Threat model

- Attackers with physical access
  - Unlock device
  - Exploit vulnerabilities to circumvent locking

Mobile Threats:

- Attackers with physical access
  - Unlock device
  - Exploit vulnerabilities to circumvent locking
- Attackers with remote access
  - Get the user to install malicious app (malware)
    - Use malware to steal sensitive data or perform malicious operations
    - Exploit various flaws in the mobile ecosystem for distribution, propagation and performing malicious functionality
  - Send malicious / malformed content to the device
    - Examples: send a malformed SMS,
    - Exploit various vulnerabilities

Protection against Physical Attacker
Authentication

- Protect against physical attacker via (mobile-specific) authentication
  - Something the user knows: PINs, Patterns, Passwords
  - Something the user is: Biometrics

Protection against Physical Attacker
Authentication via Patterns

- Attacks:
  - Smudge Attack

Protection against Physical Attacker

Authentication via Patterns / PINs

- Attacks:
  - Smudge Attack
- Another problem: entropy:
  - People tend to chose simple patterns
  - With 4 strokes, there are 1600 patterns.
- Online brute forcing PINs

Protection against Physical Attacker

Biometric authentication

- Fingerprint scanners, iris scanners, face unlock
- Standard biometric security concerns:
  - Subject to high false positives and false negatives
  - Cannot be changed
  - Not secret
- There is usually a fallback authentication (e.g., PIN)
  - The authentication strength reduces to the weakest authentication method

Protection against Physical Attacker
Next Defense:

- Protect against brute force attacks by erasing data if too many tries.
- Protect a stolen phone
  - Using GPS ”where is my phone”
  - Backup device
  - Device wipe

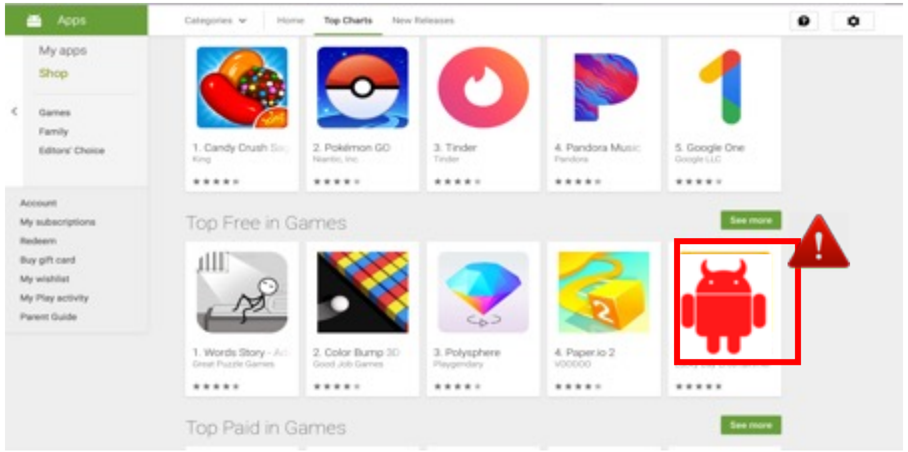
Protection against Malware

- Goal of the attacker: Lure the user into installing malware
  - Use malware to steal sensitive data or perform malicious operations
  - Exploit various flaws in the mobile ecosystem for distribution, propagation and performing malicious functionality

Characteristics of Mobile Apps / markets

- Apps in Android are Self-Signed.
- Apps can be downloaded from Google Play and from 3rd party markets
- It is easier to distribute apps on markets
- Although some markets perform automated scanning, malware is a serious issue

Malicious apps & Potentially Harmful Apps (PHAs) may appear!



Malicious Apps (malware) always on the Rise

172 malicious apps with 335M+ installs found on Google Play

by MIX — 3 months ago in APPS

Malicious apps exploit different vulnerabilities and attack vectors, introduced by different actors in the ecosystem

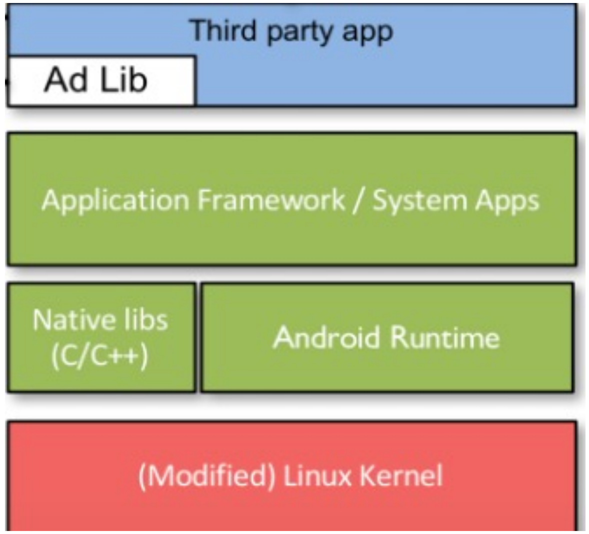
Malicious apps (malware)

- Malware exploit flaws in the mobile ecosystem
- The flaws may be introduced unintentionally:
  - Development mistakes
  - Improper market vetting
  - Buggy tools
  - …

Malicious apps (malware)

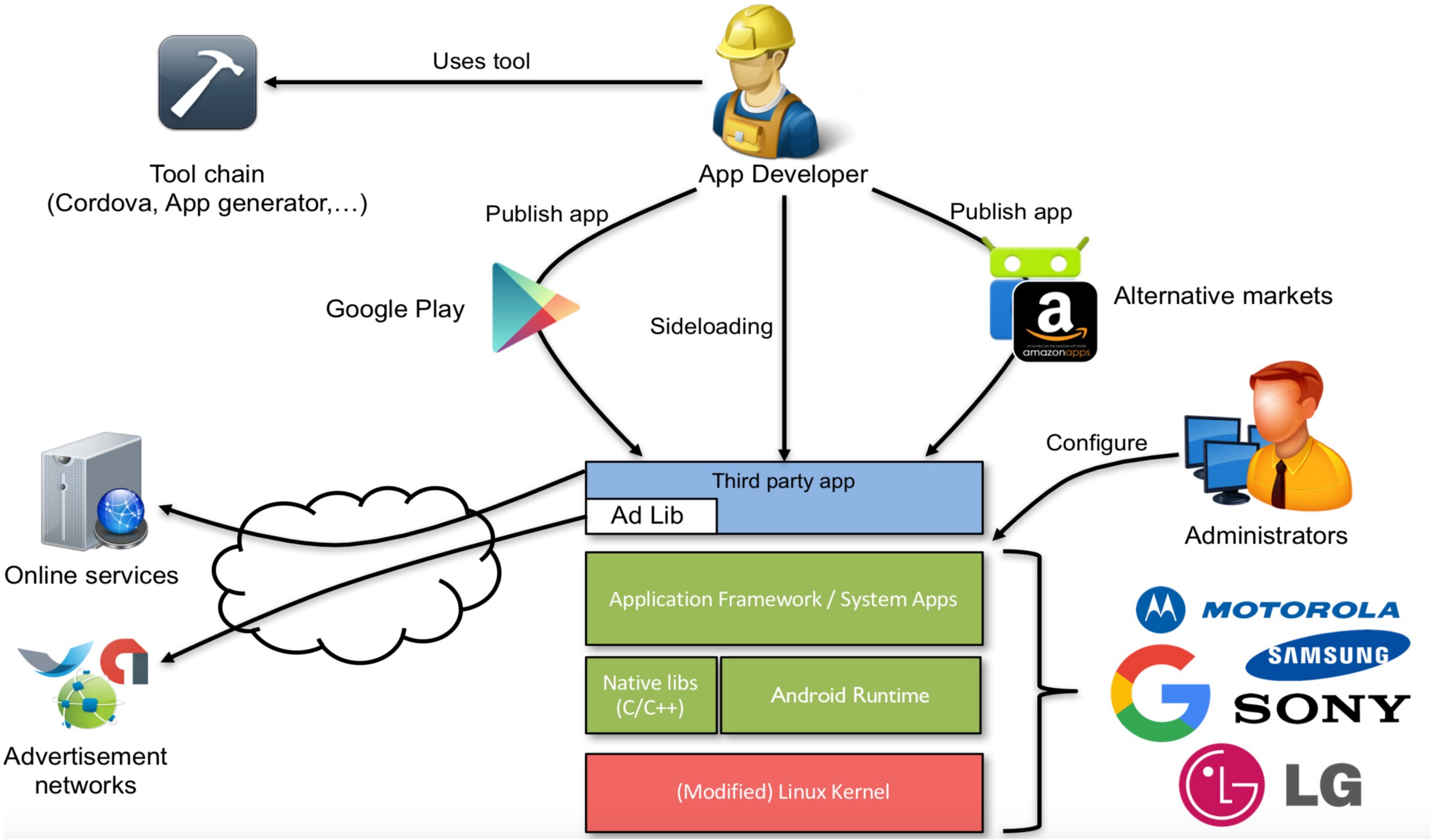
- Malware exploit flaws in the mobile ecosystem
- The flaws may be introduced unintentionally:
  - Development mistakes
  - Improper market vetting
  - Buggy tools
  - …
- The flaws may also be introduced intentionally
  - Non-malicious OEM developers leaving debugging backdoors.
  - Malicious libraries embedded in a benign app
  - Malicious insiders planting backdoors in EOM codebases
  - …

Who introduces flaws in the Android mobile ecosystem?
Background



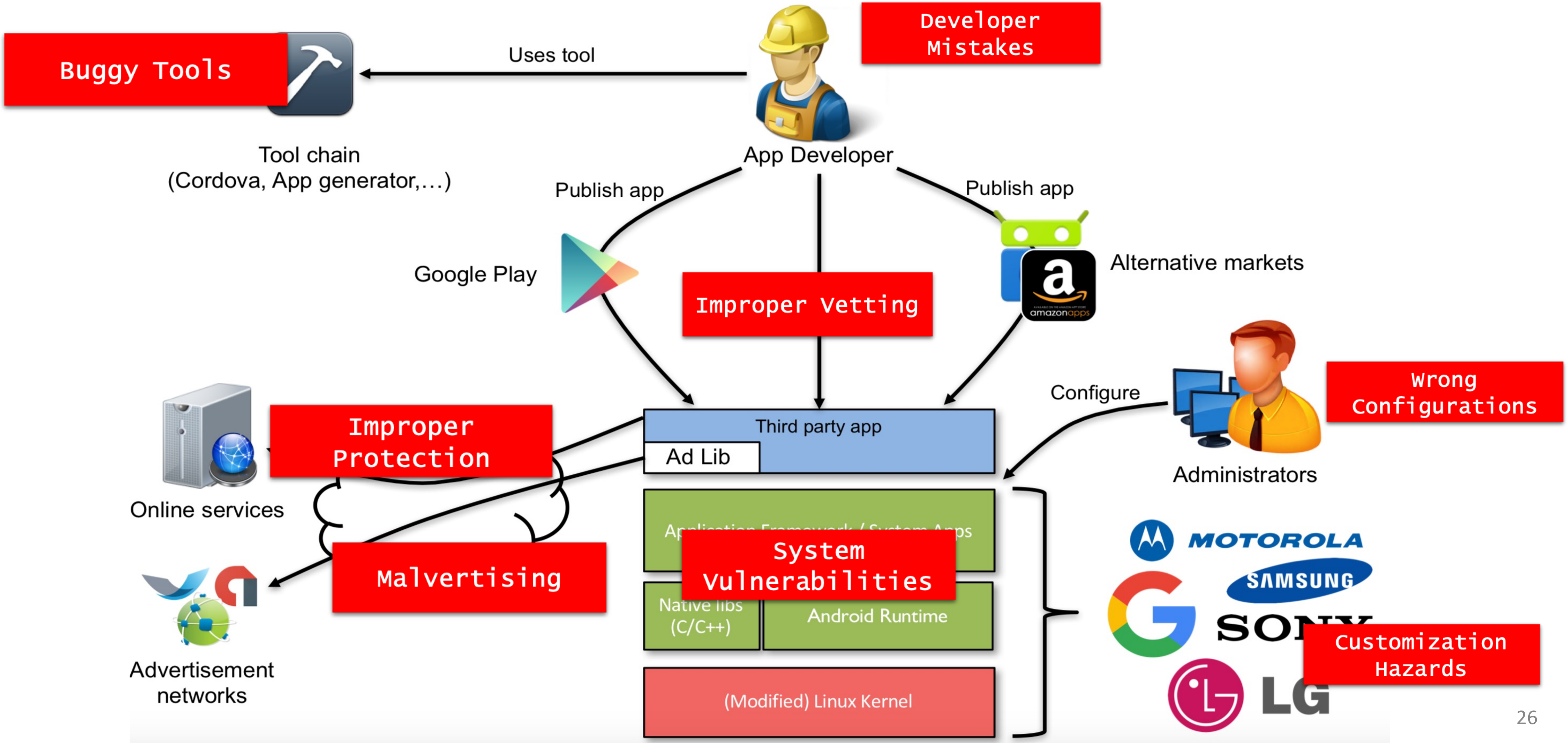
Who introduces flaws in the Android mobile ecosystem?

Actors in the Android ecosystem



Who introduces flaws in the Android mobile ecosystem?

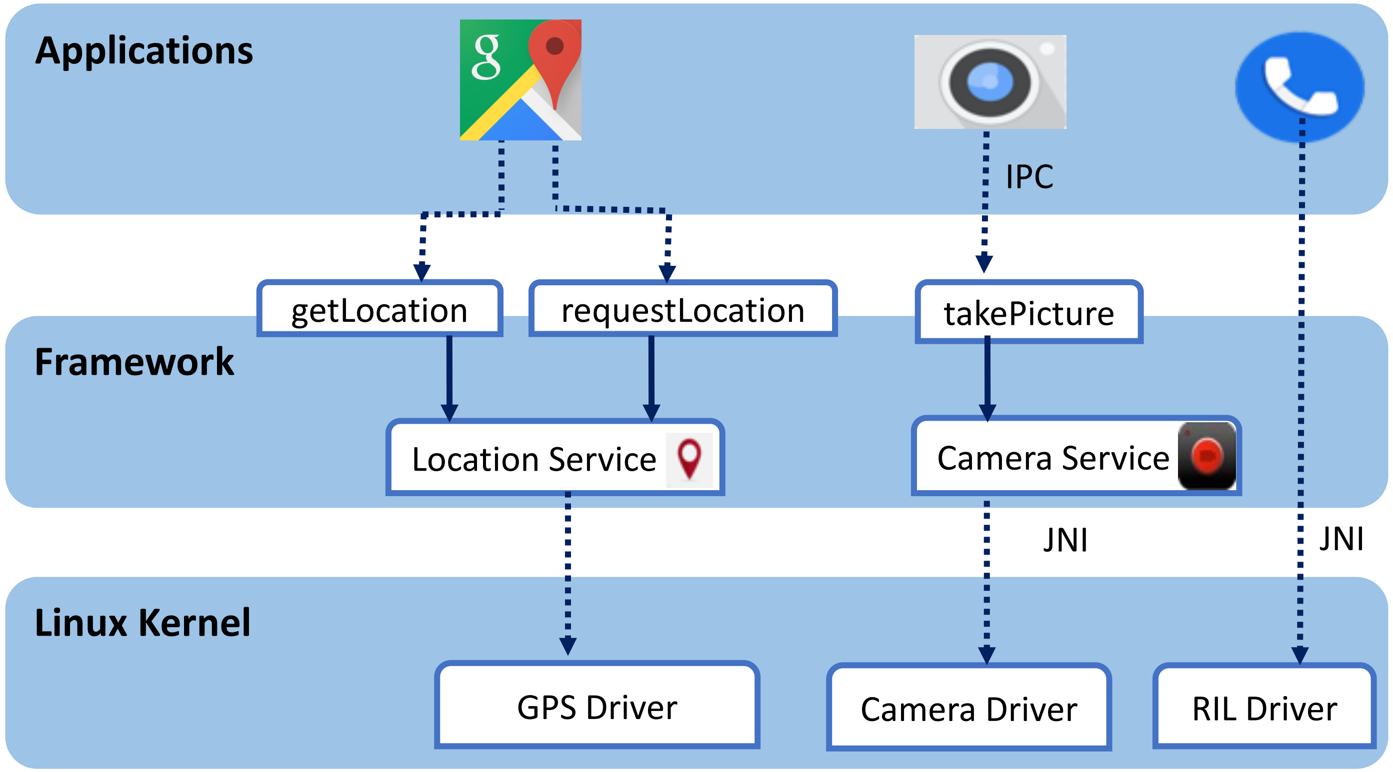
Attack vectors



Protection against Malware

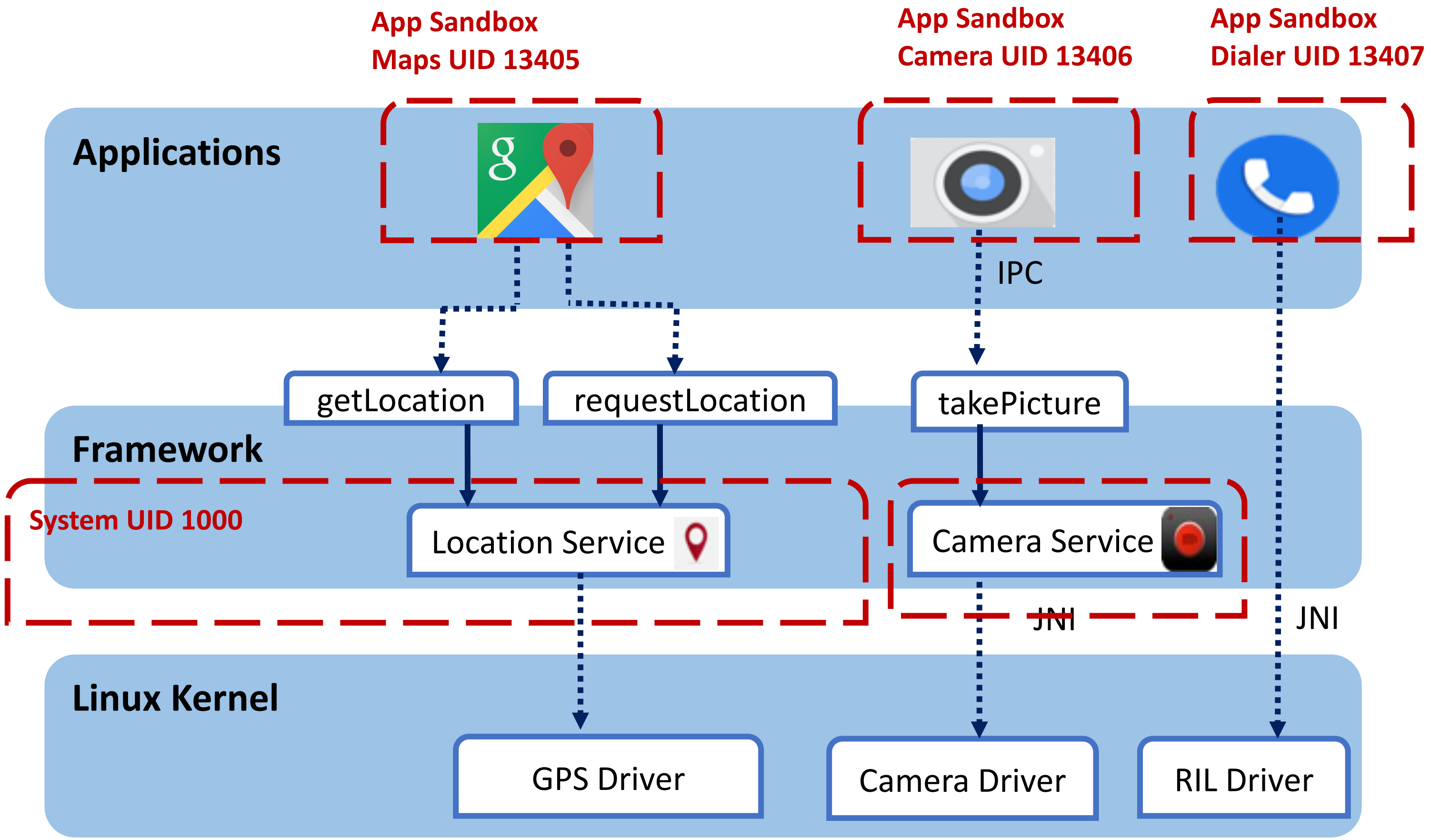
- How does Android protect various sensitive resources in the system?
  - App sandboxing
  - Access control based on permissions
  - Traditional Linux DAC

Protecting Resources in the system



Protecting Resources in the system

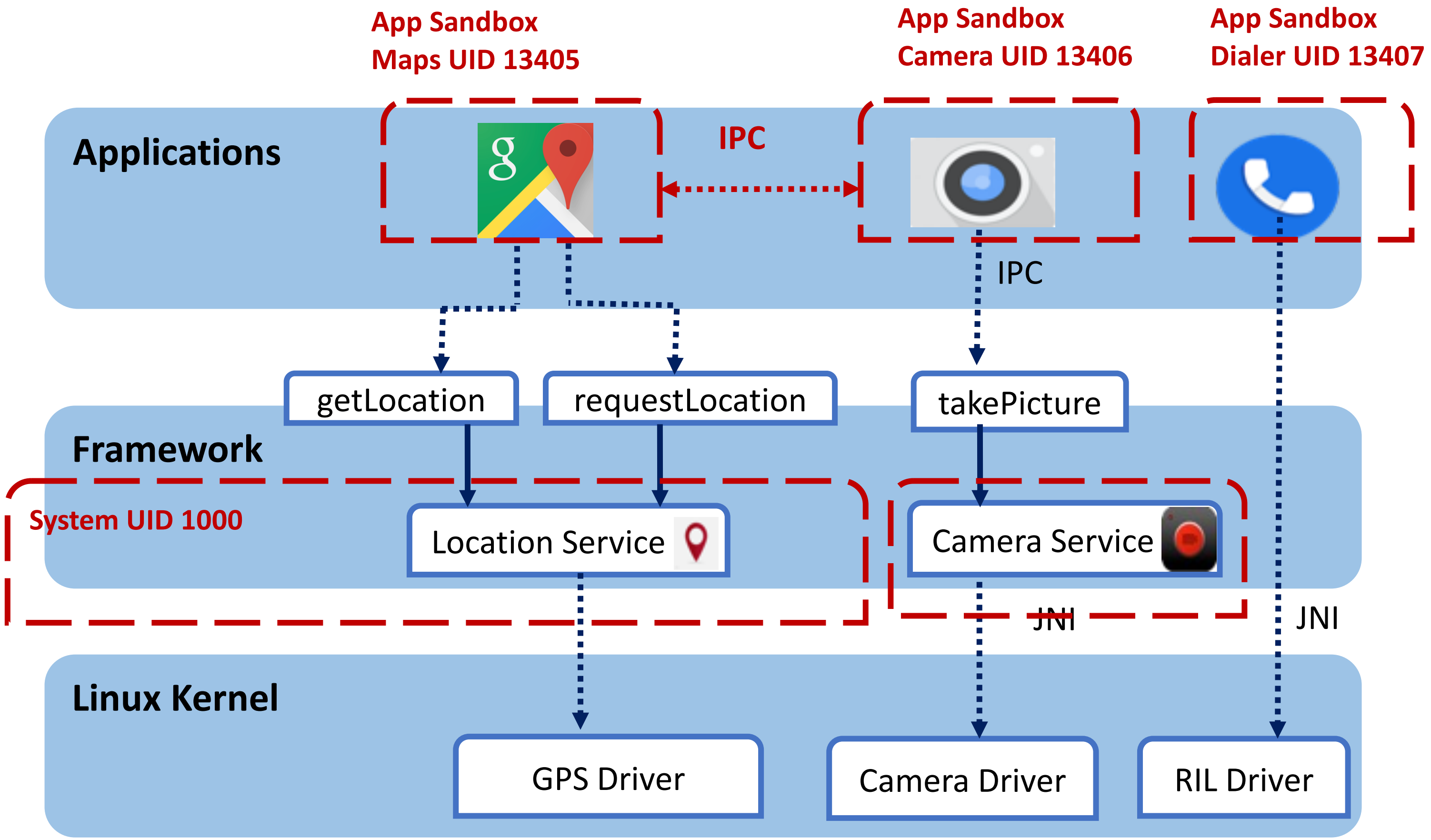
App sandboxing



- Android assigns a unique UID to each Android app and runs it in its own process
- System level processes are assigned privileged UIDs
- The UIDs are used to set up a kernel-level Application Sandbox

Protecting Resources in the system

App sandboxing



- By default, apps cannot interact with each other and have limited access to the OS
- By default, apps cannot read other apps data or invoke its functionality
- All communication goes through monitored IPC

Protecting Resources in the system

App sandboxing

- Android relies on a number of protections to enforce the application sandbox.
  - The enforcements have evolved over time to strengthen the original UID-based discretionary access control (DAC) sandbox

  - Android 5.0: SELinux provided Mandatory Access Control (MAC) separation between the system and apps
  - Android 6.0: SELinux separation was extended to isolate apps based on the running users.

