Introduction to Mobile Application Security and Static Analysis

Alper Basaran

Alper Basaran

Penetration Tester / Cybersecurity Consultant

CISSP | CISA | CPTE | CPTC | eCIR | GPEN | OSWP | GICSP

About This Course

- Overview of mobile operating systems
- OWASP Mobile Top 10
- Threat modeling
- Static analysis

The Bad News

- Mobile apps are platform dependent

Mobile Apps Are Everywhere

- Over 8.5 Billion mobile phone users worldwide
- Everyone spends almost 4 hours daily on mobile phones
- Over 88% of this time spent on mobile apps

The Mobile Threat Landscape

- Mobile devices create a new threat landscape
- Common mobile security challenges:
  - Insecure data storage
  - Weak authentication
  - Excessive app permissions
  - Insecure API
  - Mobile malware
  - Mobile OS vulnerabilities

Mobile Application Security Stakeholders

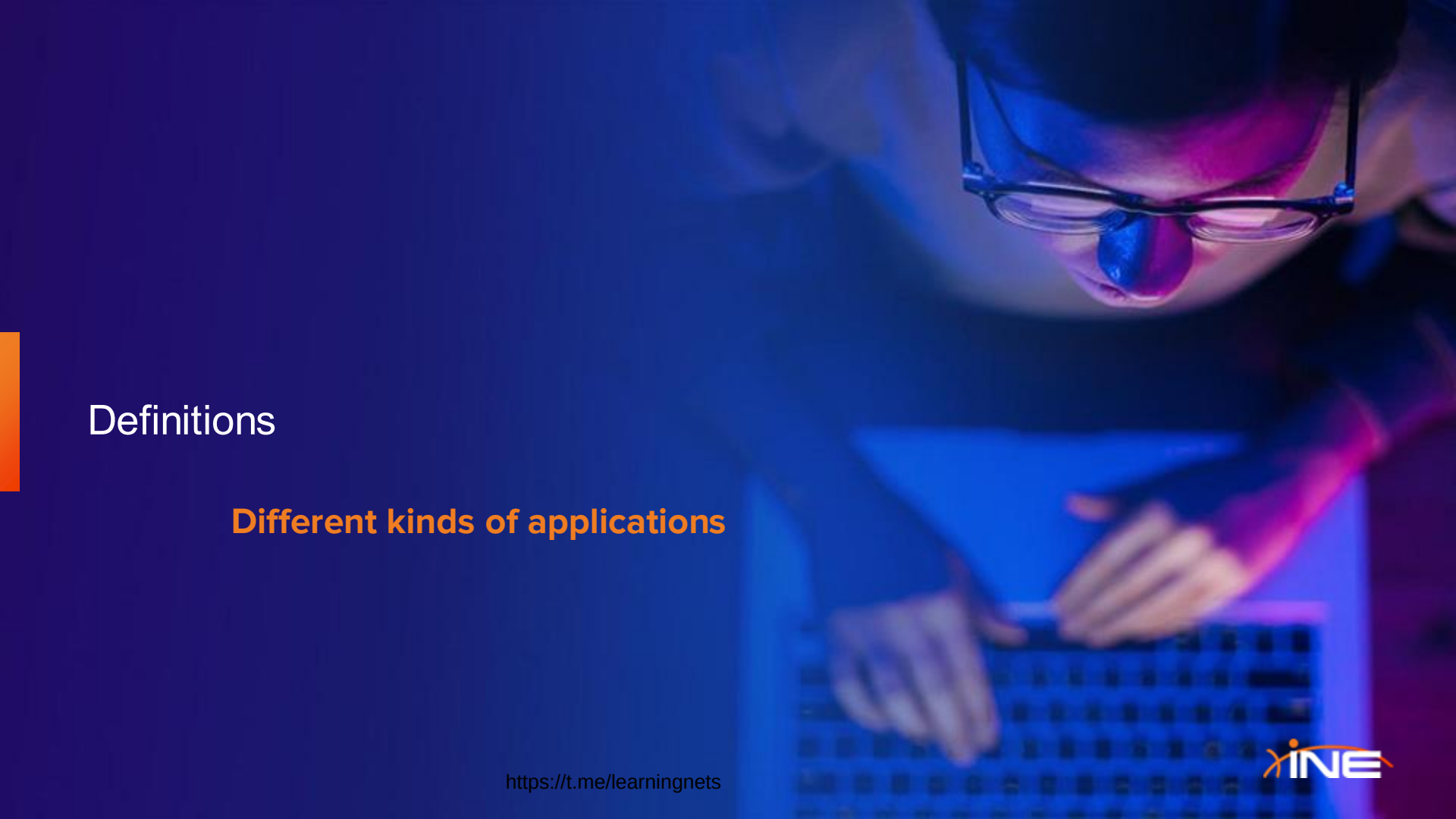
- Developers
- Organizations
- Users
- Security professionals