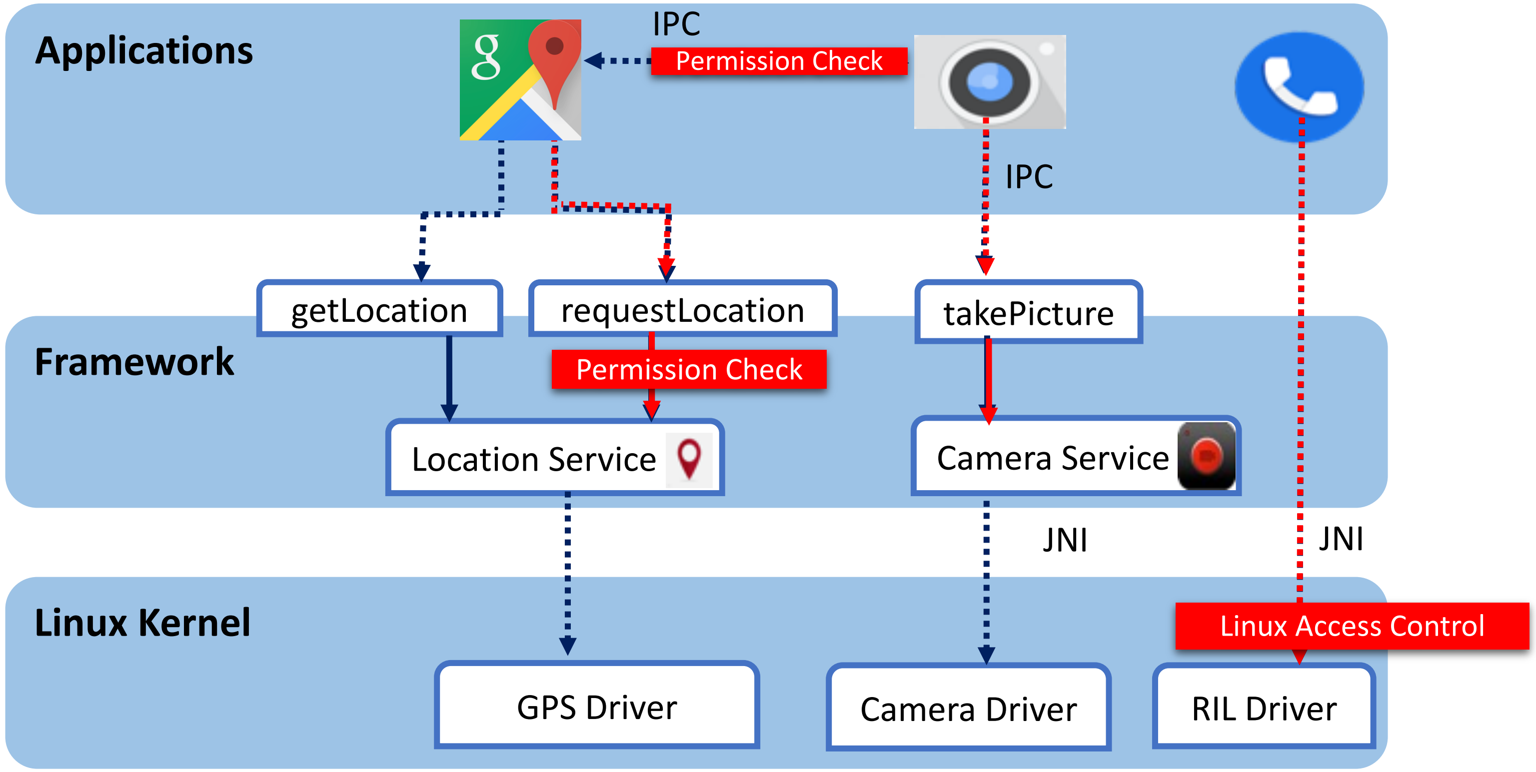
Protecting Resources in the system
App sandboxing

- Android relies on a number of protections to enforce the application sandbox.
  - The enforcements have been evolved over time to strengthen the original UID-based discretionary access control (DAC) sandbox

  - Android 9: SELinux separation was extended to provide a per-app isolation
  - Android 10: apps have a restricted raw view of the filesystem

Protecting Resources at the Linux layer

Traditional Linux ACLs



Protecting Resources at the Linux layer

Traditional Linux ACLs

- Android relies on Linux Discretionary Access Control (DAC) to protect resources at Linux layer

- Protected objects: ??
- Subjects: ??
- Rights: ??

Protecting Resources at the Linux layer

Traditional Linux ACLs

- Android relies on Linux Discretionary Access Control (DAC) to protect resources at Linux layer

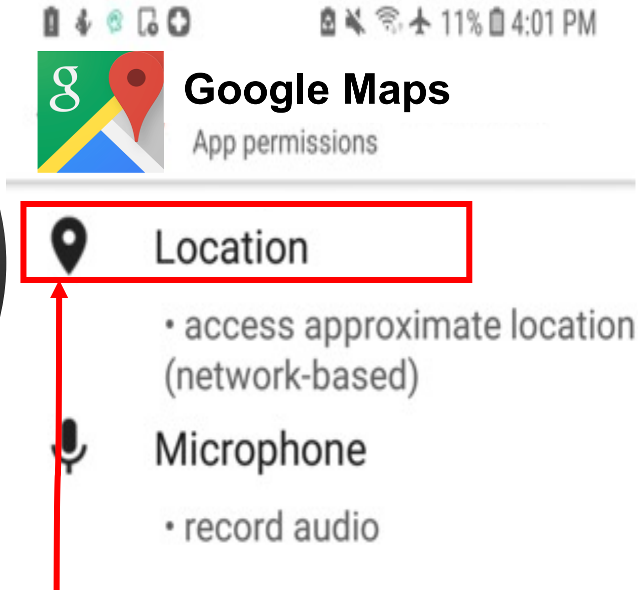
- Protected objects: Linux objects: Files (remember device drivers are special files).
- Subjects: Apps and system processes (remember each process is defined by unique UID)
- Rights: RWX

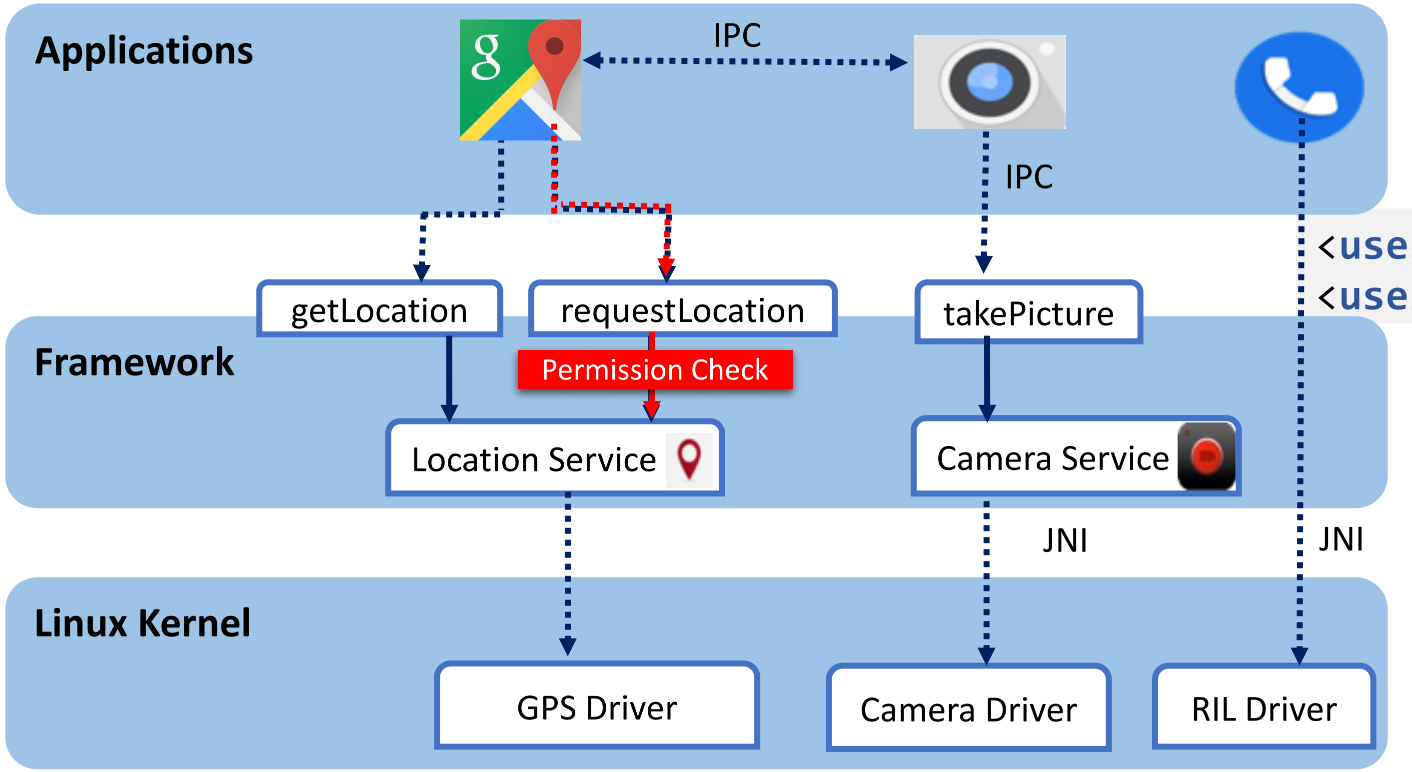
Module Outline

1. Overview of Android OS
2. Security Mechanisms
3. App Security
4. A Dive into Android Vulnerabilities and Flaws

Protecting Resources

Android Permissions

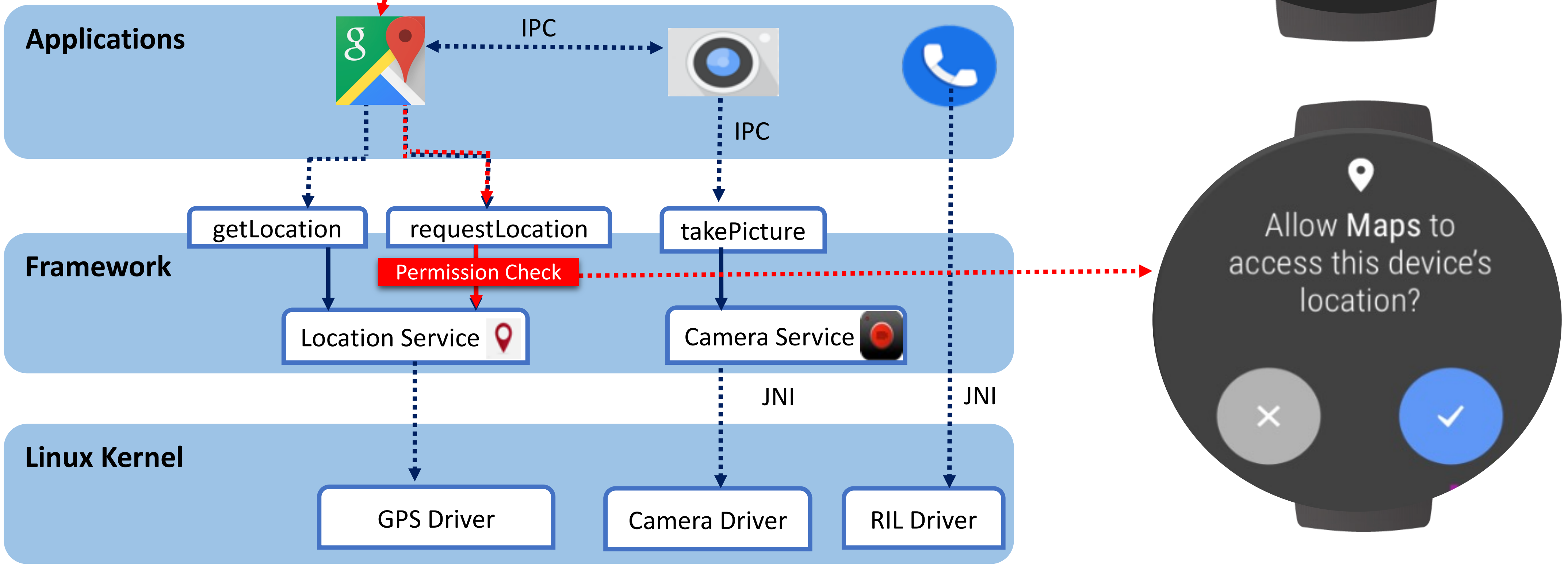





```
<uses-permission name="ACCESS_FINE_LOCATION"/>
<uses-permission name="ACCESS_COARSE_LOCATION"/>
```

Protecting Resources

Android Permissions



Protecting Resources

Android Permissions

- Permission enforcement in Android APIs

```
LocationManagerService

Location getLastLocation(LocationProvider request, …)
{
   if(caller.hasPermission(“ACCESS_FINE_LOCATION”)
        || caller.hasPermission(“ACCESS_COARSE_LOCATION”) )
   {
         …
         return mLastLocation.get(request.getProvider());
   }
   else
         // throw Security Exception
}
```

Protecting Resources
Android Permissions

- Three categories of permissions:
  - ***Install-time permissions***
  - ***Runtime permissions***
  - ***Special permissions***

- The categories indicate:
  - The scope of data that an app can access
  - The scope of functionality that an app can perform

Protecting Resources

Install-time Permissions

- Allow an app limited access to restricted data
- Allow performing actions with minimal effect on the system or on other apps

- The system grants these permissions automatically to apps during install time

- Two types:
  - Normal: Allow access to data/operations that present little risk
  - Signature: Granted to an app only when the app is signed with the same certificate as the entity (app / OS) defining the permission

Protecting Resources

Examples of install-time permissions

- ACCESS_NETWORK_STATE
- ACCESS_NOTIFICATION_POLICY
- ACCESS_WIFI_STATE
- BLUETOOTH
- BLUETOOTH_ADMIN
- BROADCAST_STICKY
- CHANGE_NETWORK_STATE
- CHANGE_WIFI_MULTICAST_STATE
- CHANGE_WIFI_STATE

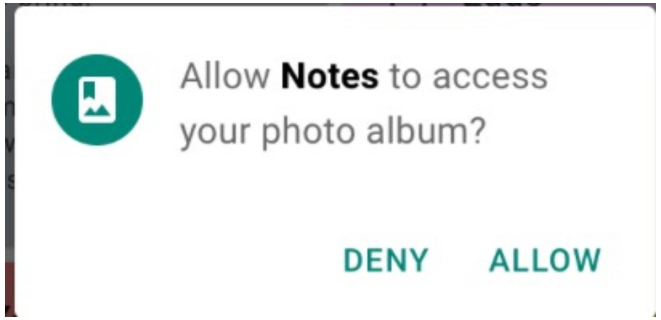
NORMAL

- BIND_AUTOFILL_SERVICE
- BIND_CARRIER_SERVICES
- BIND_CHOOSER_TARGET_SERVICE
- BIND_CONDITION_PROVIDER_SERVICE
- BIND_DEVICE_ADMIN
- BIND_DREAM_SERVICE
- BIND_INCALL_SERVICE
- BIND_INPUT_METHOD
- BIND_MIDI_DEVICE_SERVICE
- BIND_NFC_SERVICE

Signature

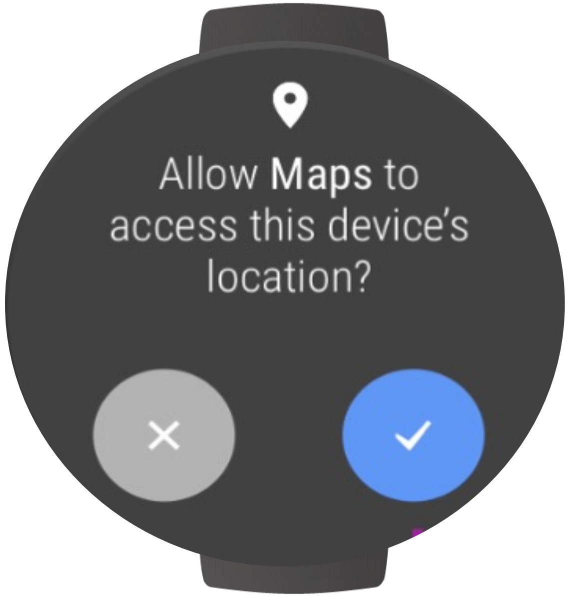
- Signature permissions aren't for use by third-party apps

Protecting Resources
Runtime Permissions



- Also known as *Dangerous permissions*
- Allow an app additional access to restricted data
- Allow performing actions with more substantial effect on the system or on other apps

- Apps need to request runtime permissions:
  - The system will present a runtime permission prompt



Protecting Resources

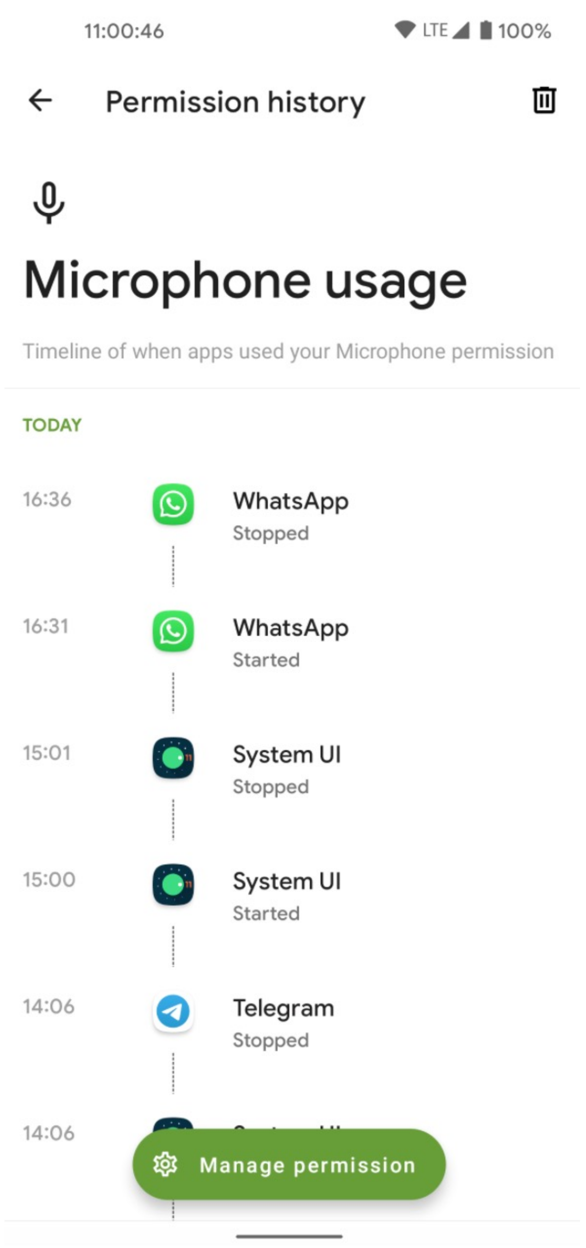
Examples of Runtime / Dangerous Permissions

- WRITE_CALENDAR
- READ_CALL_LOG
- WRITE_CALL_LOG
- PROCESS_OUTGOING_CALLS
- CAMERA
- READ_CONTACTS
- WRITE_CONTACTS
- GET_ACCOUNTS
- ACCESS_FINE_LOCATION
- ACCESS_COARSE_LOCATION

Protecting Resources

Runtime Permissions

- Location, Microphone and Camera permissions provide access to particularly sensitive information.
- Android provides mechanisms to help users be aware and monitor which apps use these permissions
- Android 12 or higher: Privacy dashboard
  - Historical view of when different apps have accesses data pertaining to these permissions
- Android 12 or higher: indicators and toggles

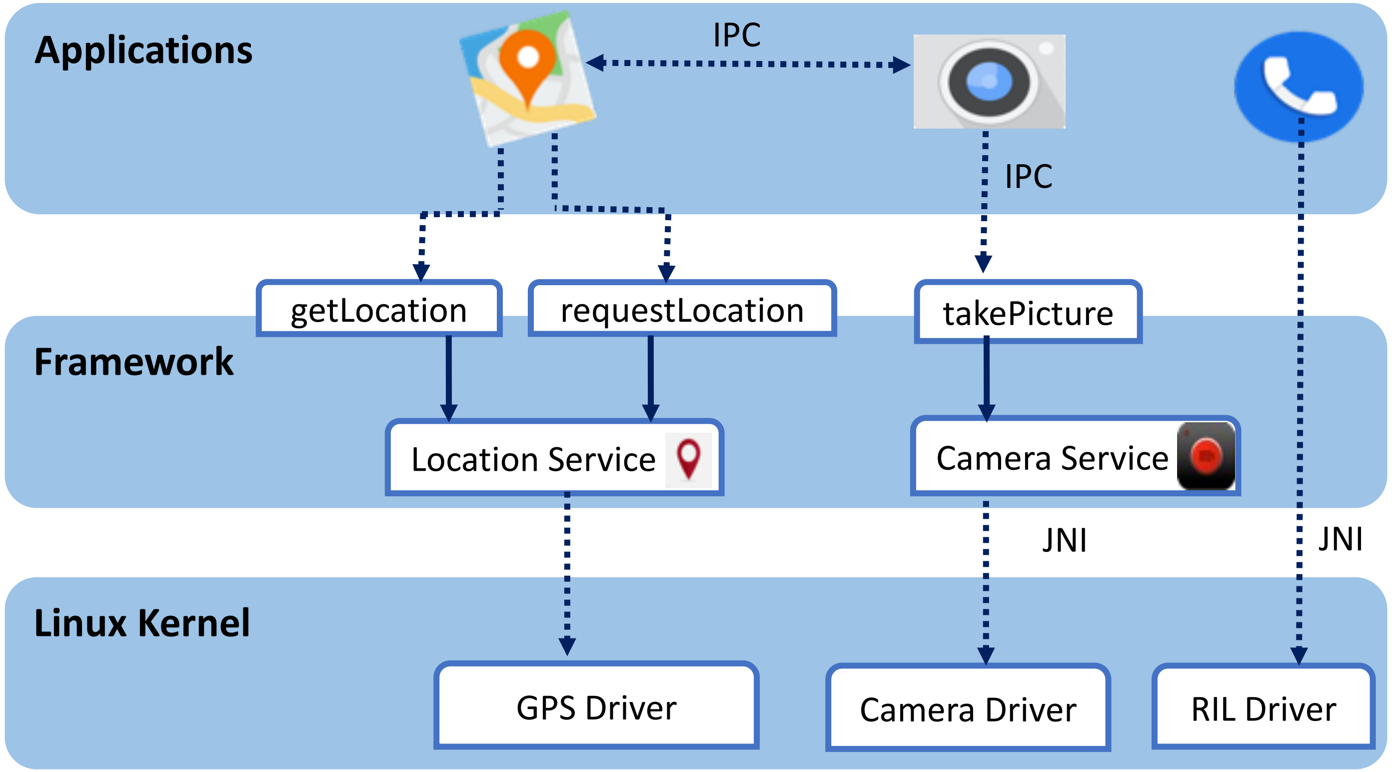


Protecting Resources
Special Permissions

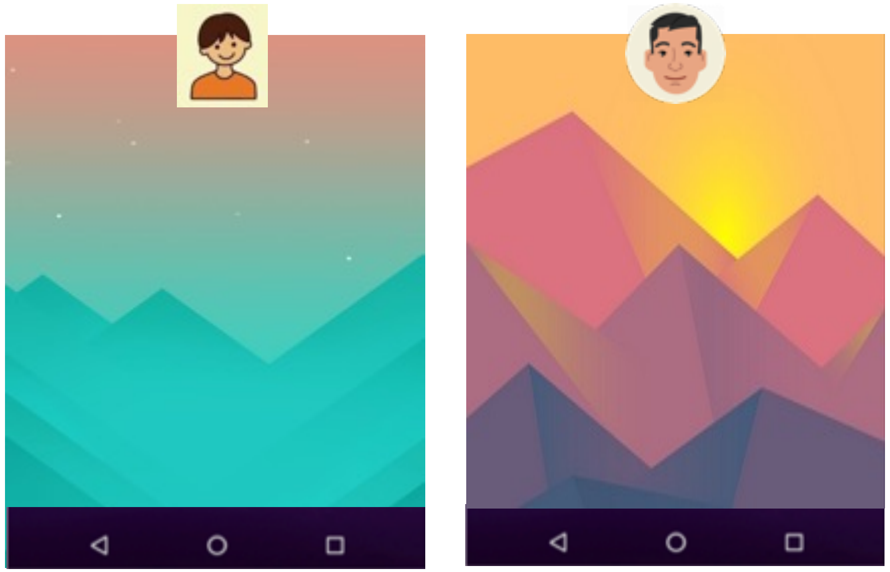
- Allow access to system resources that are highly sensitive
- Examples:
  - displaying and drawing over other apps
  - accessing all storage data
- Unlike the other categories of permissions, only the system or OEMs can define special permissions
- An app cannot obtain a special permission unless the user explicitly grants it through the Setting app.