Future Trends and Challenges

- IoT integration
- AI impact
- Increased regulations
- BYOD and Zero Trust Security models

Definitions

Different kinds of applications



What are “apps”?

- Any application that runs on a mobile OS
  - We will focus on Android and iOS

Apps are Built Differently

- Native App:
  - Developed using the platform-specific SDK
    - Android: Java or Kotlin
    - iOS: Objective-C or Swift
  - Usually runs faster
  - Can access almost every component of the device and OS

Apps are Built Differently

- Mobile web apps
  - Feels like an app but runs on the browser
  - Are limited by the browser's access level (or sandbox)
  - Allows developers to have a single code base

Apps are Built Differently

- Hybrid apps
  - Runs like a native app but uses the web browser
  - Can access more than web apps
  - A single code base can generate apps for different platforms

Apps are Built Differently

	Native Apps	Hybrid Apps	Web Apps
Performance	+ + + + +	+ + +	+ +
Device Access	Full	Limited	Very limited
Security	High	Moderate	Low
Development Time	Longest	Mid	Fastest
Single Code Base?	No	Yes	Yes

Mobile App Security Testing

What do we look for?

Why Do We Test?

+ The needs and expectations for tests may be different
+ Bug bounty hunters, penetration testers, and consultants may have different approaches

How Do We Test?

+ Black-box testing
+ White-box testing
+ Gray-box testing

How Do We Test?

	Knowledge Level	Best For	Weaknesses
Black-Box	No knowledge	Simulating real-world attacks	Limited visibility into code flaws
White-Box	Full access	Deep security analysis	Time-consuming & requires source code
Gray-Box	Partial knowledge	Realistic attack simulations	Requires some internal access

How Do We Test?

+ Penetration testing
+ Vulnerability analysis
+ Threat modeling
+ Source code analysis

How Do We Test?

- We will have two general approaches:
  - Static analysis
  - Dynamic analysis

How Do We Test?

- Static Analysis:
  - Analyze the application without running it
  - Will give you a general idea about the app
  - Can NOT help you identify runtime vulnerabilities
  - Can be very stressful if you are not a developer

How Do We Test?

- Dynamic Analysis:
  - Analyze the application while running it
  - Can help you find vulnerabilities such as:
    - Credentials sent without encryption
    - Credentials stored on the device

How Do We Test?

	Static Analysis (SAST)	Dynamic Analysis (DAST)
Execution Required?	No	Yes
Finds Hardcoded Secrets?	Yes	No
Finds Runtime Issues?	No	Yes
Finds API Weaknesses?	Limited	Yes

The Penetration Testing Process

1. Pre-engagement
2. Recon
3. App mapping
4. Exploitation
5. Reporting

The Penetration Testing Process

- Pre-engagement
  - Your approach will be different based on:
    - What the customer expects
    - What regulations expect
  - Some apps may be difficult to test
    - You may have to ask for a test version

The Penetration Testing Process

- Pre-engagement
  - Know the data
    - Define sensitive data
      - Authentication data (user creds, user info, etc.)
      - PII
      - Data protected by laws and regulations

The Penetration Testing Process

- Pre-engagement
  - Locate the data
    - Where is it stored?
    - How is it stored?
    - Where is it sent?
    - How is it sent?

The Penetration Testing Process

- Pre-engagement
  - Identify security related items
    - Hashing
    - Encryption
    - Encoding
    - Token storage (API, session, etc.)
    - Random number generators

The Penetration Testing Process

- Recon:
  - Purpose of the app
  - Developer of the app
  - Industry, stakeholders
  - How the app works

The Penetration Testing Process

- App mapping:
  - App architecture:
    - How the app works (e.g. how it manages user sessions)
    - How the app communicates
    - What the app communicates with
  - Threat modeling

The Penetration Testing Process

- Exploitation:
  - Not all vulnerabilities are relevant or exploitable
  - Look at:
    - Damage potential
    - Discoverability
    - Reproducibility
    - Exploitability
    - Data or users impacted

The Penetration Testing Process

- Reporting:
  - You are only as good as your report
  - Reports should have:
    - Executive summary
    - Definition of the scope
    - Methods used
    - Findings
    - Recommendations

Differences Between Web and Mobile Application Security

A look at the threat landscape

Quick Overview of Web Applications

- Accessed via a browser from any platform
  - Desktop
  - Mobile
  - TV, fridge, etc.
- Security mostly focused on servers
  - Corporate / sensitive / critical data kept on servers

Quick Overview of Mobile Applications

- Dedicated app installed on the device
- May include offline functions
  - Camera
  - NFC
  - Bluetooth
  - GPS

Key Differences in Security Challenges

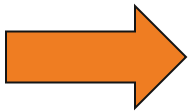
- Mobile apps are an important targets: 54% of all web traffic globally
- Malicious apps find their way to app stores
- Reverse engineering is an real threat

Common Security Vulnerabilities

- Web Applications:
  - SQL Injection
  - Cross-Site Scripting
  - Cross-Site Request Forgery
- Mobile Applications:
  - Insecure Data Storage
  - Reverse Engineering
  - Weak API Security

Common Security Vulnerabilities

- Web Applications:
  - SQL Injection
  - Cross-Site Scripting
  - Cross-Site Request Forgery
- Mobile Applications:
  - Insecure Data Storage
  - Reverse Engineering
  - Weak API Security



CAVEAT: Some vulnerabilities such as XSS will be reported by automated tools however...

Understanding Security Best Practices

+ Protecting User Data
+ Preventing Breaches
+ Building User Trust
+ Compliance Requirements

Key Areas of Security Best Practices

+ Secure Development Lifecycle (SDLC)
+ Data Protection
+ Authentication and Authorization
+ Secure Communication

Practical Tips for Security Implementation

+ Regular Updates
+ Code Security
+ Monitoring and Incident Response
+ Employee Training

Common Threats and Vulnerabilities in Mobile Apps

Why Are Mobile Apps Targeted?

+ The value of mobile data
+ Wide attack surface
+ Lack of developer awareness

Common Mobile Application Vulnerabilities: Insecure Data Storage

- Storing sensitive data like credentials or session tokens in plaintext.
- Common storage issues:
  - Using SharedPreferences or unencrypted SQLite databases.
- **Impact:** Data theft if the device is compromised.

Common Mobile Application Vulnerabilities: Insecure Communication

- Transmitting sensitive data over HTTP instead of HTTPS.
- Failing to validate SSL/TLS certificates.
- **Impact:** Data interception via man-in-the-middle (MITM) attacks.

Common Mobile Application Vulnerabilities: Weak Authentication and Authorization

- Hardcoded credentials in the app.
- Poor implementation of user authentication mechanisms.
- Missing access control for backend APIs.
- **Impact:** Unauthorized access and privilege escalation.