Protecting Framework Resources

Multi-user Access Control



✓ **Multi-User Feature**

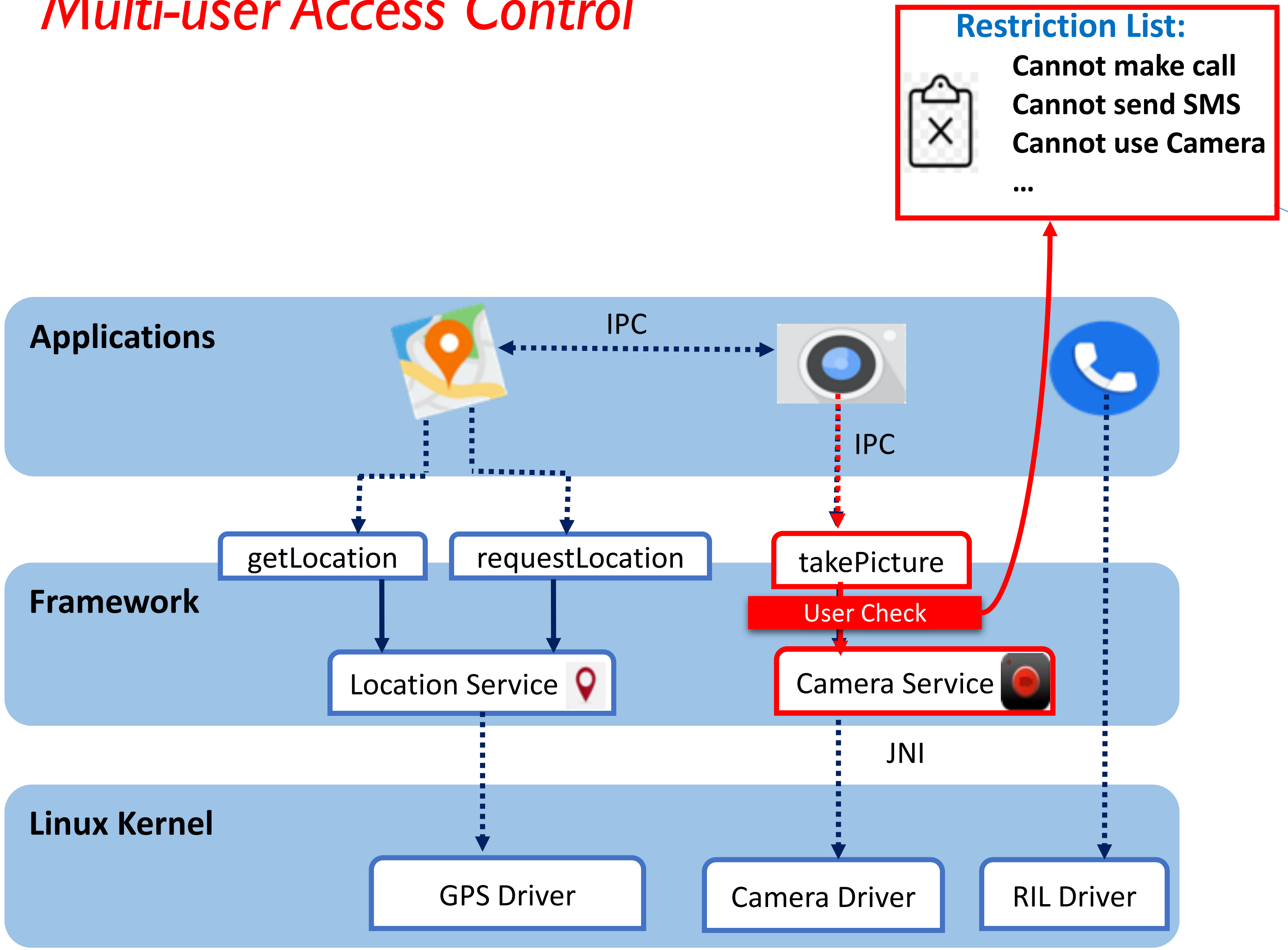


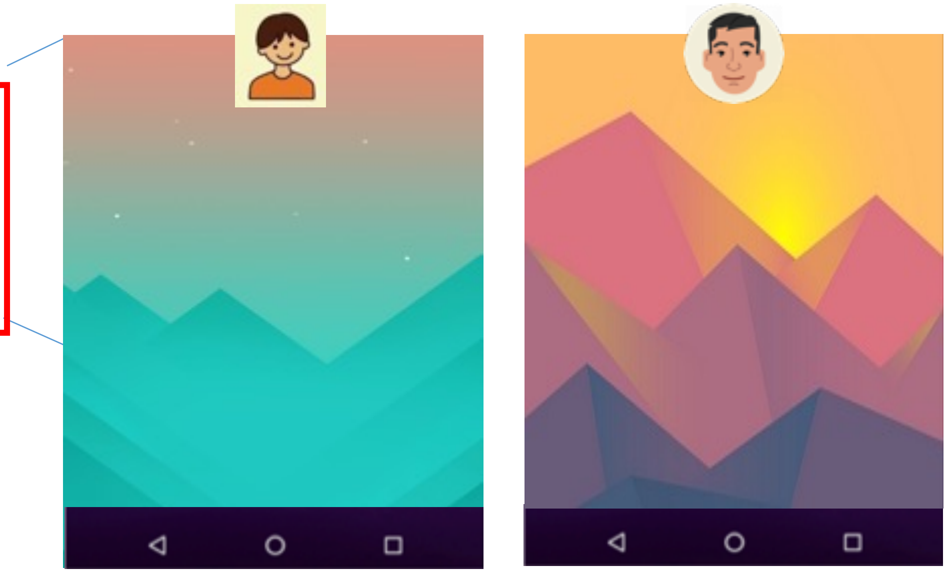
New Security Requirements

- Privilege Difference between users
- Isolation of users' apps and data

Protecting Framework Resources

Multi-user Access Control





New Security Requirements

Privilege Difference between users

Isolation of users' apps and data

Protecting Framework and Apps

Permission assignment

- Apps request permissions to access sensitive resources.
  - request android.permission.SEND_SMS to send a text message
  - request android.permission.WRITE_SECURE_SETTINGS to configure sensitive device properties
  - …
- All permissions requested / granted to an app are assigned to the app's *UID*

Protecting Framework and Apps

Permission assignment

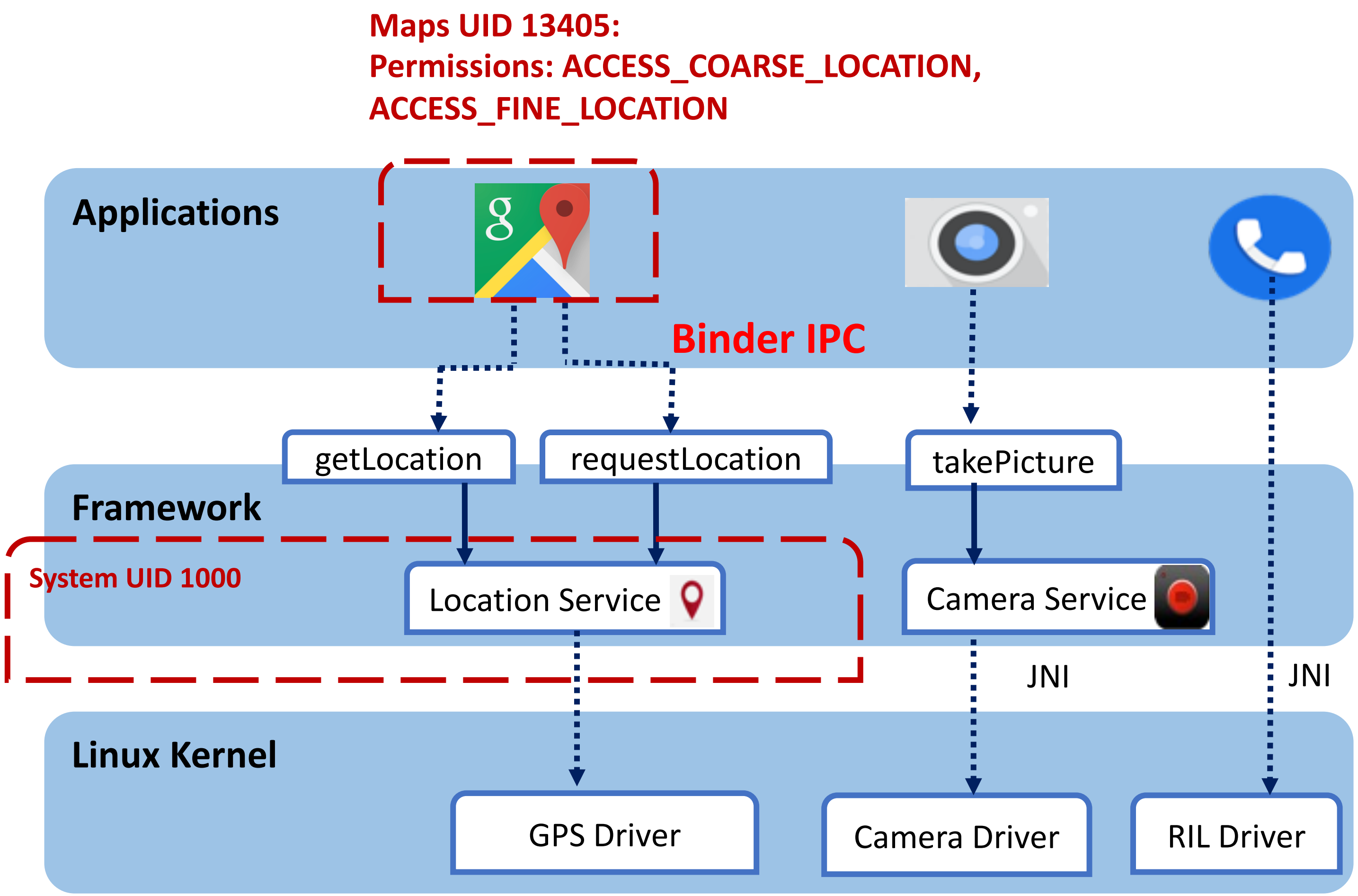
- All permissions requested / granted to an app are assigned to the app's *UID*
- *Examples*

```
Package [com.google.android.apps.docs] (9e13ae4):
  userId=10186
  pkg=Package{7af35a4 com.google.android.apps.docs}
  codePath=/product/app/Drive
 install permissions:
   android.permission.DOWNLOAD_WITHOUT_NOTIFICATION: granted=true
   com.google.android.c2dm.permission.RECEIVE: granted=true
   android.permission.USE_CREDENTIALS: granted=true
   com.google.android.providers.gsf.permission.READ_GSERVICES: granted=true
   android.permission.MANAGE_ACCOUNTS: granted=true
   com.google.android.googleapps.permission.GOOGLE_AUTH.OTHER_SERVICES: granted=true
   android.permission.NFC: granted=true
   com.google.android.googleapps.permission.GOOGLE_AUTH.writely: granted=true
   android.permission.FOREGROUND_SERVICE: granted=true
   android.permission.WRITE_SYNC_SETTINGS: granted=true
   android.permission.RECEIVE_BOOT_COMPLETED: granted=true
```

- *A UID that is assigned to an app remains unchanged while the app is installed, running, and updated on a device*
- *A PID (process ID) can change*

Protecting Framework and Apps

Permission assignment



- System service APIs enforce access control.
- How does the framework trace the calling UID?
- ***Through Binder IPC mechanism***

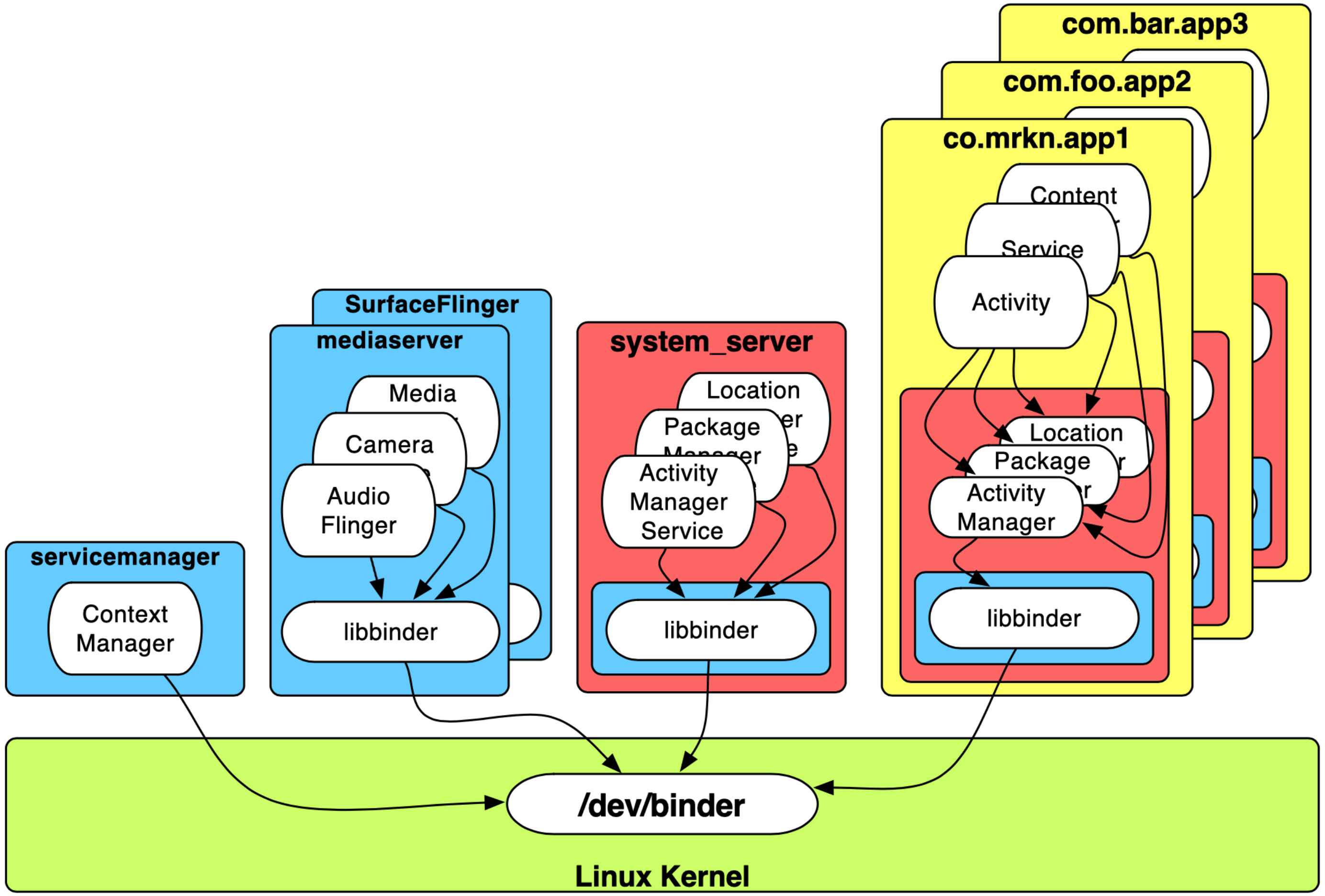
Binder IPC (Inter-Process Communication)

- Binder: A core part of a lightweight remote procedure call mechanism
- Android apps communicate with the framework system services via binder IPC interface
- Android apps *can also communicate with each other* via binder IPC
- Binder IPC enables *information sharing while ensuring:*
  - *Privilege Separation*
  - *Stability*

Binder IPC (Inter-Process Communication)

- Essential to Android
- Originally from OpenBinder
  - First implementation used in Palm Cobalt
  - Binder was ported to Linux and open sourced in 2005
  - Completely rewritten for Android in 2008
- Its design focuses on scalability, stability, flexibility, low-latency/overhead, easy programming model

Binder IPC (Inter-Process Communication)



Binder IPC (Inter-Process Communication)

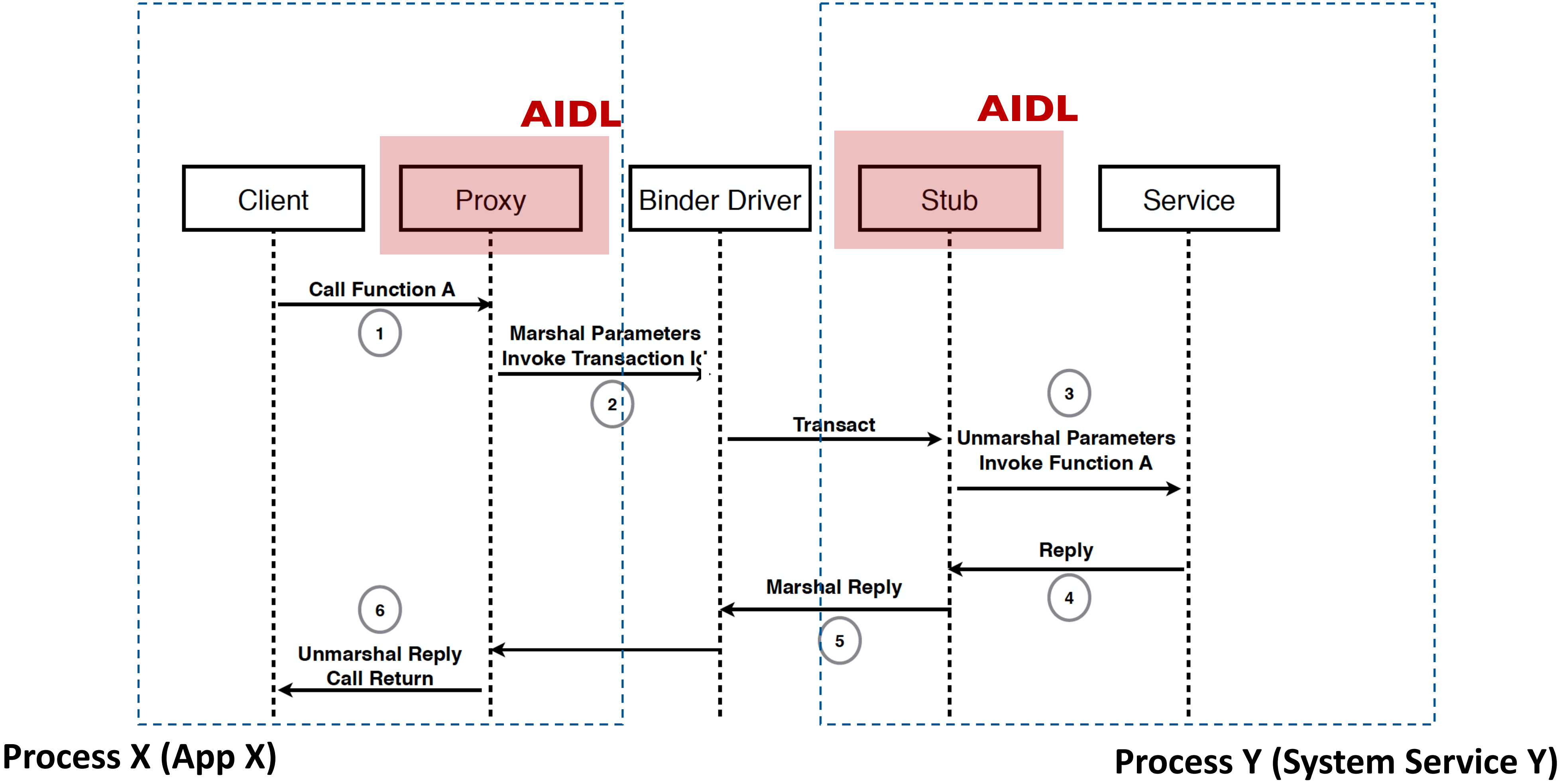
- Why Binder IPC specifically?
  - Death notification mechanism
  - Owners of binder services are notified when no longer referenced
  - Automatic management of thread pools
  - synchronous and asynchronous invocation models

Binder IPC (Inter-Process Communication)

- Why Binder IPC specifically?
  - Follows a simple programming interface that clients and services agree upon for communication
  - Android Interface Definition Language (AIDL)
  - APIs in remote service objects -- defined in the interface, can be invoked as if local.