Common Mobile Application Vulnerabilities: Excessive Permissions

- Apps requesting unnecessary permissions, such as access to location or contacts.
- **Impact:** Exposure of sensitive data that the app doesn't need to function.

Common Mobile Application Vulnerabilities: Insecure APIs

- APIs with insufficient input validation or authentication.
- Overexposed endpoints accessible to unauthorized users.
- **Impact:** Data leaks, unauthorized transactions, or system compromise.

Common Mobile Application Vulnerabilities: Reverse Engineering

- Lack of obfuscation makes it easier for attackers to decompile and analyze apps.
- Extracting API keys or modifying app logic.
- **Impact:** Misuse of backend services or bypassing security measures.

Common Mobile Application Vulnerabilities: Outdated Components

- Using third-party libraries with known vulnerabilities.
- **Impact:** Exploitation of unpatched vulnerabilities in libraries or frameworks.

Why Vulnerabilities Persist?

+ Prioritization of Features Over Security
+ Lack of Security Awareness
+ Third-Party Dependencies
+ Testing Challenges

Mitigating Common Vulnerabilities

- Secure Data Storage
- Secure Communication
- Authentication and Authorization
- Code Security
- Third-Party Library Management

Bugs from Disclosed Bug Bounty and Research Reports

Overview of Bug Bounty Programs and Research Reports

- What are Bug Bounty Programs?
  - Platforms where security researchers report vulnerabilities for monetary rewards.
- Examples: HackerOne, Bugcrowd, Synack, and individual company programs.

Overview of Bug Bounty Programs and Research Reports

- Value of Research Reports:
  - Detailed analysis of vulnerabilities, often including proof-of-concept (PoC) exploits.
  - Published by independent researchers or cybersecurity firms.

Overview of Bug Bounty Programs and Research Reports

- Relevance to Mobile Penetration Testing:
  - Real-world insights into exploitation techniques.
  - Patterns in common vulnerabilities across apps.
  - Beware of “*fear mongers*”

Common Vulnerabilities Found in Bug Bounty Reports
Insecure Data Storage

- Case Study: A financial app storing authentication tokens in plaintext, leading to account takeovers.
- Exploitation: Accessing stored data via a rooted device or file extraction tools.
- Mitigation: Encrypt sensitive data using hardware-backed keystores.

Common Vulnerabilities Found in Bug Bounty Reports
Improper Authentication

- Case Study: A fitness app bypassing login mechanisms by manipulating API calls.
- Exploitation: Using tools like Burp Suite to modify requests.
- Mitigation: Enforce server-side authentication and validate session tokens.

Common Vulnerabilities Found in Bug Bounty Reports
API Misconfigurations

- Case Study: An e-commerce app exposing sensitive endpoints without authentication.
- Exploitation: Directly accessing endpoints via tools like Postman.
- Mitigation: Use proper authentication, rate limiting, and access controls.

Common Vulnerabilities Found in Bug Bounty Reports
Unvalidated Input Leading to Injection Attacks

- Case Study: A chat app vulnerable to NoSQL injection, allowing data exfiltration.
- Exploitation: Manipulating queries in API calls to fetch unauthorized data.
- Mitigation: Validate all inputs and use parameterized queries.

Common Vulnerabilities Found in Bug Bounty Reports
Code Obfuscation Failures

- Case Study: An entertainment app exposing hardcoded API keys in the decompiled APK.
- Exploitation: Reverse engineering the app.
- Mitigation: Use code obfuscation tools and avoid hardcoding sensitive data.

Tools and Techniques Used by Researchers

- Reverse Engineering Tools:
  - JADX, APKTool: Decompile and analyze Android APKs.
  - Ghidra: Reverse engineer compiled binaries.

Tools and Techniques Used by Researchers

- Network Analysis Tools:
  - Burp Suite: Intercept and modify network requests.
  - Wireshark: Capture and analyze network traffic.

Tools and Techniques Used by Researchers

- API Testing Tools:
  - Postman: Manually test API endpoints.
  - OWASP ZAP: Automate API vulnerability scans.

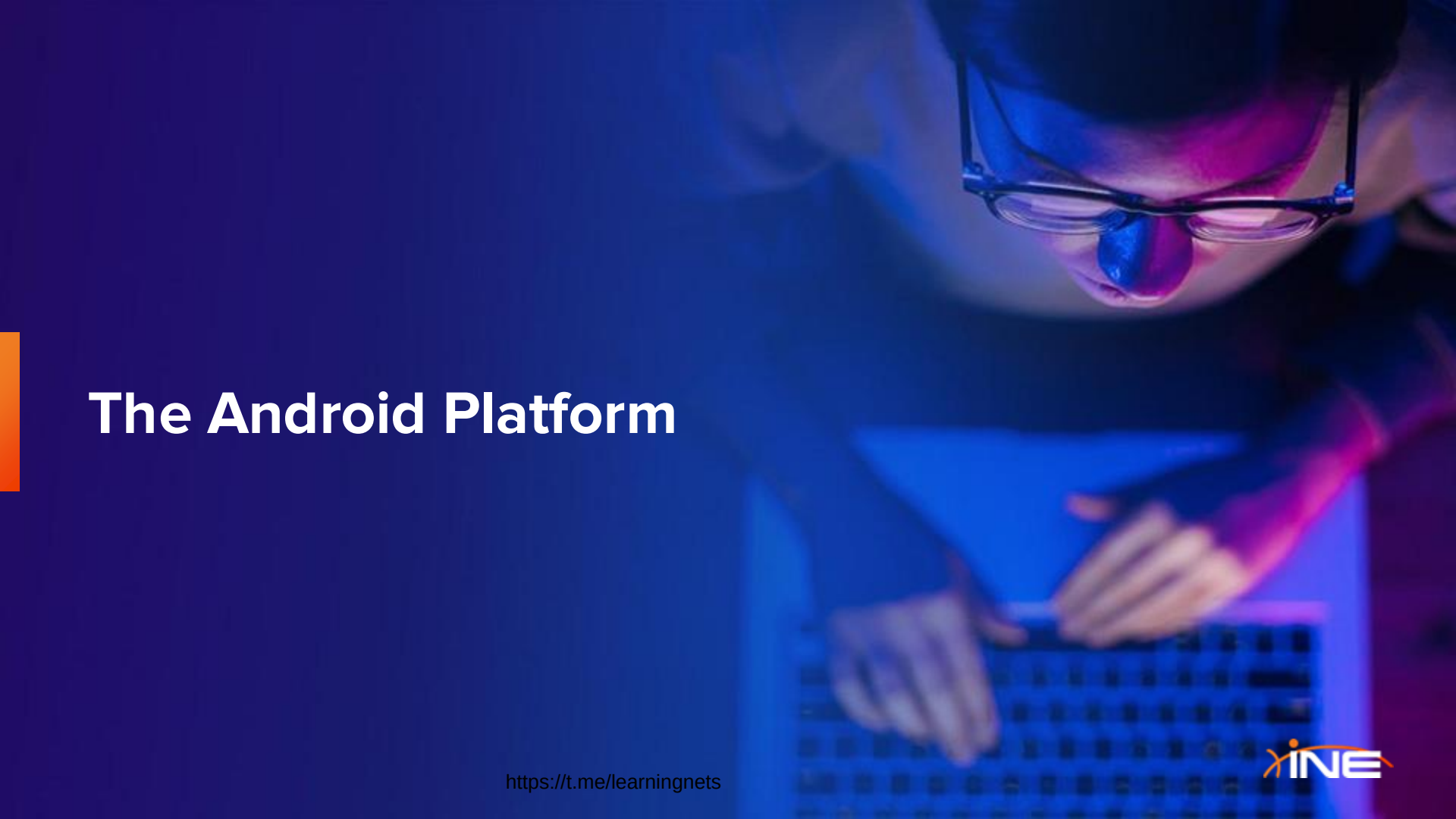
Tools and Techniques Used by Researchers

- Runtime Analysis Tools:
  - Frida: Inject scripts into apps to manipulate runtime behavior.
  - Xposed Framework: Customize app behavior on rooted devices.