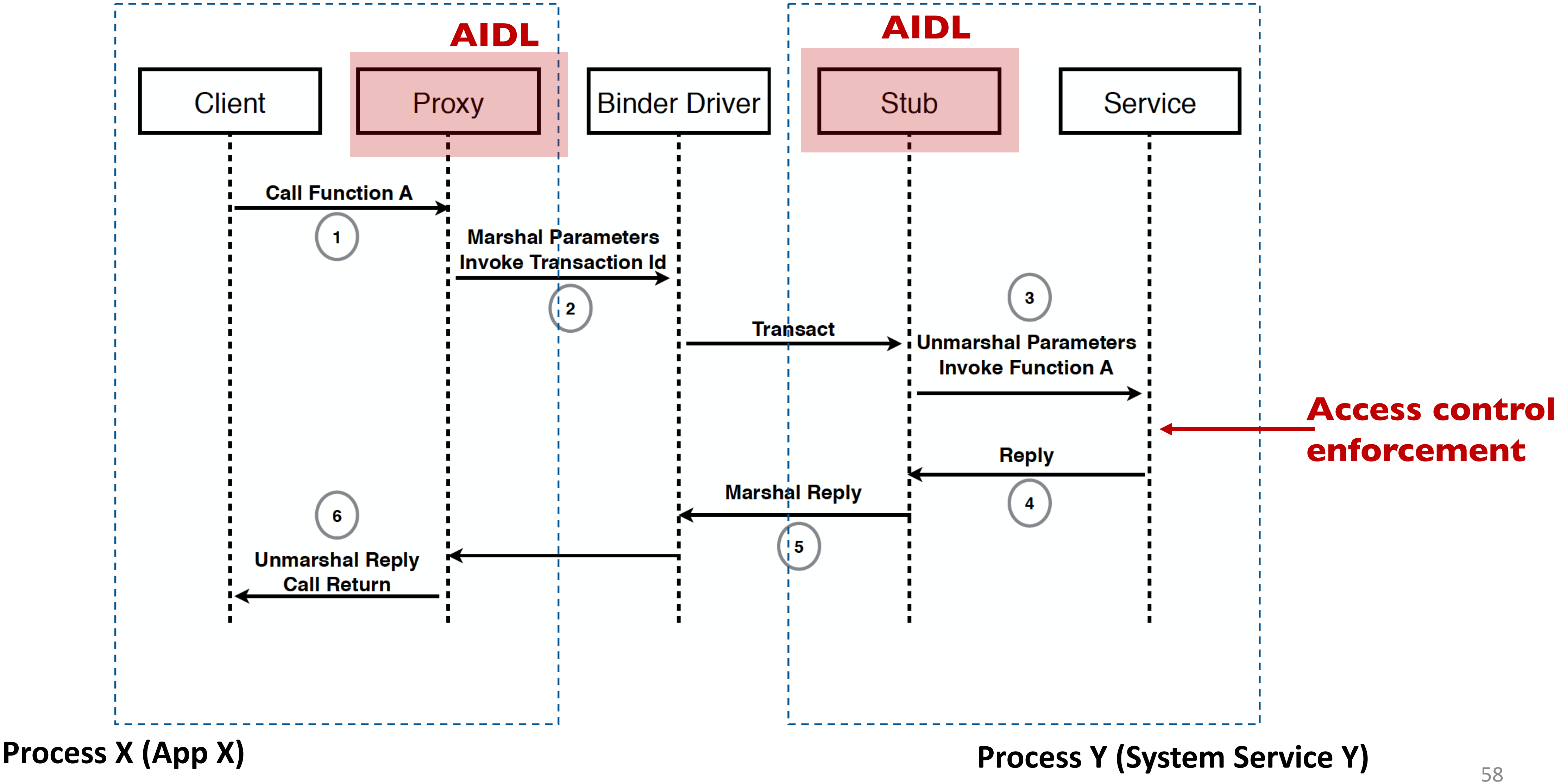
Binder IPC (Inter-Process Communication)

Remote Binder Transactions



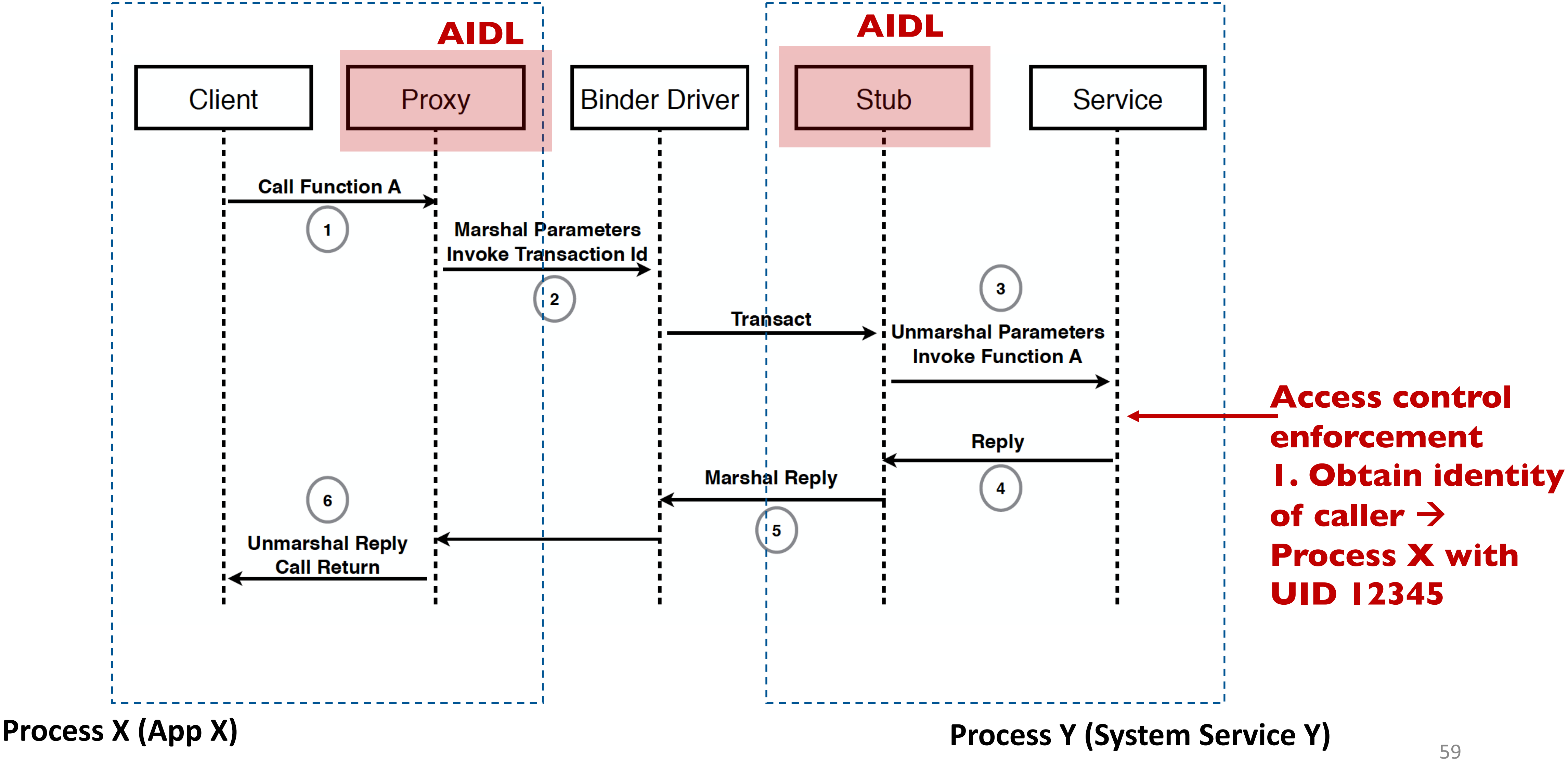
Binder IPC (Inter-Process Communication)

Remote Binder Transactions



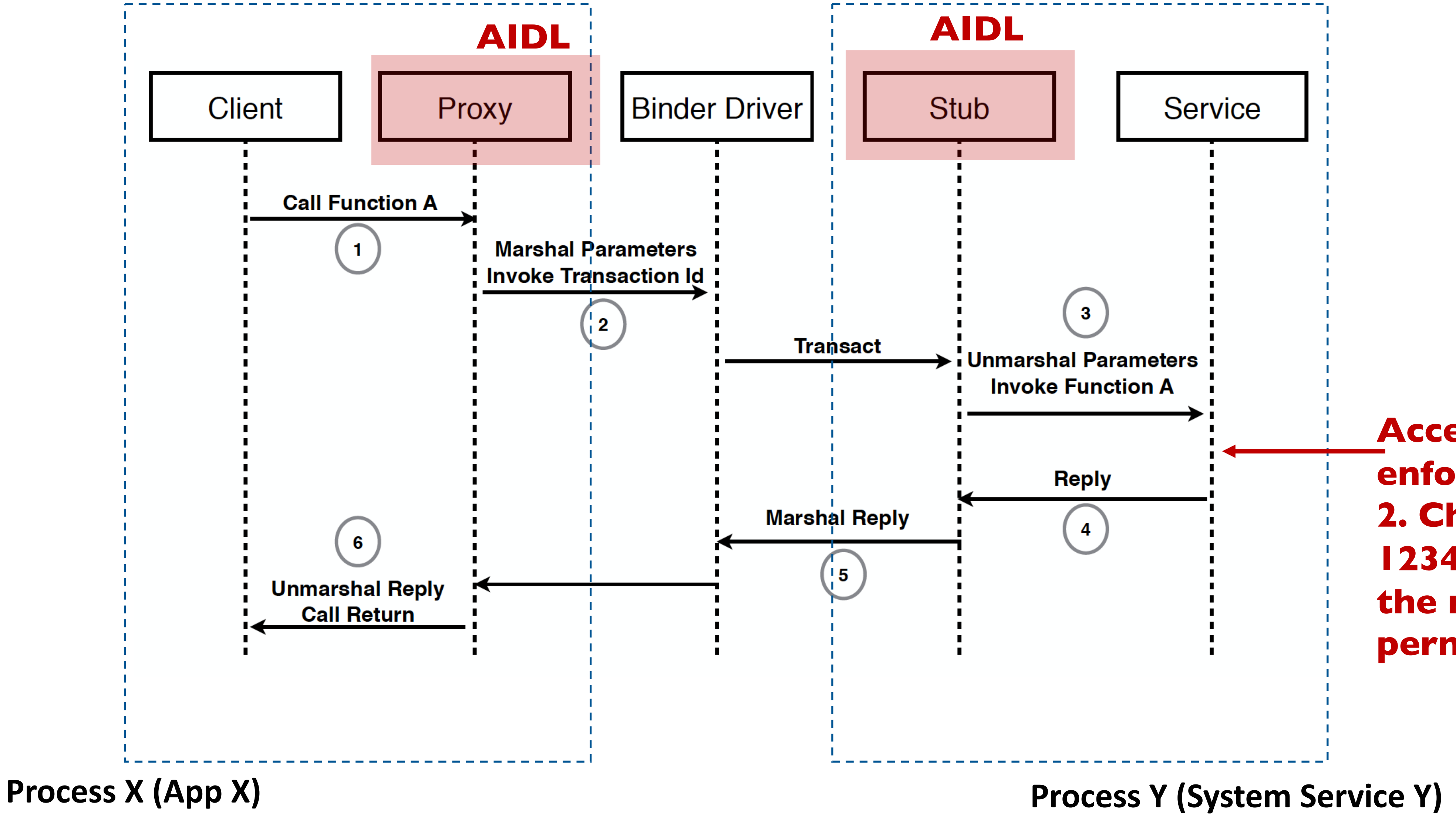
Binder IPC (Inter-Process Communication)

Remote Binder Transactions



Binder IPC (Inter-Process Communication)

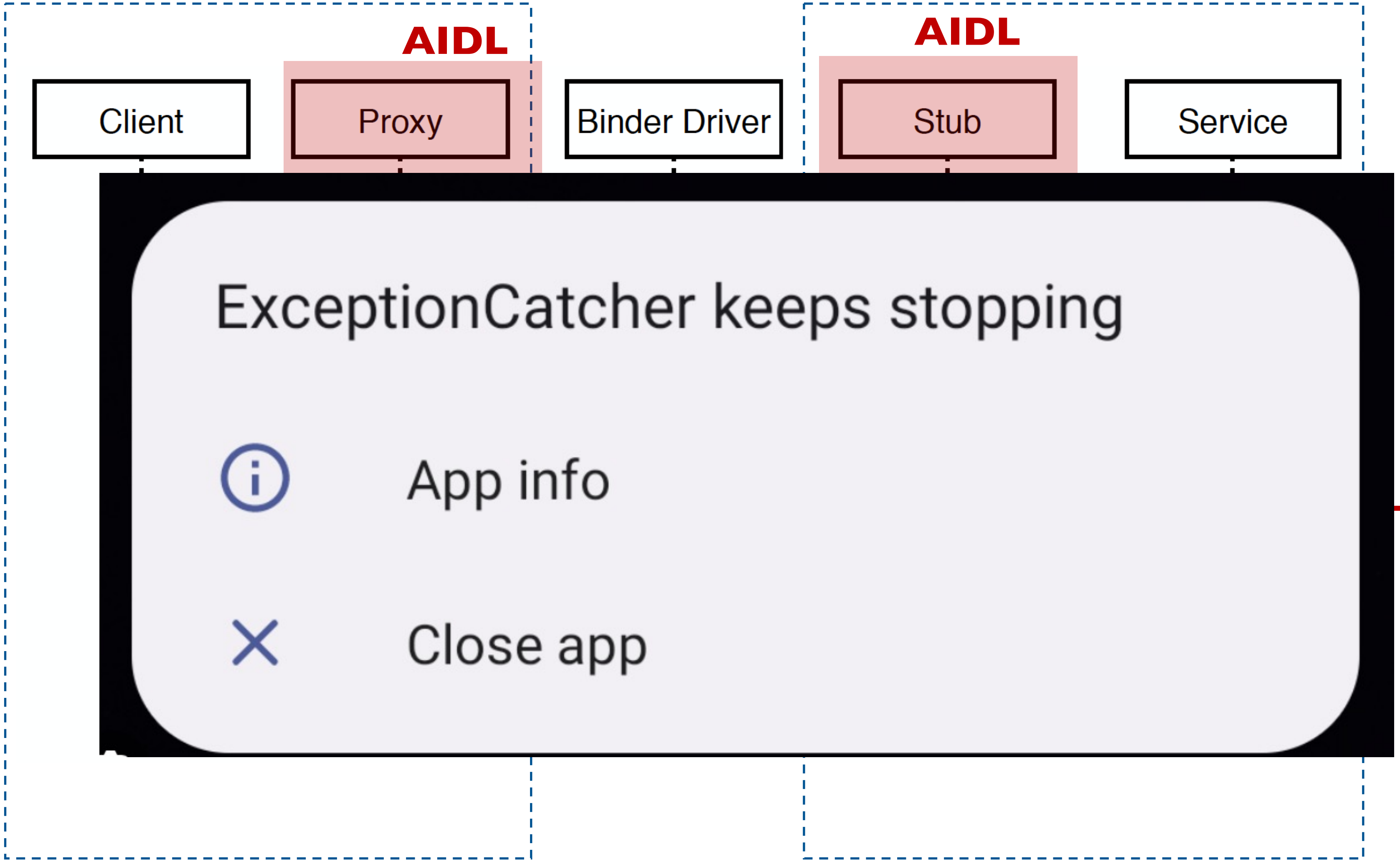
Remote Binder Transactions



Access control enforcement
2. Check if UID 12345 is assigned the required permission

Binder IPC (Inter-Process Communication)

Remote Binder Transactions



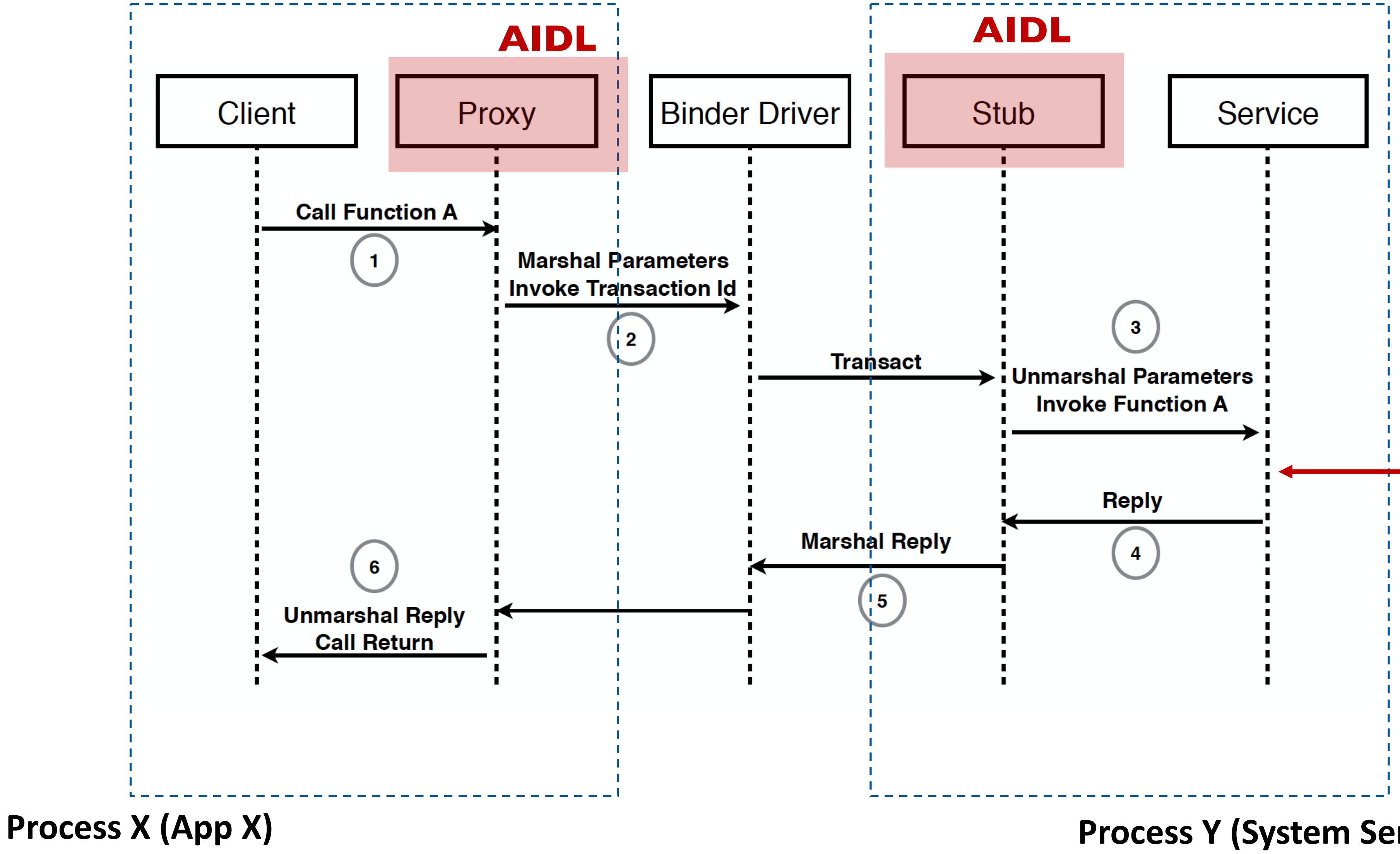
Access control enforcement
If not, the transaction will be interrupted

Process X (App X)

Process Y (System Service Y)

Binder IPC (Inter-Process Communication)

Remote Binder Transactions

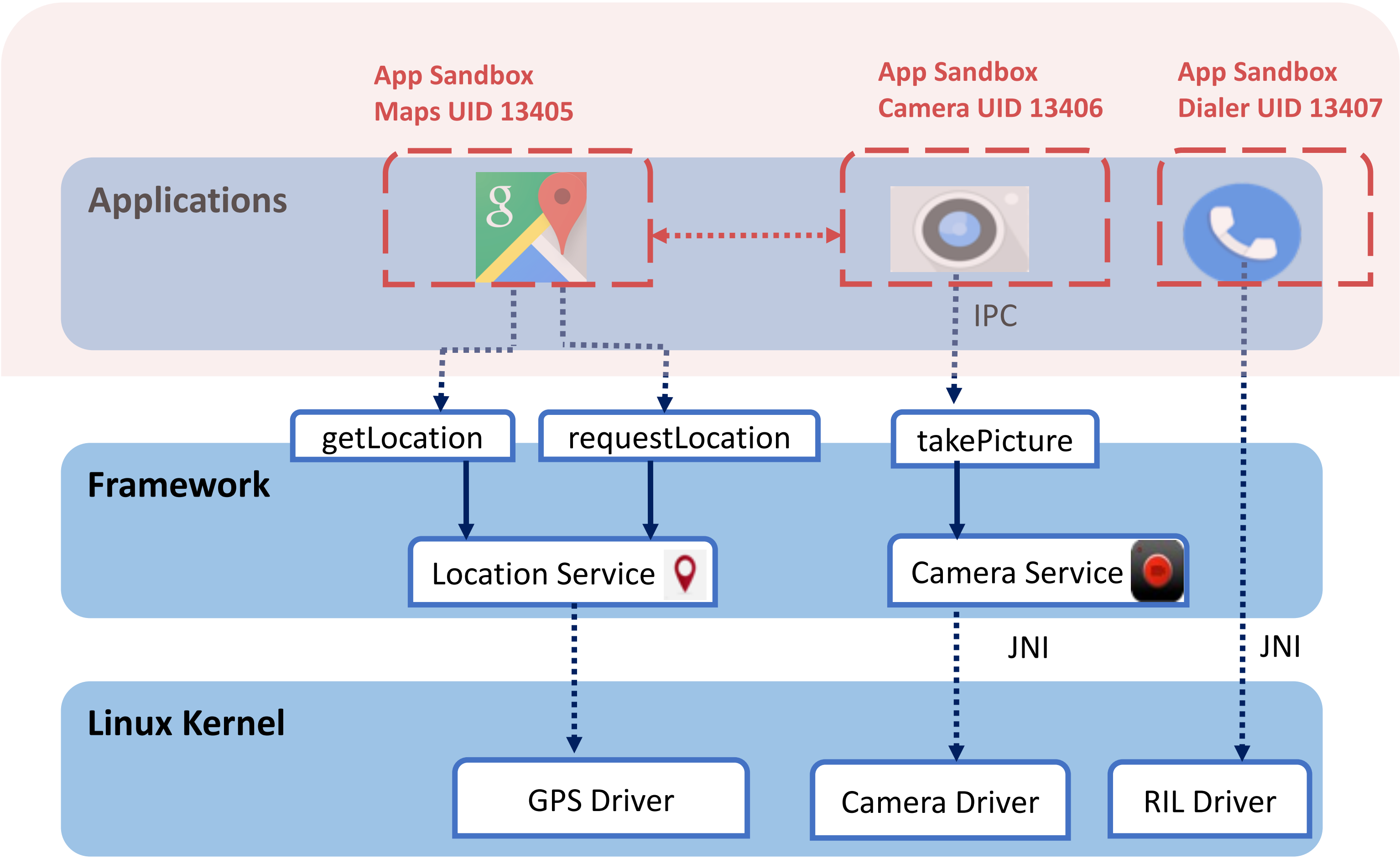


Access control enforcement
Otherwise, access to A's functionality is granted

Binder IPC (Inter-Process Communication)

- Why Binder IPC specifically? Other Security reasons
  - Identify UIDs (and PIDs) of senders and receivers
  - Unique token for an object across boundaries

Protecting Apps



- By default, apps cannot interact with each other.
- By default, apps cannot read other apps data or invoke its functionality
- Android allows sharing between apps via different forms of inter-app communication

Protecting Apps
Inter-App Communication

- Some app might not request permissions to access a sensitive resource or perform a privilege operation
  - Rather, they *can delegate this job to other apps.*

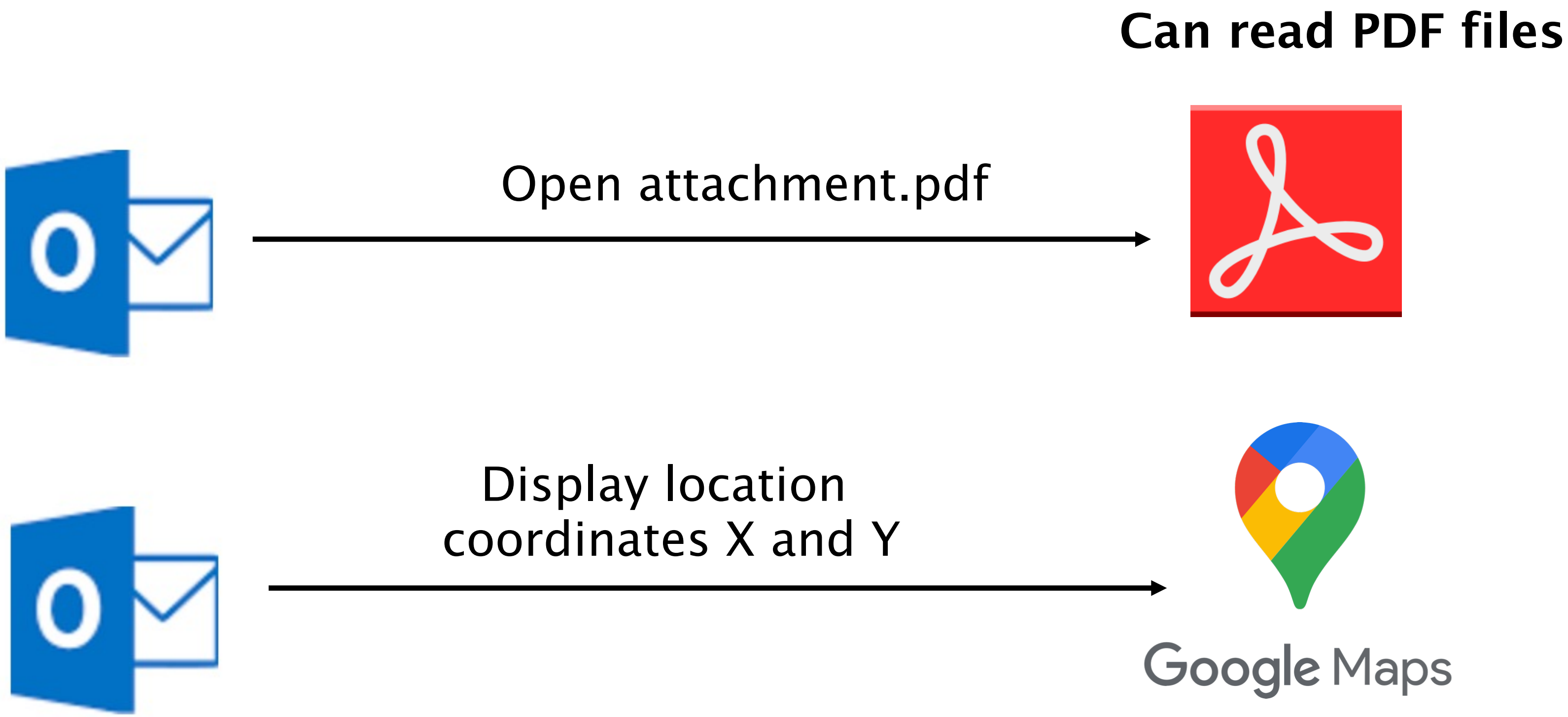
- Functionality sharing/reuse is highly encouraged in Android