Best Practices for Mobile Penetration Testing

Reading and understanding real world vulnerabilities will give you ideas on which vulnerabilities might affect the application your are testing.

The Android Platform

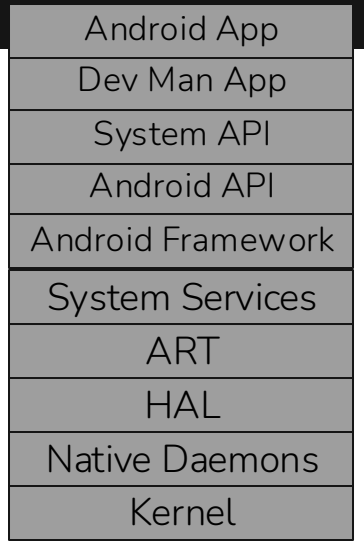


Android

- Cheaper = More widely used
- Open source = Available on more hardware
- Also easier to test!

Android

+ Linux-based and open source



Android Kernel

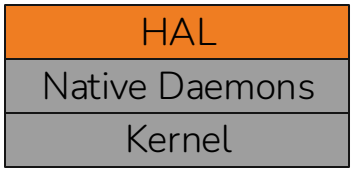
- Central part of the operating system
- Talks to the hardware
- Not really relevant for mobile application testing

Native Daemons and Libraries

+ Run in the background and are not under the control of the user
+ Interact directly with the kernel

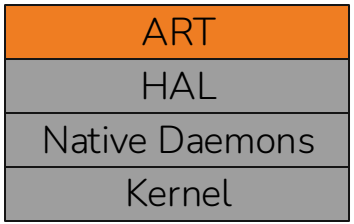
Hardware Abstraction Layer (HAL)

- HALs are usually implemented between the hardware and the software
- Allows hardware vendors to implement device-specific features without modifying higher-level layers



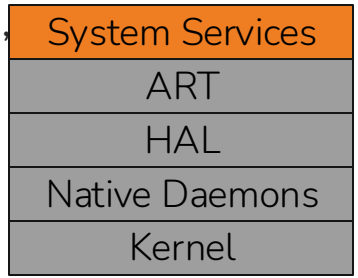
Android Runtime (ART)

+ Translates app's bytecode into processor-specific instructions



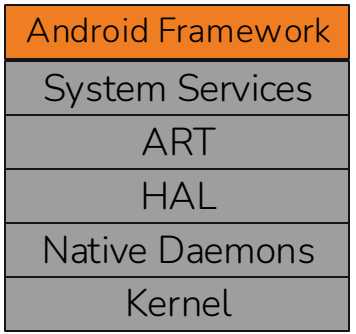
System Services

- Background process that communicate with lower-levels to allow the use of hardware components
- Also runs: power manager, account manager, battery service, alarm manager, clipboard service, notification manager, etc.
- "*Heart of Android*"



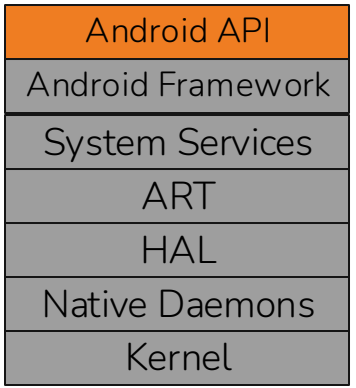
Android Framework

- Java classes, interfaces, other compiled code on which apps are built
- Partially exposed through the Android API



Android API

+ Publicly available API for Android app developers



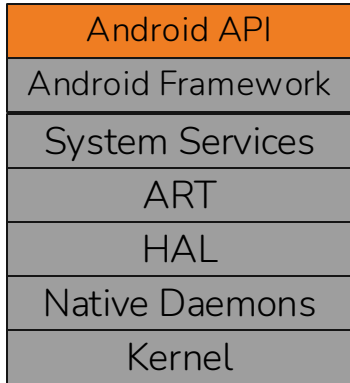
Recap

```
package com.example.cameralist

import android.hardware.camera2.CameraManager
import android.os.Bundle
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val textView: TextView = findViewById(R.id.camera_list)
        val cameraManager = getSystemService(CAMERA_SERVICE) as CameraManager
        val cameraIds = cameraManager.cameraIdList

        textView.text = "Available Cameras:\n" + cameraIds.joinToString("\n")
    }
}
```


Recap

```
package com.example.cameralist

import android.hardware.camera2.CameraManager
import android.os.Bundle
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val textView: TextView = findViewById(R.id.camera_list)
        val cameraManager = getSystemService(CAMERA_SERVICE) as CameraManager
        val cameraIds = cameraManager.cameraIdList

        textView.text = "Available Cameras:\n" + cameraIds.joinToString("\n")
    }
}
```

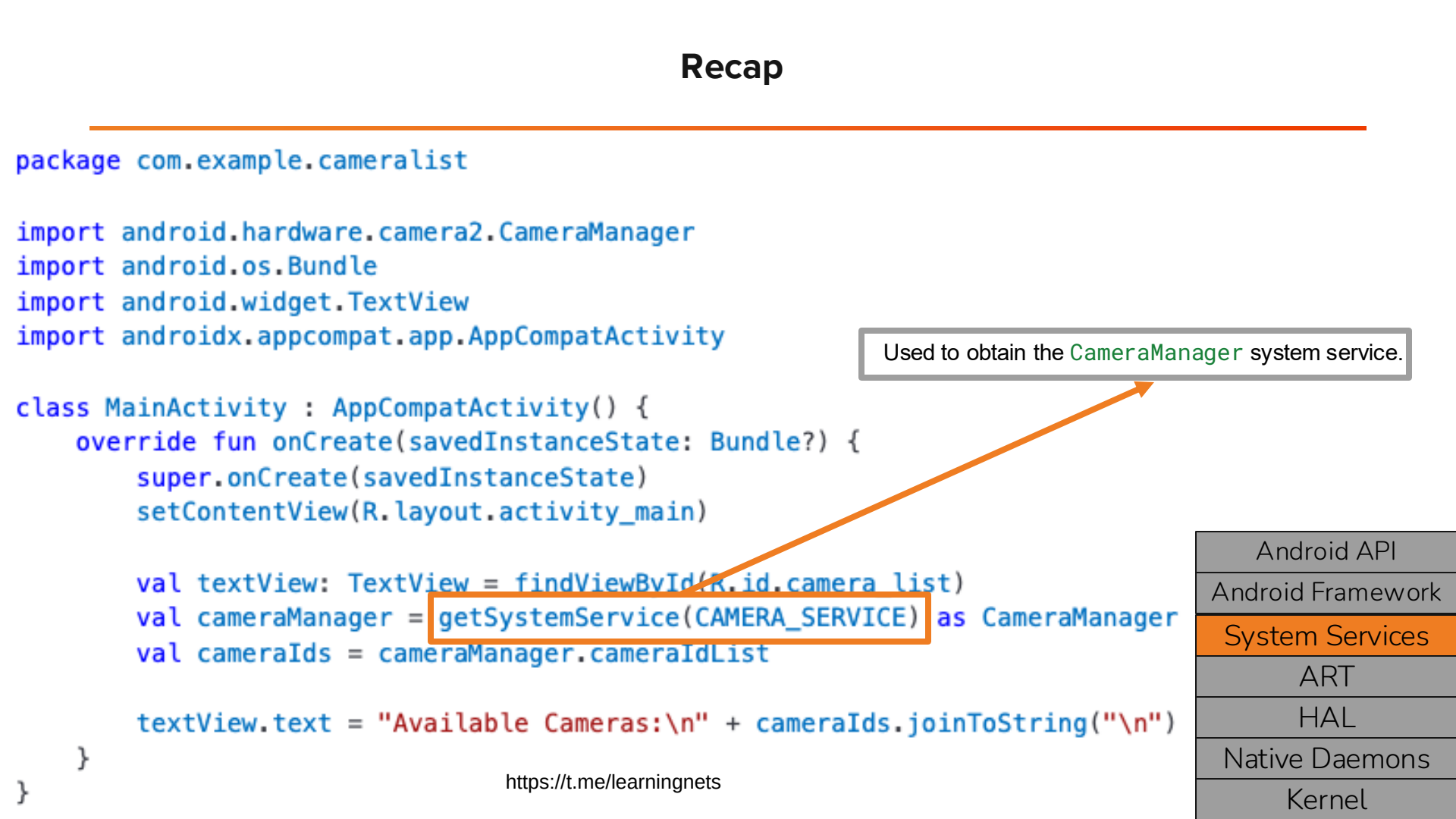
Used to access camera devices and retrieve the list of available cameras on the device.