- Functionality sharing/reuse occurs through app-level interactions

Inter-app communication

Motivating examples

- Functionality sharing/reuse



Inter-app communication

Available Mechanisms

- Android apps can communicate with each other via different mechanisms:
  - Use traditional Linux mechanisms such as shared files, pipes, etc.
  - Use Android specific mechanisms:
    - Binder IPC
    - Intents
    - Messenger
    - Content Providers

IPC via Intents

- Android supports a simple form of IPC via Intents
- Intents are messaging objects that can be used by an app to request an action from another app component

- Interaction between apps is done at their level of components

IPC via Intents

- Android supports a simple form of IPC via Intents
- Intents are messaging objects that can be used by an app to request an action from another app component

- Interaction between apps is done at their level of components
    - Start Activities
    - Start Services
    - Delivering Broadcasts

IPC via Intents

- Intents pass a messaging object from a calling app to another app
- Steps:

1. An app needs to declare that it can handle a specific functionality
    - PDF Viewer app can declare that it can open / display pdf files
    - Google Maps app can declare that I can allow displaying a specific coordinate on the app

IPC via Intents

- Intents pass a messaging object from a calling app to another app
- Steps:

1. An app needs to declare that it can handle a specific functionality
   - PDF Viewer app can declare that it can open / display pdf files
   - Google Maps app can declare that I can allow displaying a specific coordinate on the app
2. Other apps will send intents to apps that can handle the functionality

IPC via Intents

- Intents pass a messaging object from a calling app to another app

1. Declare the ability to handle pdf viewing

```
<activity android:name=".FileViewer">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <data android:mimeType="application/pdf" />
    </intent-filter>
</activity>
```

IPC via Intents

- Intents pass a messaging object from a calling app to another app

2. Send intent to pdf viewer

1. Declare the ability to handle pdf viewing

```java
Intent intent = new Intent();
intent.setAction("android.intent.action.VIEW");
intent.setType("application/pdf");
intent.setData(Uri.parse("content://email/attachment/file.pdf"));
startActivity(intent);
```

```xml
<activity android:name=".FileViewer">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <data android:mimeType=“application/pdf" />
    </intent-filter>
</activity>
```

IPC via Intents

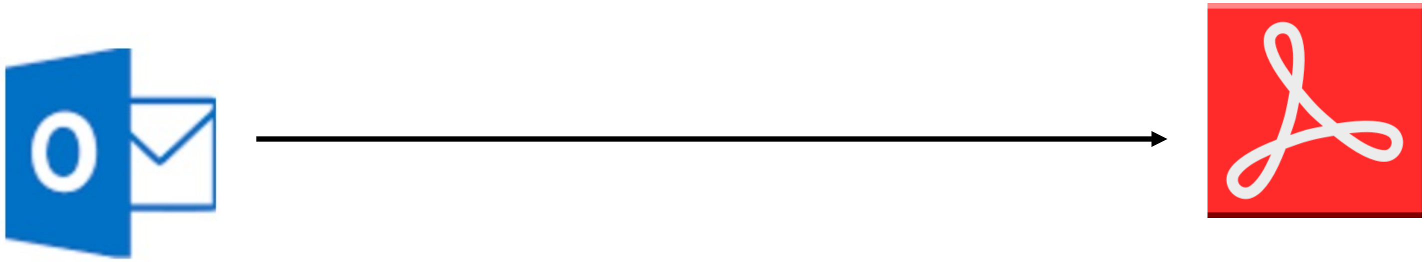
- There are two types of intents in Android:

1. Explicit intents
2. Implicit intents

IPC via Intents

- There are two types of intents in Android:

1. Explicit intents
    - Specify the target app component that should handle the intent



```
Intent intent = new Intent();
Intent.setComponent("com.adobe.FileViewer");
```

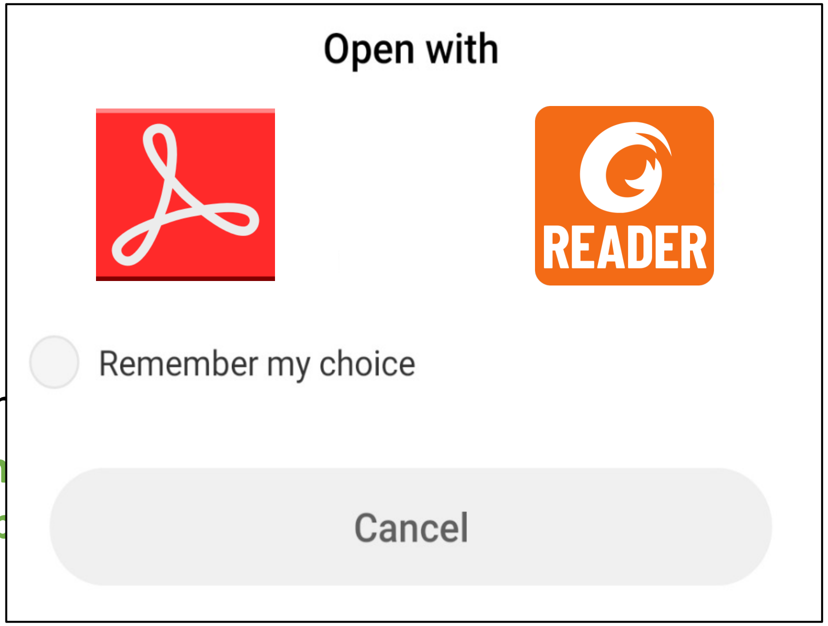
IPC via Intents

2. Implicit intents
    - The target app component is not specified
    - The action to be performed is specified

```
Intent intent = new Intent();
Intent.setAction("android.intent.action.VIEW");
intent.setType("application/pdf");
```

IPC via Intents

2. Implicit intents
   - The target app component is not specified
   - The action to be performed is specified
   - The Android OS will resolve the components that can handle the request
     - If more than one, the user may get to pick his preferred target
     - Sometimes, the target is selected automatically




```
Intent intent = new Inter
Intent.setAction("android.in
intent.setType("application/p
```

App components

- App components are the building blocks of an Android app.
- Each component is an entry point to the app, through which the system or other apps can access the app.

- Components are defined in the app Manifest
- *AndroidManifest.xml*
  - describes information about the app
  - defines the components using a specific syntax
  - the set of permissions that the app needs to get access to the resources
  - …

App components

- *AndroidManifest.xml*

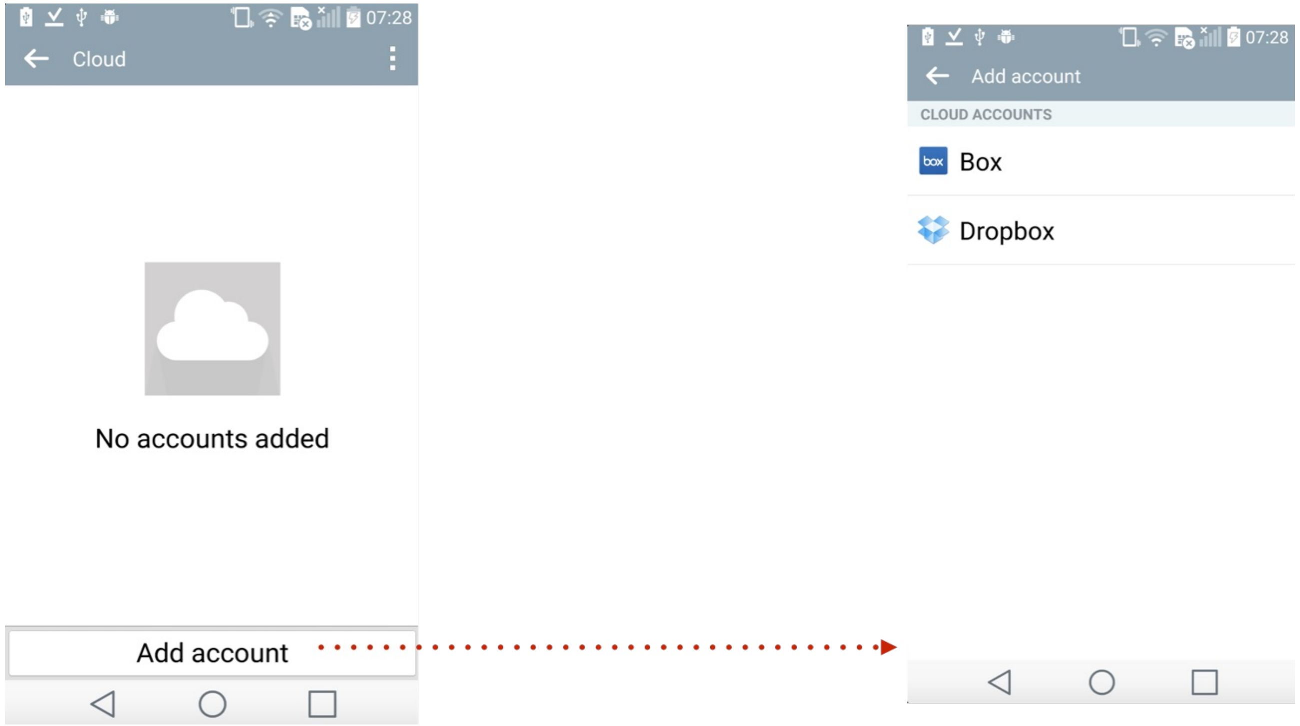
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
     package="com.wujeng.data.android"
     android:versionCode="1"
     android:versionName="1.0">

    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".ControllerActivity"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name=".StartupIntentReceiver">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED" />
                <category android:name="android.intent.category.HOME" />
            </intent-filter>
    </receiver>
    <service android:name=".DataService"
             android:exported="true"
             android:process=":remote">
    </service>
    </application>
    <uses-sdk android:minSdkVersion="10" />
    <uses-permission android:name="android.permission.INTERNET">
    </uses-permission>
</manifest>
```

App components

- Four types:
  - Activities
  - Services
  - Broadcast receivers
  - Content providers

App components
Activity

- An Activity represents a one user task; a single screen with a user interface
- Examples: Cloud info screen, add account screen



App components

Service

- A Service is a background processing component that runs long-running operations.
- A service does not provider a user interface
- Bound services category: a component can bind to it to interact with it and even perform interprocess communication (IPC).
- Examples: screen savers, notification listeners, music player

App components
Service

- A Service is a background processing component that runs long-running operations.

- Declaring a Service in AndroidManifest.xml

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".ExampleService" />
      ...
  </application>
</manifest>
```

App components
Broadcast receiver

- Receiver is a specialized event handler that allows apps to listen and respond to system-wide events broadcasts.

- Many broadcasts originate from the system
  - Battery is running low
  - The system has completed booting up
  - The screen is turned off

App components

Broadcast receiver

- Receiver is a specialized event handler that allows apps to listen and respond to system-wide events broadcasts.

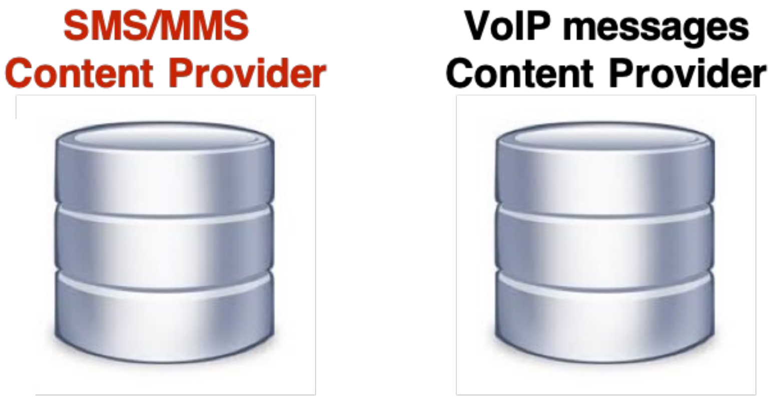
- Registering a Broadcast Receiver in AndroidManifest.xml

```
<receiver android:name=".MyBroadcastReceiver" android:exported="false">
    <intent-filter>
        <action android:name="ACTION_BATTERY_CHANGED" />
    </intent-filter>
</receiver>
```

- Broadcasts can also be registered programmatically

App components
Content providers

- Content Provider: Database wrapper; stores and manages application data.
- Standard interface that connects data in one process with code running in another process
  - e.g., SMS content provider, Contacts content provider.
- Accessing / operating on data stored in a content provider is performed through CRUD APIs



App components

Content providers

- Defining a content provider in the Manifest

```
<provider
          android:name="com.android.SMSContentProvider"
          android:authorities="sms"
          android:enabled="true"
          android:exported="true">
</provider>
```

- The content provider can be accessed using Content URIs

- Example URIs: **content://sms/inbox_sms; content://sms/outbox_sms**

Protecting app components

- Why should Android protect app components?

Send SMS on my behalf

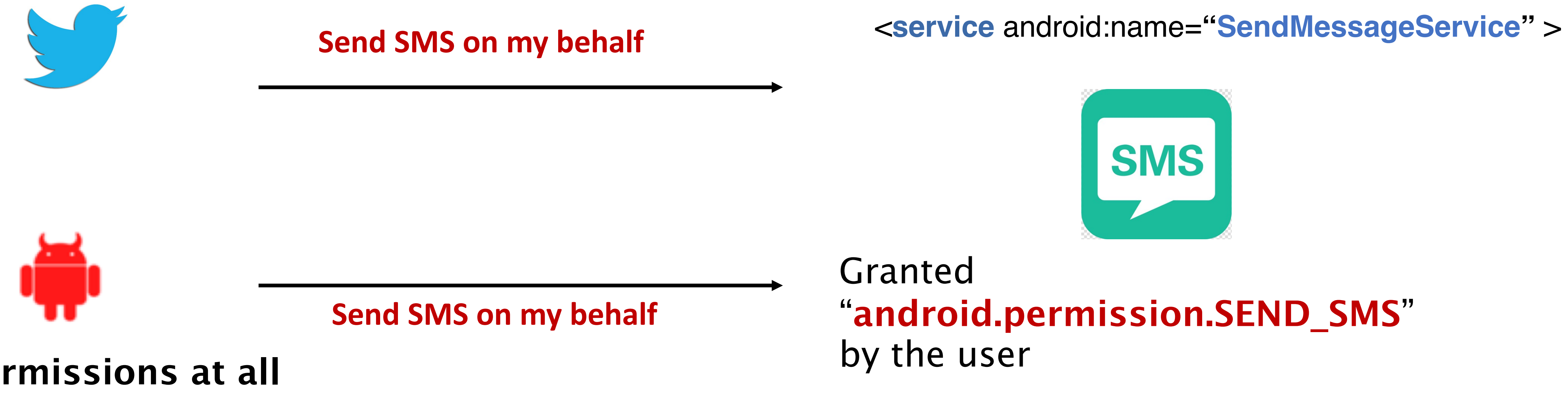
<**service** android:name=“**SendMessageService**” >

Granted
“**android.permission.SEND_SMS**”
by the user

```
Intent intent = new Intent();
Intent.putExtras(SMSMessage);
Intent.setComponent(“SendMessageService”);
startService(intent);
```

Protecting app components

- Why should Android protect app components?



Protecting app components

- Android provides various security mechanisms to protect app components:
- Enforced at Manifest declaration of components
  - Exported Flag
  - Permissions
  - Broadcasts-specific protection: protected broadcasts

Protecting app components

- Android provides various security mechanisms to protect app components:
- Enforced at Manifest declaration of components
  - Exported Flag
  - Permissions
  - Broadcasts-specific protection: protected broadcasts
- Programmatic
  - Permissions
  - …

Protecting app components

Exported Flag

- Setting exported flag to false ensures that a sensitive app component is only accessible to the defining app.

```
<receiver android:name=".MyBroadcastReceiver" android:exported="false">
    <intent-filter>
        <action android:name="ACTION_BATTERY_CHANGED" />
    </intent-filter>
</receiver>
```

Protecting app components

Permissions

- Apps can use permissions to protect components
  - A calling app needs to request / be granted that permission to access the component

- Activities, services and broadcast receivers can declare a “android:permission” element at the component definition

Protecting app components

Permissions

- Apps can use permissions to protect components
  - A calling app needs to request / be granted that permission to access the component
- Activities, services and broadcast receivers can declare a "android:permission" element at the component definition
- Content Providers can further declare "android:readPermission", "android:writePermission".
- Permissions can be either standard Android permissions or custom permissions defined by the apps

Protecting app components

- Why should Android protect app components?

Send SMS on my behalf

<service android:name="SendMessageService" >

Send SMS on my behalf

No permissions at all

Granted "android.permission.SEND_SMS" by the user

Protecting app components

Permissions

- Apps can use permissions to protect sensitive components

Add Permission requirement!!

<service name=“SendMessageService”
android:permission = “android.permission.SEND_SMS” >

Granted
“**android.permission.SEND_SMS**”

SMS

Granted
“**android.permission.SEND_SMS**”
by the user

No permissions at all

Protecting app components

Protected broadcasts

- Apps can use protected broadcasts to protect receivers
  - Only the system can send a protected broadcast
- This is important when triggering the receiver is expected to be done only by the system.
  - For example, only the system should inform apps that the phone has finished booting, that battery is running low, etc.

Protecting app components

Protected broadcasts

- The system reserves certain broadcast actions
  - Only the system can send protected broadcast actions

```
<receiver android:name="masterClear" >
 <intent-filter>
      <action android:name="MASTER_CLEAR" />
</intent-filter>
</receiver>
```


```
<protected-broadcast android:name=" MASTER_CLEAR" />
```

Advanced Topics

Permission Maps & Access Control Anomalies

Research Trends in Mobile Security

- Framework Security
  - Access control evaluation
  - Access control enhancement
- App Security
  - App-Specific Vulnerabilities
  - Access Control and permission analysis
  - Malware detection
- User Authentication
  - Biometric authentication
- Covert channels
  - …

Android Access Control Analysis

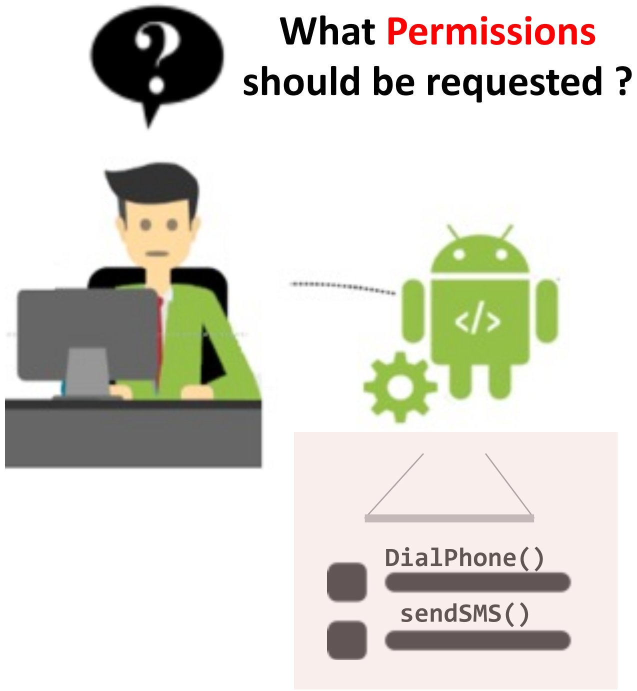
Permission Maps Extraction

Framework Security
Access Control Analysis

- Motivation
  - Lack of an understanding of Android Access Control
  - Incomplete / Missing security documentation and specification
  - Highly customized ecosystem
- This could lead to:
  - Access control anomalies
  - Potential vulnerabilities !!

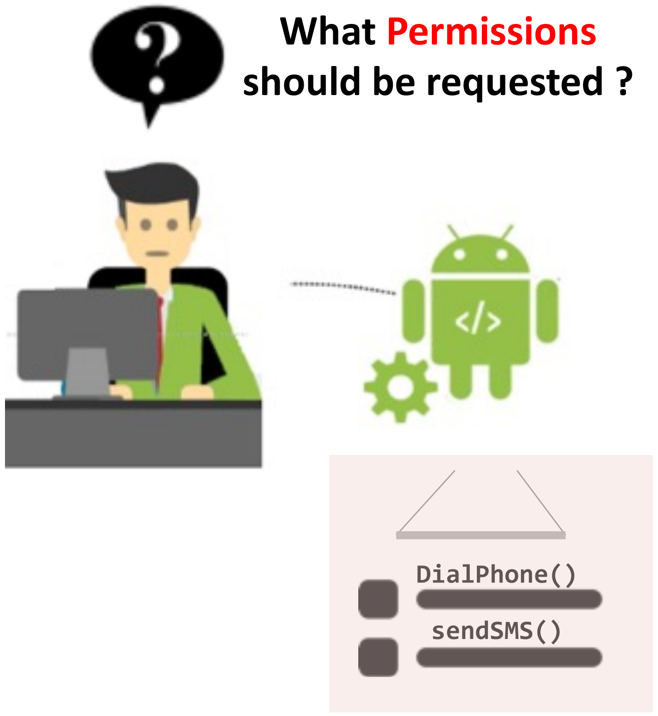
Framework Security
Access Control Analysis

- Lack of an understanding of Android Access Control
- Incomplete / Missing security documentation and specification



Framework Security
Access Control Analysis

- Lack of an understanding of Android Access Control
- Incomplete / Missing security documentation and specification



permission.CALL_PHONE

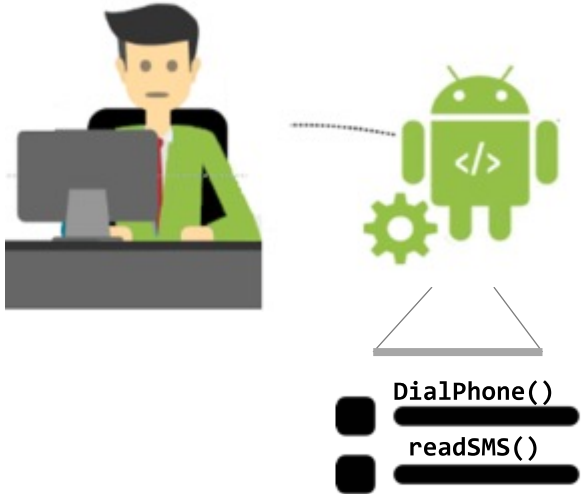
permission.broadcast_SMS

permission.WRITE_SMS

permission.SEND_SMS

Framework Security
Access Control Analysis

- An imprecise / incorrect security specification could lead to the following:
  - Wrong specification to developers
  - Over-privileged apps



Too Many Permissions

permission.CALL_PHONE

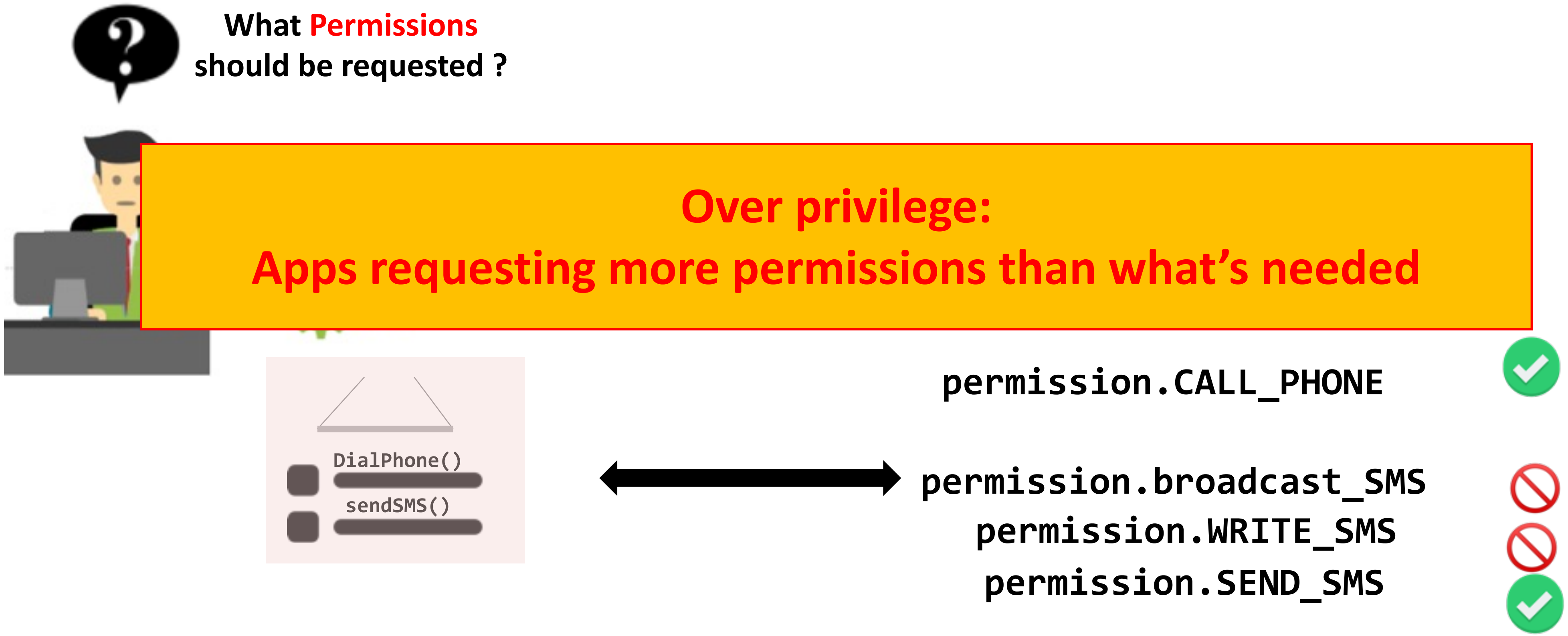
permission.broadcast_SMS

permission.WRITE_SMS

permission.SEND_SMS

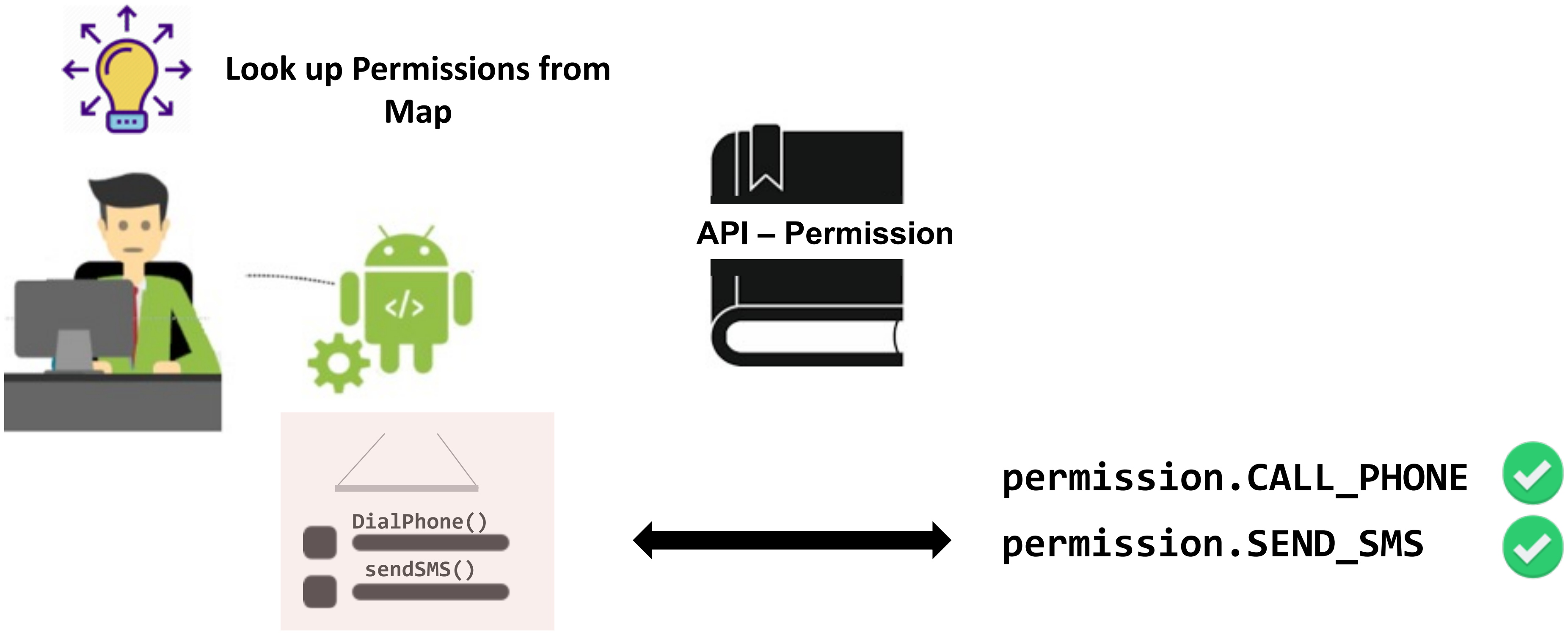
Framework Security
Access Control Analysis

- An imprecise / incorrect security specification could lead to the following:
  - Wrong specification to developers
  - Over-privileged apps



Framework Security
Access Control Analysis

- Solution: ***API to Permission Maps***



Framework Security
Access Control Analysis

- Research Efforts have been proposed to construct the maps

- Dynamic Approaches
  - Use feedback directed API fuzzing
  - Dynamically log permission checks for an API execution

- Static Approaches
  - Construct control flow graphs of APIs
  - Report reachable permission checks from an API

Dynamic Analysis

- Dynamic analysis uses techniques that evaluate a program in real time
- Could be carried out in a virtual environment or on an actual device
- It executes (or emulates) and monitors programs to look for specific behaviors characterizing a vulnerability or a property

- Under the context of Android, dynamic analysis has been used for various tasks
  - Assessing the security of Android apps (e.g., malware detection)
  - Analyzing framework access control

Static Analysis

- Static analysis uses techniques that parse program code or bytecode
- Analyzes the code to check some program properties

- Under the context of Android, static analysis has been used for various tasks
  - Assessing the security of Android apps (e.g., vulnerability identification, detecting app clones)
  - Analyzing framework access control (particularly, permissions).

Dynamic versus Static Analysis

Static Analysis

- More efficient
- Low computation cost (usually)
- Can provide a complete picture of all possible program paths
- May report unfeasible paths
- Cannot handle obfuscated code
- Cannot handle dynamically loaded code

Dynamic Analysis

- More informative, as it can provide specific details about a behavior during runtime.
- Can handle highly obfuscated code.
- Coverage problems – may miss to execute interesting behavior

Framework Security

Constructing Permission Maps through Dynamic Analysis

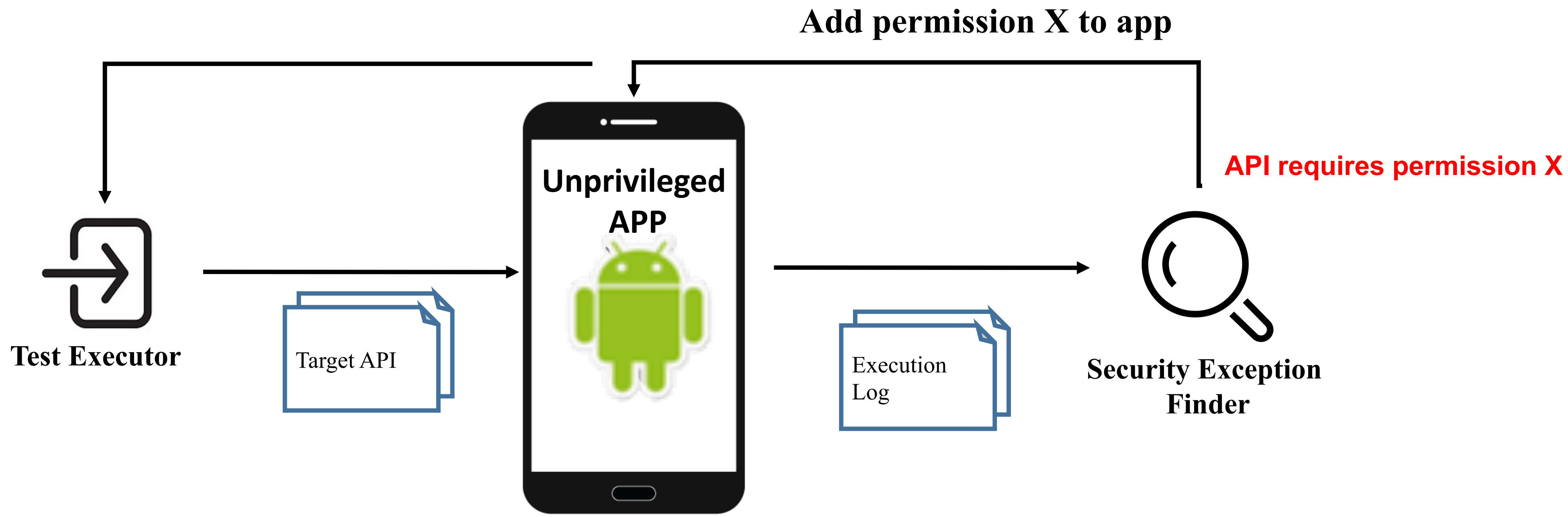
- *Recap:* Access control enforcement in Android

```
WifiService

Void setWifiApEnabled(…)
{

   if(caller.hasPermission(“android.permission.CHANGE_WIFI_STATE”) &&
caller.hasPermission(“android.permission.CONNECTIVITY_INTERNAL”))
   {
         …
         //perform actual enabling(…);
   }
   else
         // throw Security Exception
}
```

API setWifiApEnabled requires
android.permission.CHANGE_WIFI_STATE
AND android.permission.CONNECTIVITY_INTERNAL

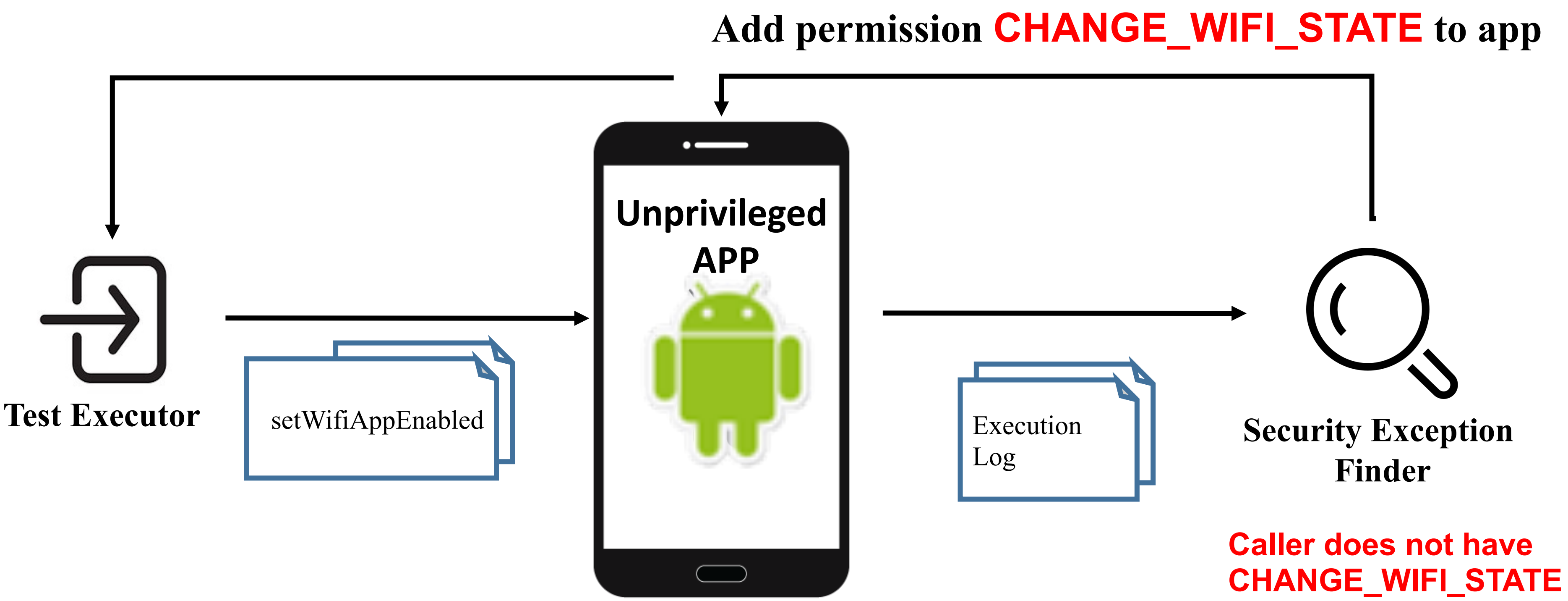
Framework Security
Constructing Permission Maps through Dynamic Analysis

- Approach: Invoke the APIs from unprivileged apps and detect the checks that protect them



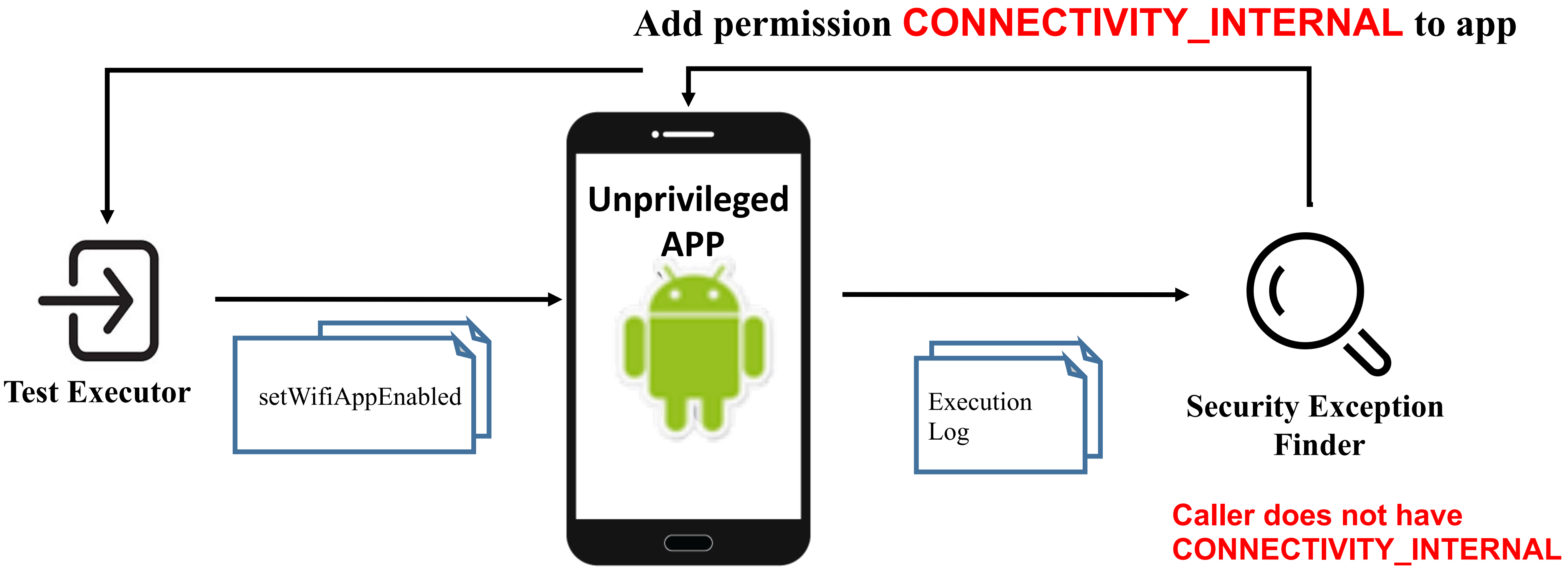
Framework Security
Constructing Permission Maps through Dynamic Analysis

- Approach: Invoke the APIs from unprivileged apps and detect the checks that protect them



Framework Security
Constructing Permission Maps through Dynamic Analysis

- Approach: Invoke the APIs from unprivileged apps and detect the checks that protect them



Framework Security

Constructing Permission Maps through Dynamic Analysis

- Approach: Invoke the APIs from unprivileged apps and detect the checks that protect them

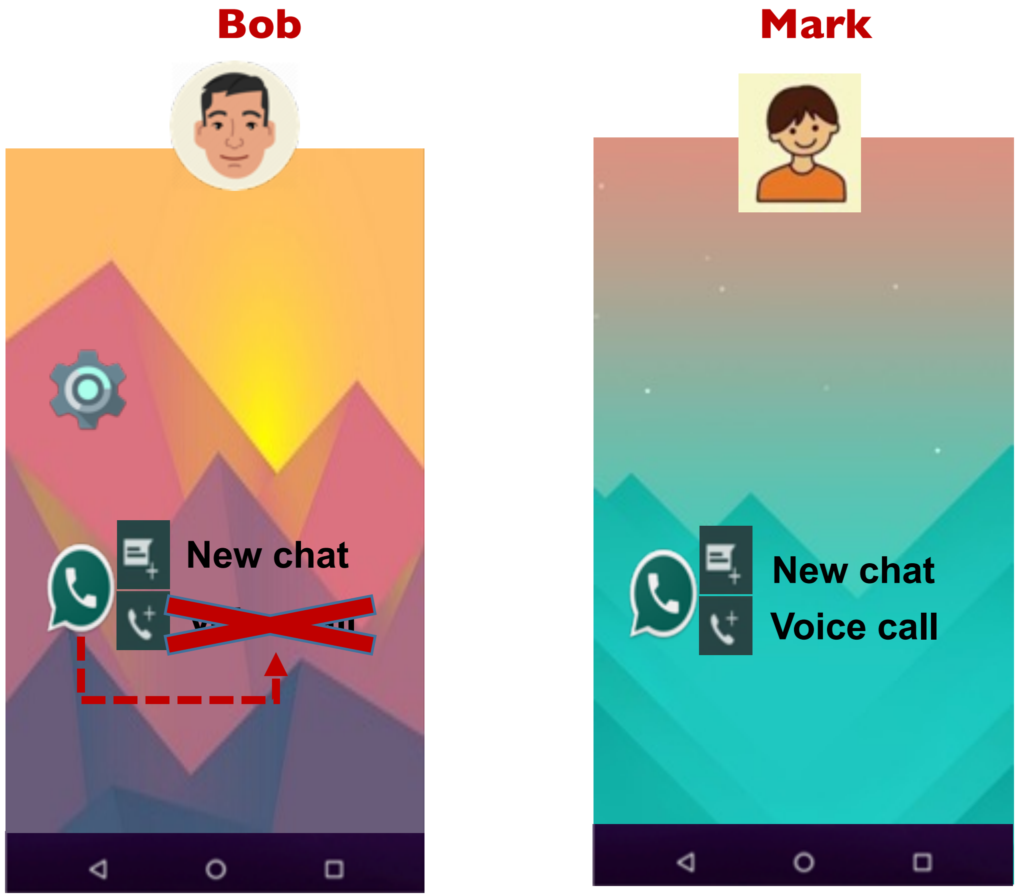
Framework Security

Constructing Permission Maps through Dynamic Analysis

- Certain permission enforcement might not be encountered unless specific inputs are supplied.
- Solution: Fuzzing

Framework Security

Constructing Permission Maps through Dynamic Analysis

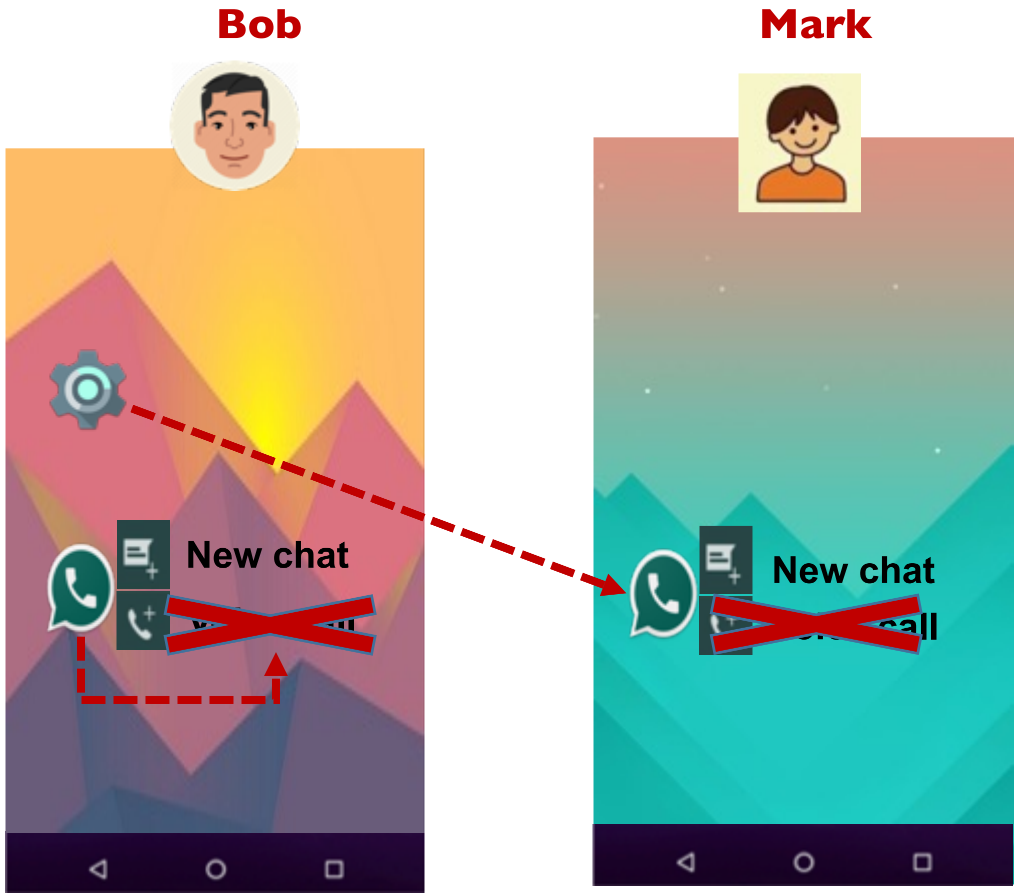


- Scenario 1:

  Bob disables his Whatsapp's voice calling

Framework Security

Constructing Permission Maps through Dynamic Analysis



- Scenario I:

  Bob disables his Whatsapp's voice calling

  Should not require any permission to disable its own component

- Scenario II:

  Bob disables voice calls for Mark

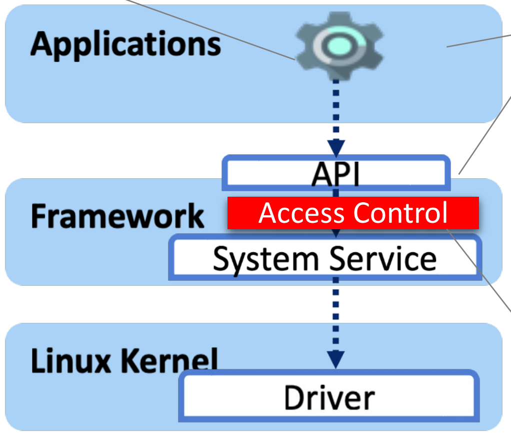
  Should require a permission to disable
  - A component in other apps
  - A component in other users

- Intuitively, the two scenarios demand different permissions

Framework Security

Constructing Permission Maps through Dynamic Analysis

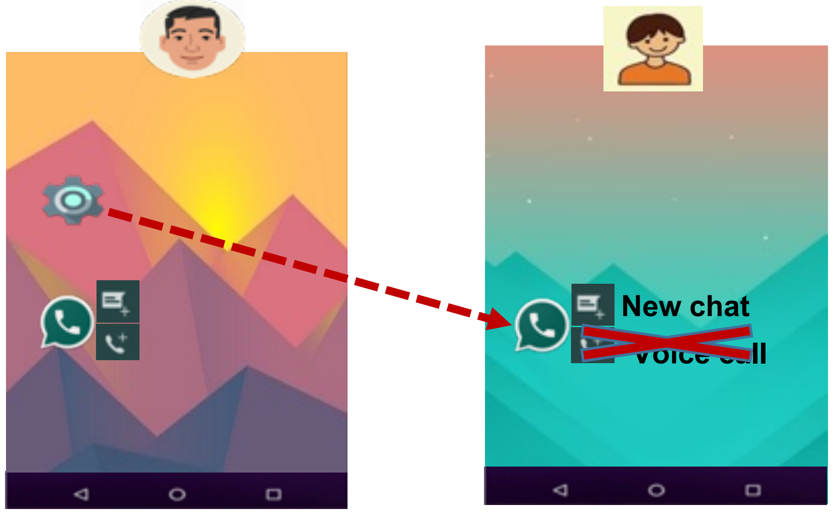
```
disableComponent (int userID, int appId));
```




```
disableComponent(int userID, int appID) {
  if (callerUserId != userID)         // Bob's User id is not equal to Whatsapp's owner (Mark's User id)
    if (!hasPermission(INTERACT_ACROSS_USERS)) exception;

  if (callerUid != appID)             // Setting app's Uid is not equal to whatsapp's Uid
    if(!hasPermission(CHANGE_ENABLED_SETTING)) exception;

  disableState(...);
```

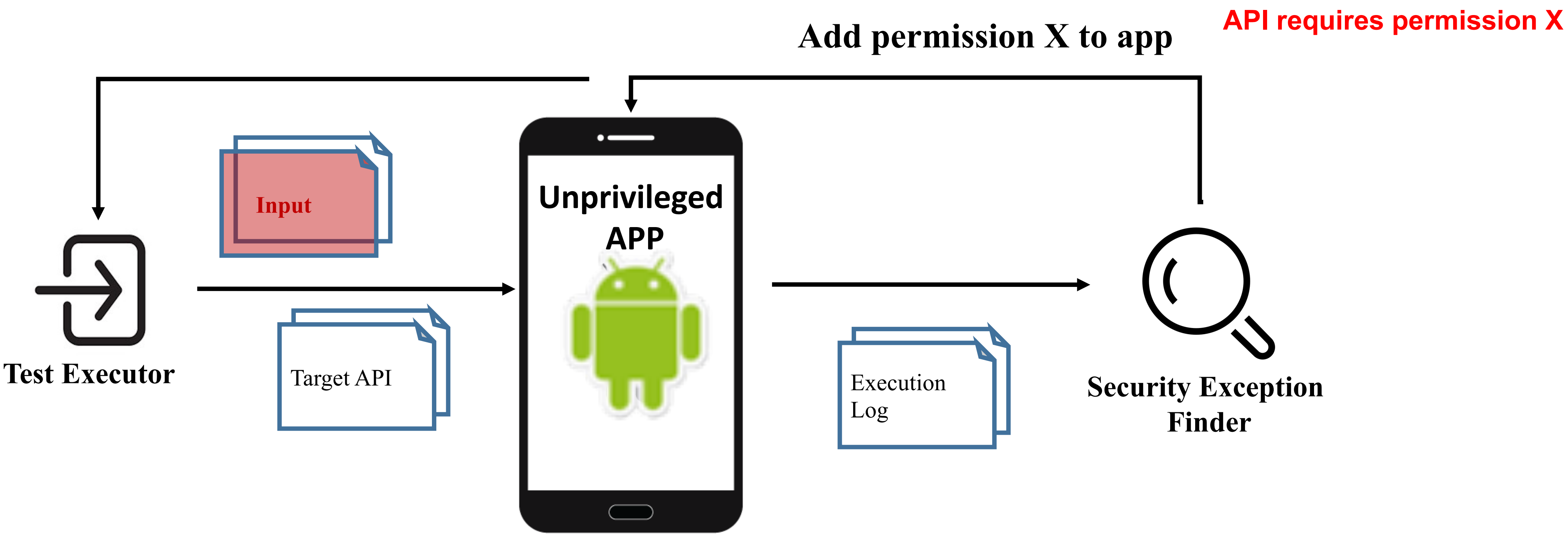
Bob (User 0)

Mark (User 1)



Framework Security
Constructing Permission Maps through Dynamic Analysis

- Generate different inputs



Framework Security

Constructing Permission Maps through Dynamic Analysis

```
disableComponent(int userID, int appID) {
  if (callerUserId != userID())
    if (!hasPermission(INTERACT_ACROSS_USERS)) exception;

  if (callerUid != appID)
    if(!hasPermission(CHANGE_ENABLED_SETTING)) exception;

  disableState(...);
```

Input : **arg0 =** callerUserId
V
Perm = INTERACT_ACROSS_USERS

Λ

Input : **arg1 =** callerUid
V
Perm =CHANGE_ENABLED_SETTING

Framework Security

Constructing Permission Maps through Static Analysis

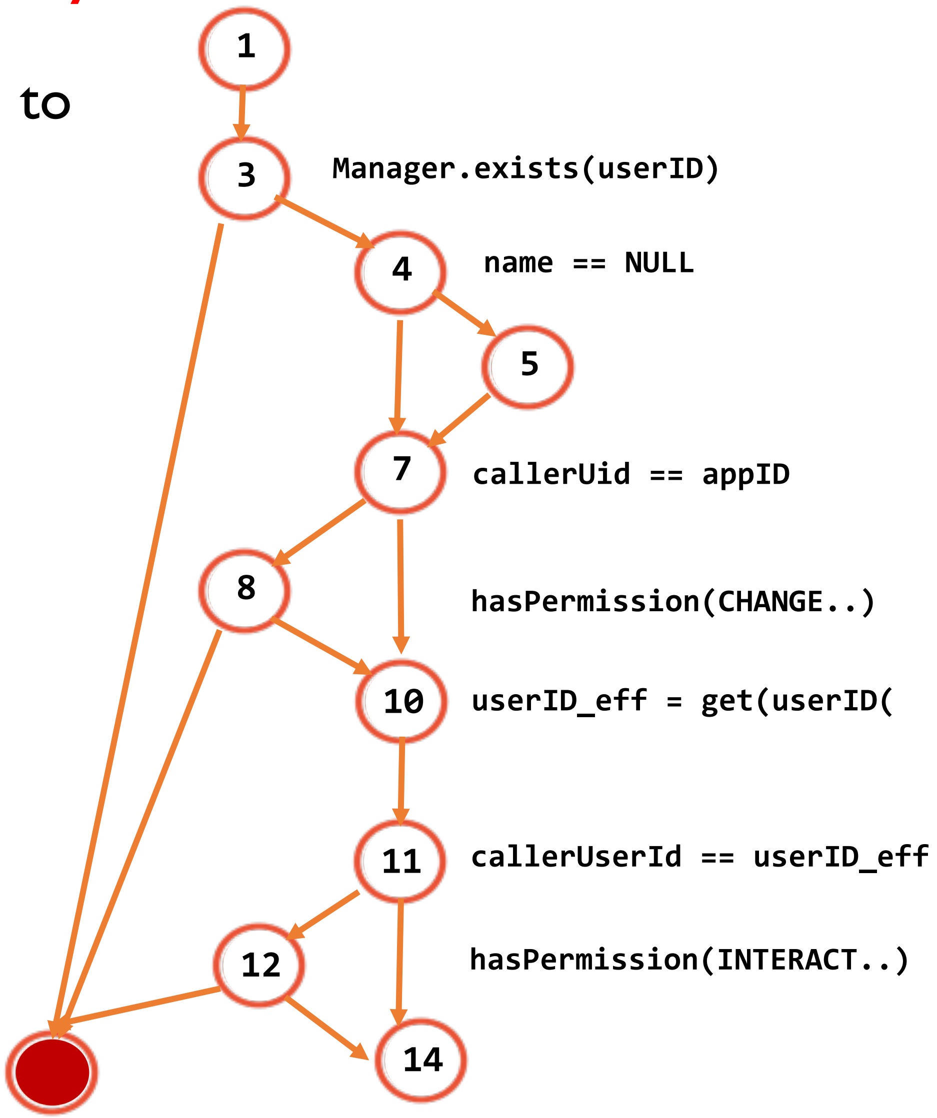
- Static analysis approaches proceed as follows:
  - Identify entry points (i.e., APIs) defined in the framework.
  - Build a control flow graph (cfg) of each API
  - Perform a reachability analysis on the cfg
  - Identify access control enforcement methods
    - Path insensitive:
    - Path sensitive

Framework Security

Constructing Permission Maps through Static Analysis

- Given a target API, static analysis approaches analyze its CFG to identify access control checks

```
1: disableComponent(int userID, int appID, String name) {
2:
3:   if (!Manager.exists(userID)) return;
4:   if (name == null)
5:     isApp = true;
6:
7:   if(callerUid!= appID)
8:     if(!hasPermission (CHANGE_ENABLED_SETTING) exception;
9:

10: userID_eff =  get(userID);
11: if (callerUserId!= userID_eff)
12:    if(!hasPermission(INTERACT_ACROSS_USERS)) exception;
13:
14: disableState(...);
```

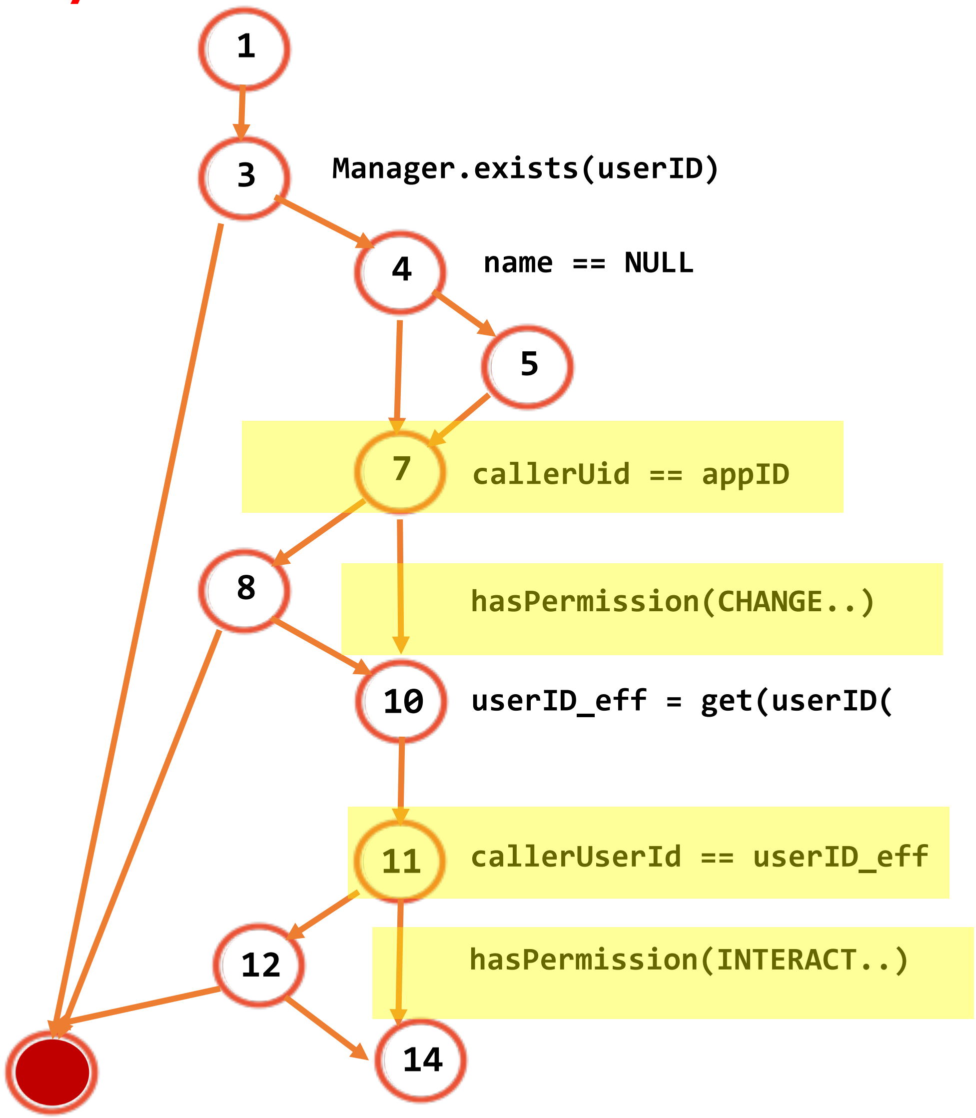


- CFG is quite complex

Framework Security

Constructing Permission Maps through Static Analysis

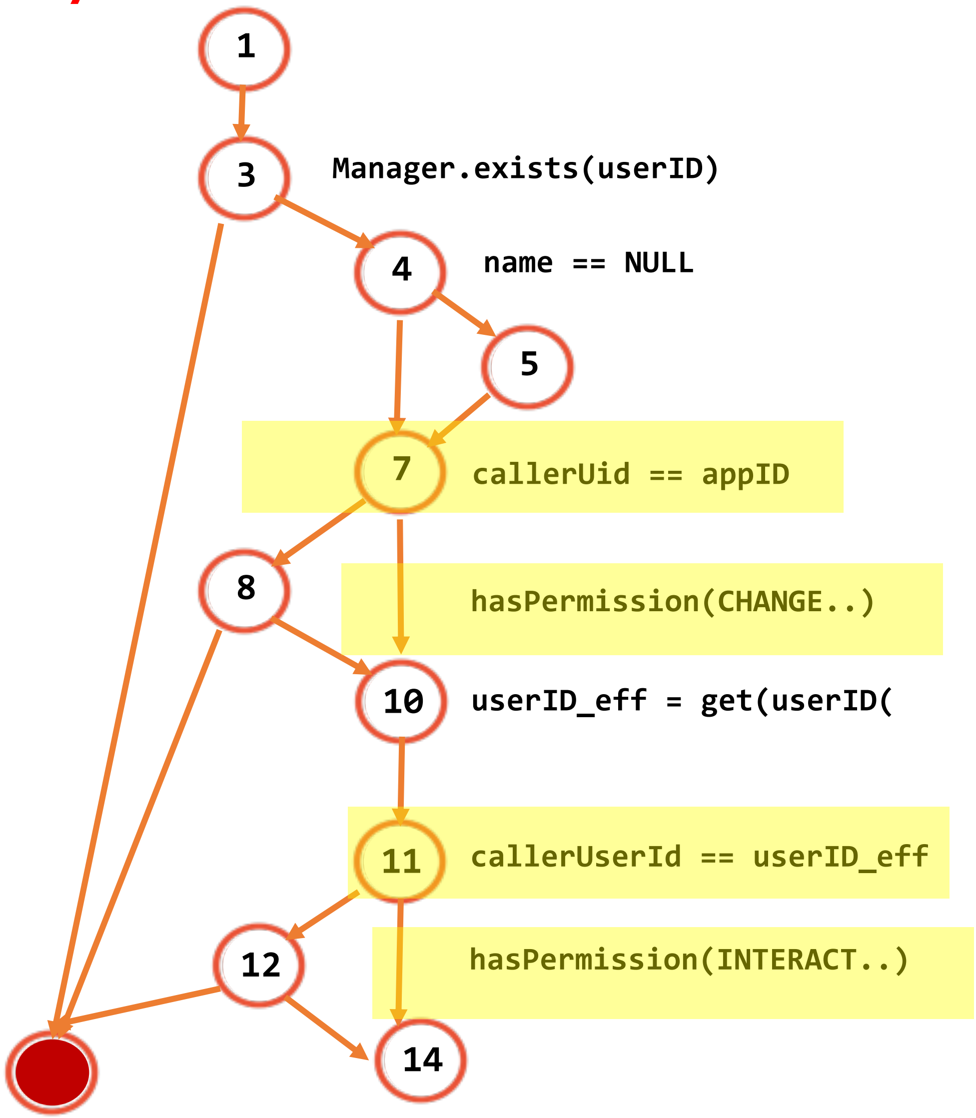
- Not all nodes in the cfg are of interest in the construction of the api - permission maps



Framework Security

Constructing Permission Maps through Static Analysis

- Permission Map can be constructed either in a path-insensitive or path-sensitive fashion
- Path-insensitive:
    - Report a union of all identified permissions
- Path-sensitive:
    - Permission Map is constructed by extracting path conditions of all paths from the entry point
    - Each path denotes a way to acquire the needed access.
    - Permission map is a first-order logic formula formed by the disjunction of these path conditions

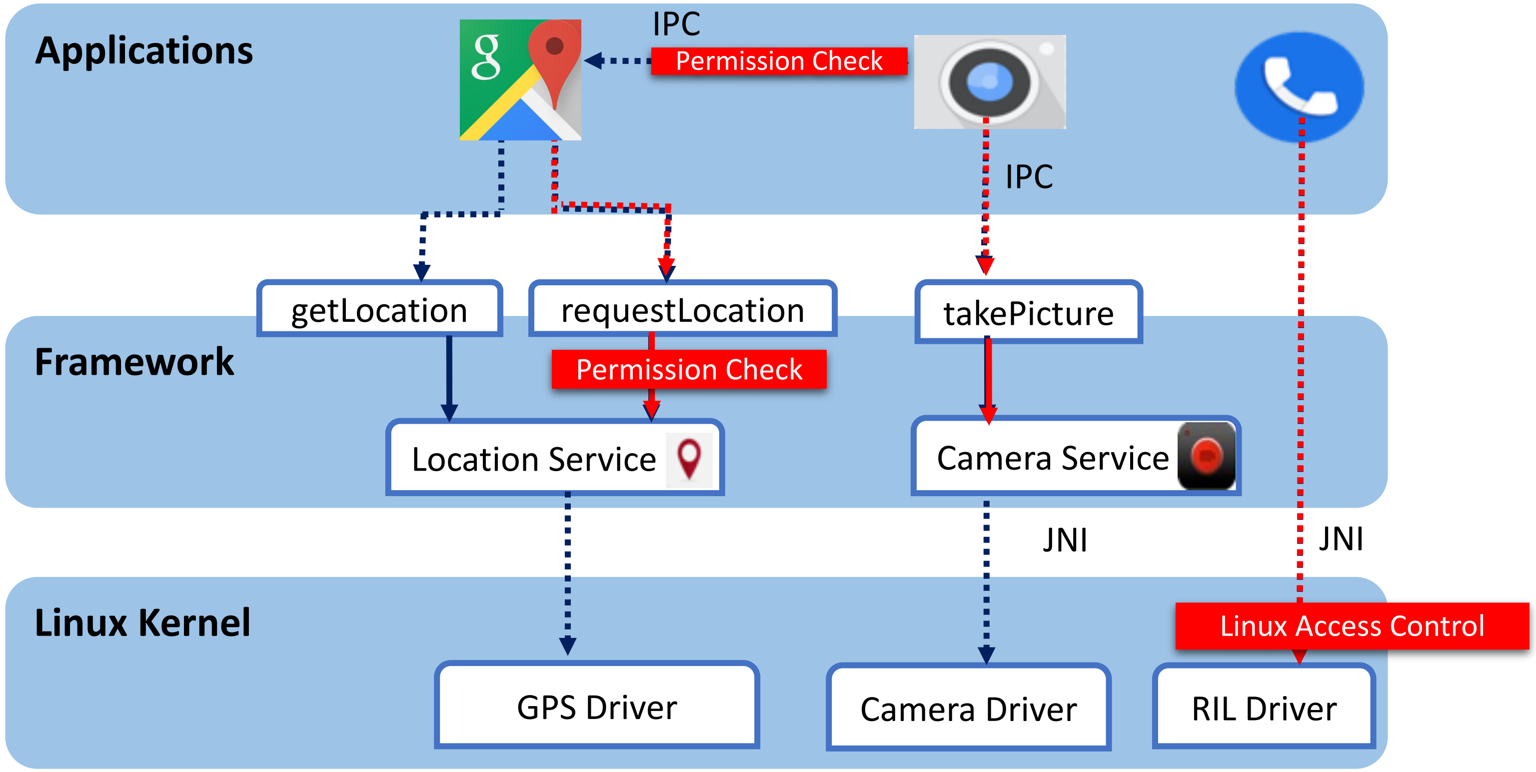


Android Access Control Analysis

Vulnerability Detection

Framework Security
Access control enforcement

- Recap: Protecting different resources in various layers of the OS

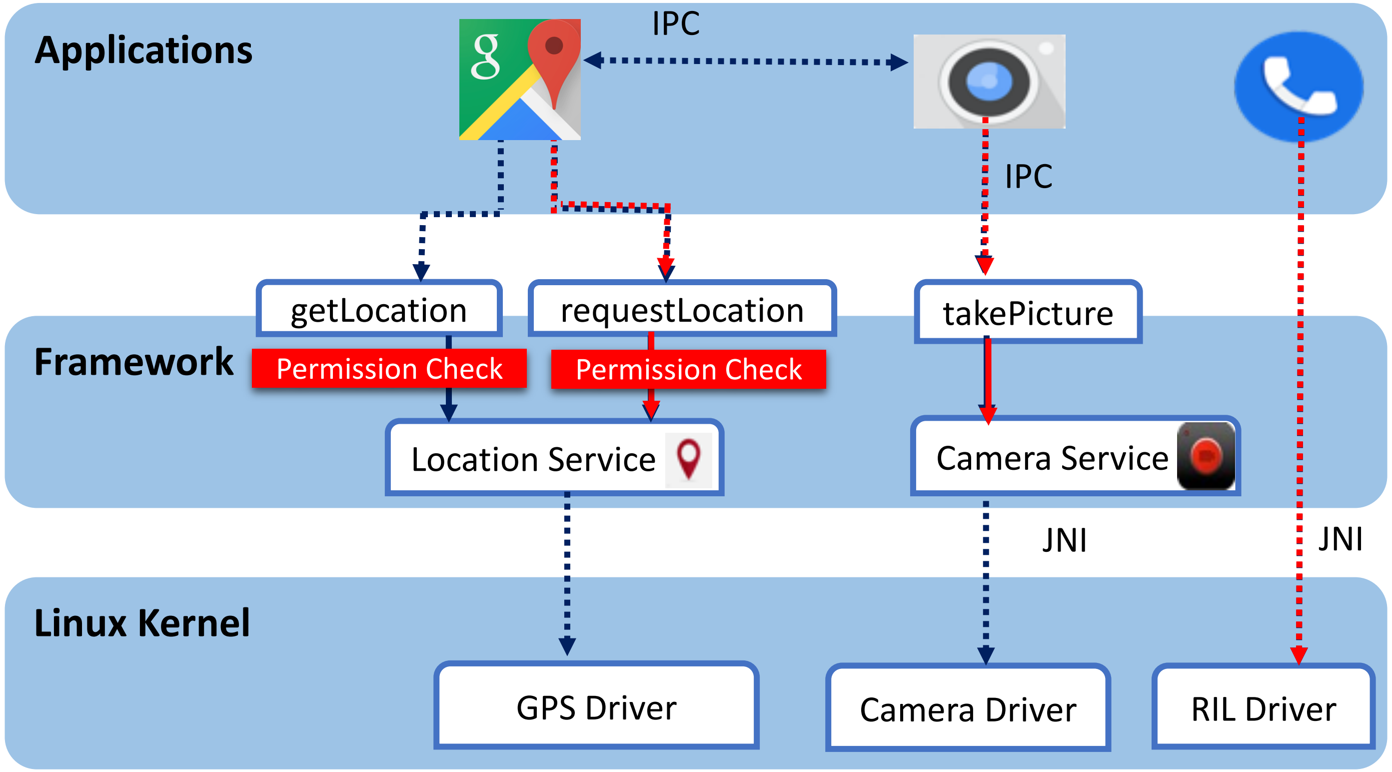


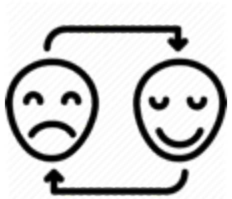
Framework Security

Access control enforcement: ***EFFECTIVE??***

Lack of an Oracle: It's difficult to determine if a resource is correctly protected

Approximate Solution: Compare Access Control enforcement across multiple instances of the same resource

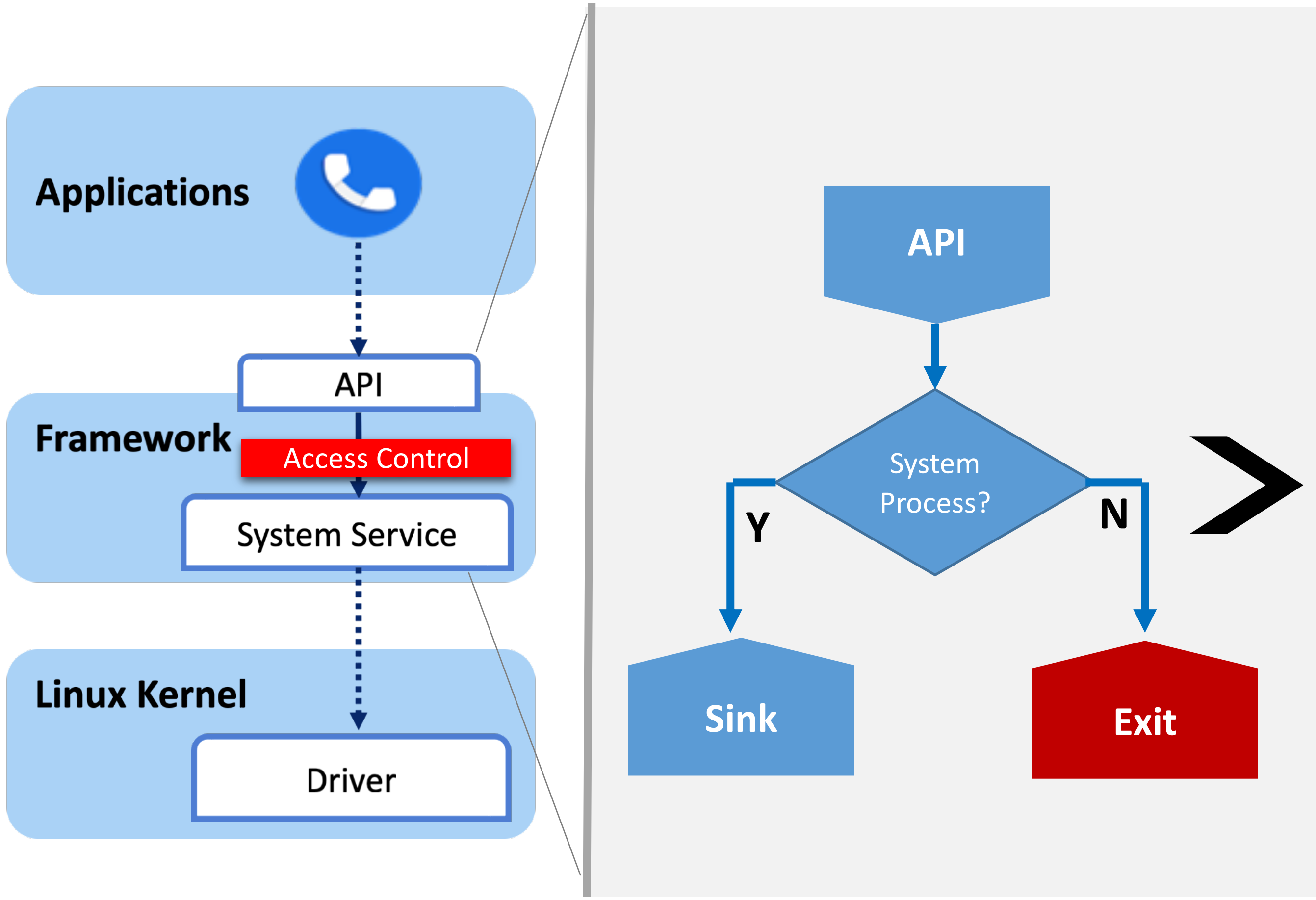




Inconsistencies are Potential Vulnerabilities

Comparing API Access Control Enforcements

- Android Access Control features Diversity / Complexity
- No Gold Standard to implement Access Control

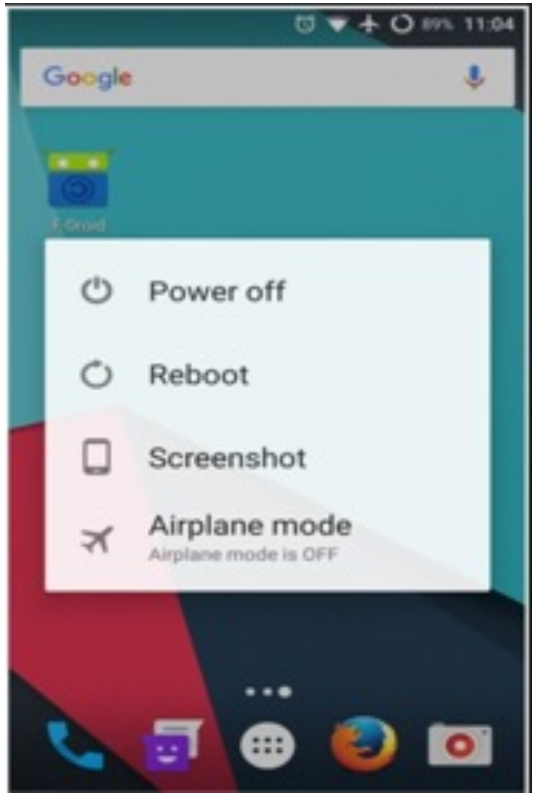


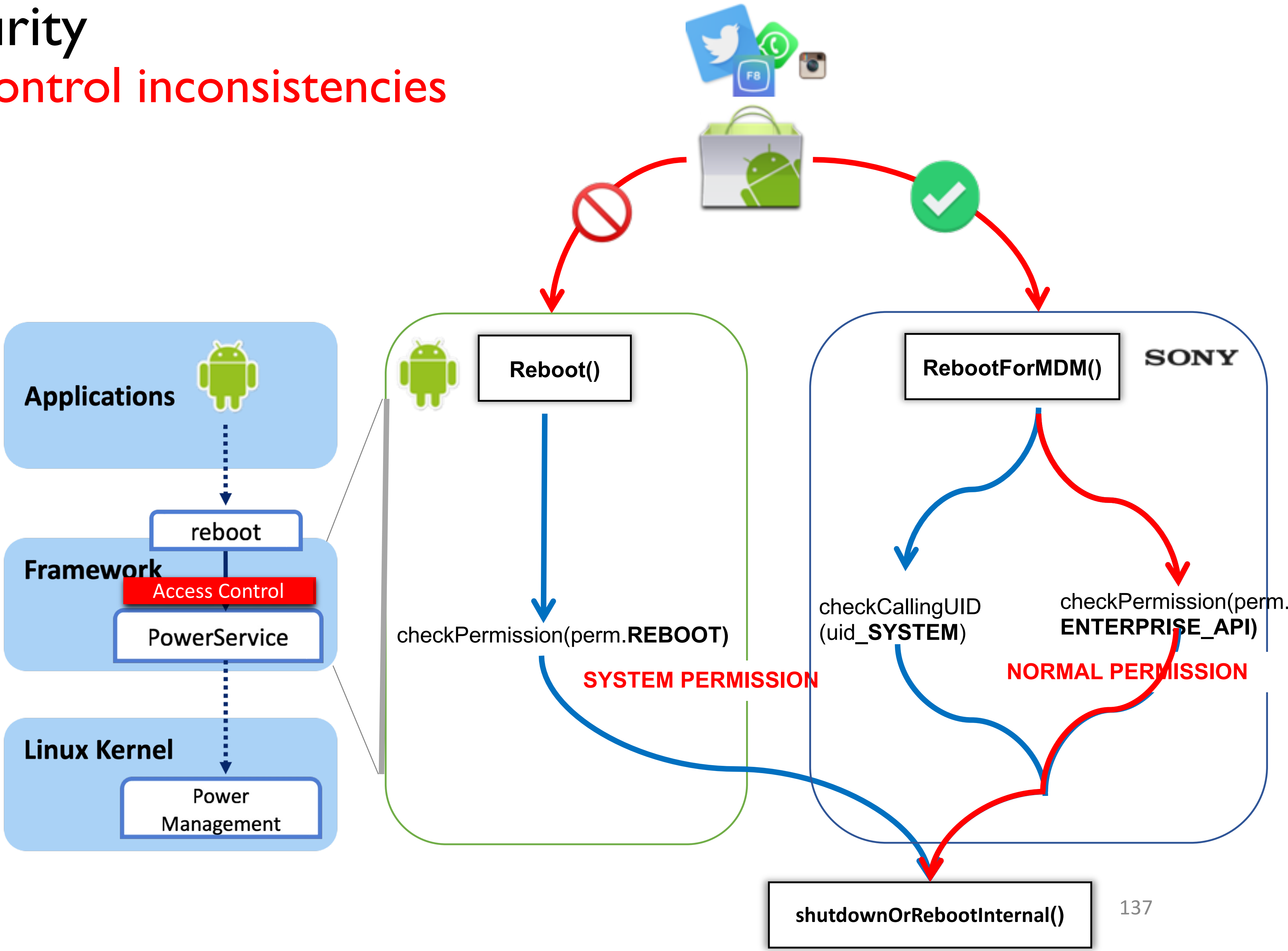
- `packageInfo(CallingAppUid).`**`Signature`**` == [Platform_Signature]`
- `packageInfo(CallingAppUid).`**`Flags`**` & [FLAG_SYSTEM] != 0`
- `CallingAppUid == `**`SYSTEM_UID`**
- `CallingAppPid == `**`Process.myPid()`**
- …

Framework Security

Detecting access control inconsistencies

Exploitable case





Framework Security
Detecting access control inconsistencies

- Approximate solutions:
  - Perform convergence analysis for two APIs
  - Extract access control enforcement for the APIs as a union
  - Inconsistency is detected if the paths reveal different access control checks.

- More precise solutions:
  - Perform convergence analysis for two APIs
  - Extract access control enforcement along each individual execution path of an API
  - Normalize access control enforcement to account for diversity

Framework Security

Detecting access control inconsistencies

- Normalizing access control based on program structures:

Case: Multiple permissions are enforced

```
public boolean requestRouteToHostAddress(...){
  enforceCallingPermission("permission.CHANGE_NETWORK_STATE");
  enforceCallingPermission("permission.CONNECTIVTY_INTERNAL");
  addRouteToAddress(...);
```

NORMAL

SYSTEM

∨

Normalized Value = Max (NORMAL, SYSTEM)

=> SYSTEM

Framework Security

Detecting access control inconsistencies

- Normalizing access control based on program structures:

Case: Either permission is enforced

```
public boolean getSubscriberId(...){
  try{
    enforceCallingPermission("READ_PRIVILEGED_PHONE_STATE");
  }catch(SecurityException){
    enforceCallingPermission("READ_PHONE_STATE");
  }
  return mPhone.getSubscriberId();
```

SYSTEM

DANGEROUS

∨

Normalized Value = Min (DANGEROUS, SYSTEM)

=> DANGEROUS

App Security

Component Hijacking Vulnerabilities

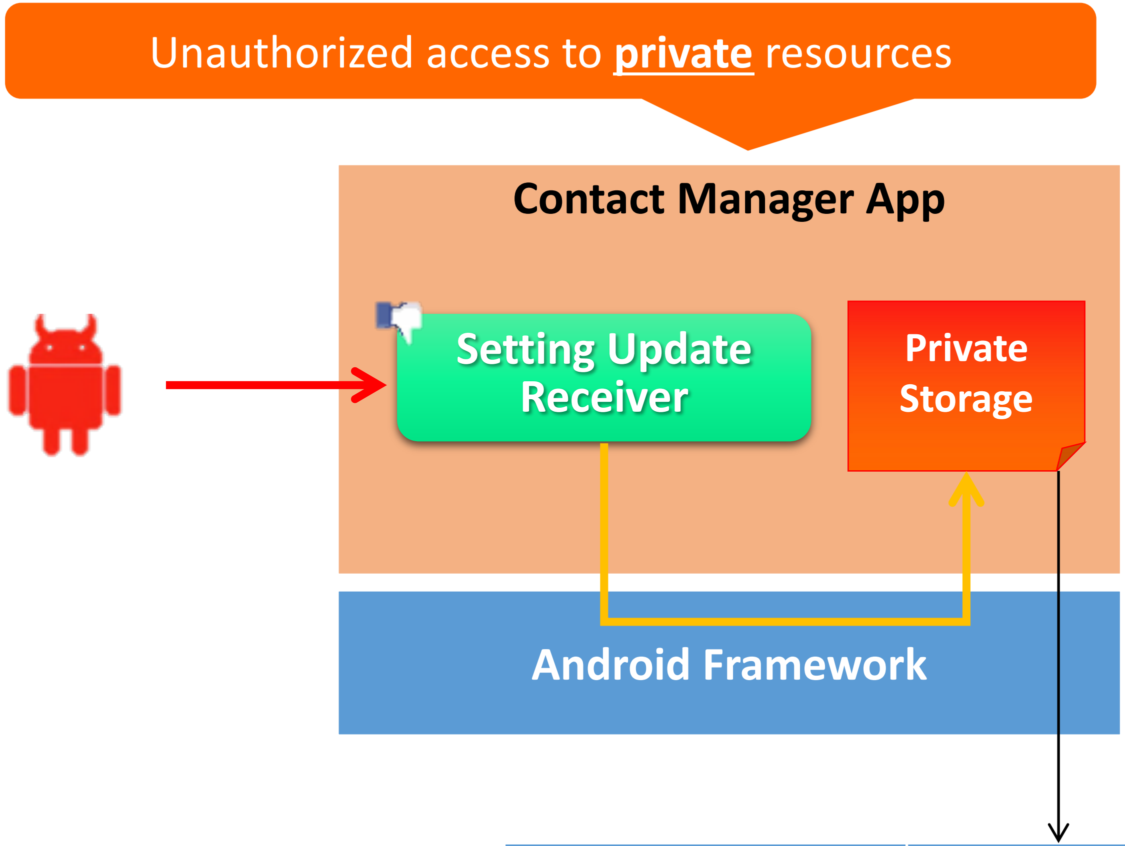
Security concerns in mobile apps
Component Hijacking (or permission re-delegation attacks)

- Class of attacks that seek to gain unauthorized access to protected sensitive resources through under-protected app components

- Unauthorized access could reflect:
  - Invocation of a sensitive API (i.e., an API that enforces access control).
  - Read sensitive data (attack a.k.a. Content Leaks)
  - Write to sensitive data (attack a.k.a. Content Pollution)
  - Combination of the above.

Security concerns in mobile apps

Example of Component Hijacking



Setting Update Receiver

- Accepts external updates
- App Internal DB is permission protected
- Write to critical area

Key	Value
VoIP_Prefix	“1234”
Is_App_Lisenced	false

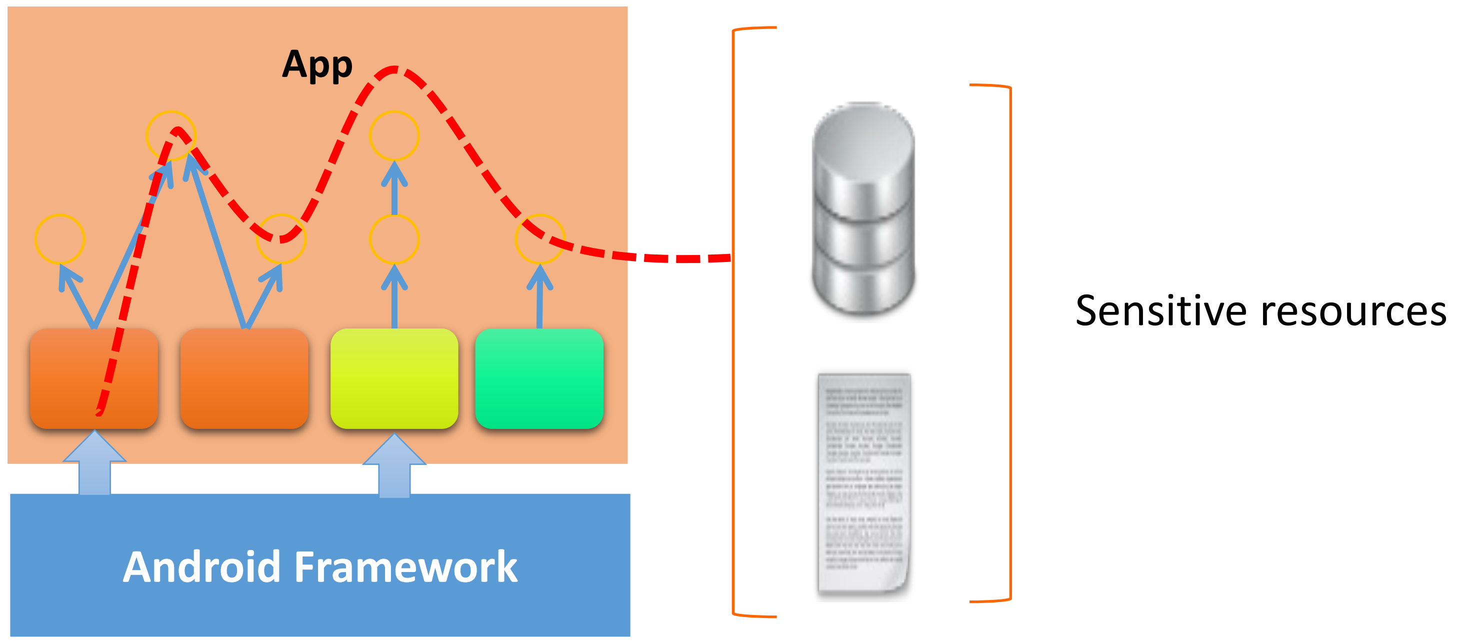
Security concerns in mobile apps
Vetting apps for Component Hijacking

- Identify sensitive resources reachable from an app component
- Compare the protection specification of the app component against that of the sensitive resource
  - If the component's protection is weaker, a hijack-enabling flow is detected

Security concerns in mobile apps

Vetting apps for Component Hijacking

- Challenges:
  - Component hijacking is also possible on a chain of components
  - Hijack-enabling flows could span across component boundaries



Security concerns in mobile apps
Vetting apps for Component Hijacking

- Challenge:
  - Component hijacking is also possible on a chain of components
  - Hijack-enabling flows could span across component boundaries

- Addressing this challenge requires:
  - Tracking flows across components
  - Assessing the collective effect of individual flows and identify the target flow of interest
  - Modeling the asynchronous nature of inter-app component interaction

App Privacy

Information Leakage

Privacy concerns in mobile apps
Information Leakage

- Apps may have access to sensitive information:
  - Sensor and device specific: IMEI, GPS coordinates, etc.
  - User specific: SMS messages, banking information, etc.

- Apps may leak information:
  - Send sensitive information to an external server
  - Via various channels and mechanisms -- e.g., SMS, email, directly or using other apps.

Privacy concerns in mobile apps
Information Leakage

- Why would apps leak user information?
- Apps installed from third-party markets maybe potentially harmful
  - Ads
  - Identity theft
  - Tracking the user
  - Etc.

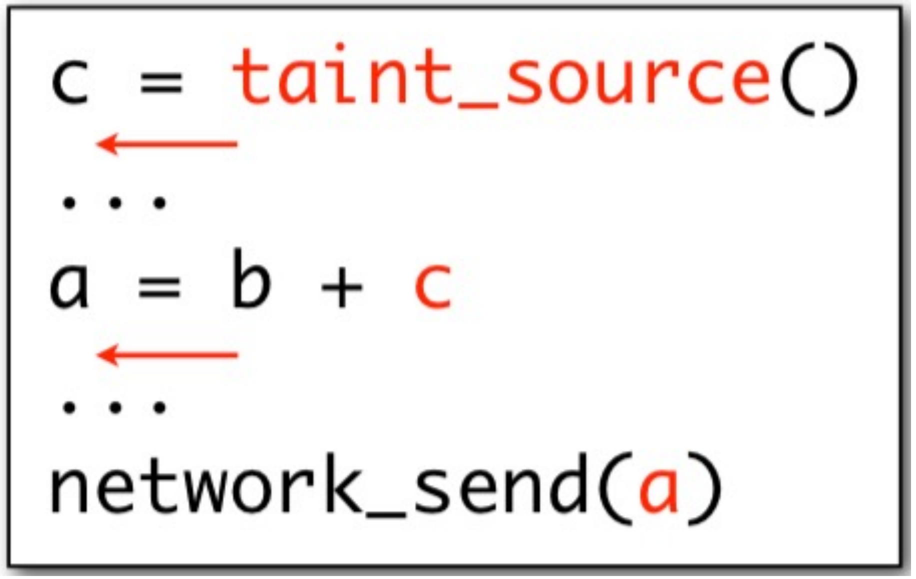
Privacy concerns in mobile apps
Challenges for detecting information leakage

- Cannot deploy traditional information leakage detection solutions
  - Limited computation power
  - Limited battery
- Cannot detect certain sensitive information
  - Indistinguishable from non-sensitive information
- Requires monitoring inter-app communication
  - Facebook may share information with Twitter

Privacy concerns in mobile apps

Solutions for detecting information leakage

- Dynamic Taint Analysis is a technique that allows tracking information flow between sources and sinks
- Any program value that depends on a tainted source is considered tainted.
- At high level, it involves three stages:
  - Taint source
  - Taint propagation
  - Taint sink



Privacy concerns in mobile apps

Solutions for detecting information leakage

- Examples of sources
  - APIs allowing to read IMEI
  - Sensitive Database query methods
- Examples of sinks:
  - APIs allowing to send messages
  - Network APIs

Privacy concerns in mobile apps

Solutions for detecting information leakage

- TaintDroid[1] is a classic solution for dynamic taint analysis in Android.

- It is an extension to the Android platform that allows tracking the flow of privacy sensitive data through third-party apps

Privacy concerns in mobile apps

Solutions for detecting information leakage

- TaintDroid works as follows:
  - It automatically labels data from target sources.
  - It transitively applies labels as the data propagates through the various program variables, files, and inter-process messages.
  - It checks if the labeled data is leaving the system via target sinks.
- TaintDroid logs the application responsible for transmitting the sensitive (tainted) data over the internet (or other external channels).

Recap

- Overview of Android OS
- Security Mechanisms
- App Security
- Advanced Topics: Permission Maps and Access Control Anomalies