Recap

```
package com.example.cameralist

import android.hardware.camera2.CameraManager
import android.os.Bundle
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val textView: TextView = findViewById(R.id.camera_list)
        val cameraManager = getSystemService(CAMERA_SERVICE) as CameraManager
        val cameraIds = cameraManager.cameraIdList

        textView.text = "Available Cameras:\n" + cameraIds.joinToString("\n")
    }
}
```

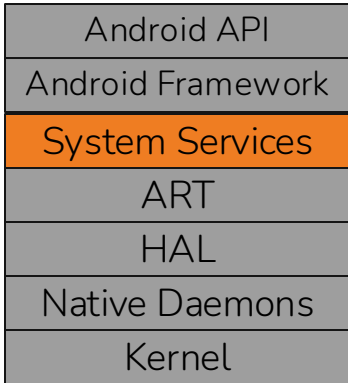
Used to obtain the `CameraManager` system service.



Recap

```
package com.example.cameralist

import android.hardware.camera2.CameraManager
import android.os.Bundle
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val textView: TextView = findViewById(R.id.camera_list)
        val cameraManager = getSystemService(CAMERA_SERVICE) as CameraManager
        val cameraIds = cameraManager.cameraIdList

        textView.text = "Available Cameras:\n" + cameraIds.joinToString("\n")
    }
}
```

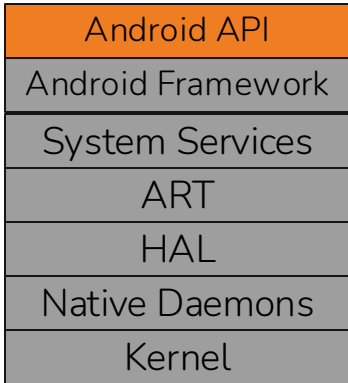
Retrieves the list of available camera IDs.



Recap

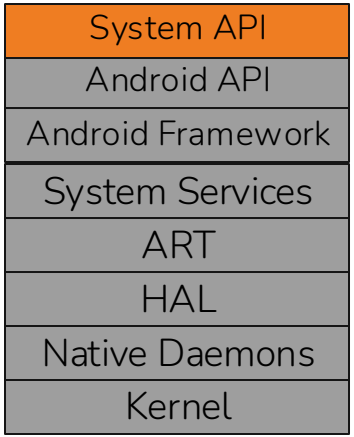
```
package com.example.cameralist

import android.hardware.camera2.CameraManager
import android.os.Bundle
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val textView: TextView = findViewById(R.id.camera_list)
        val cameraManager = getSystemService(CAMERA_SERVICE) as CameraManager
        val cameraIds = cameraManager.cameraIdList

        textView.text = "Available Cameras:\n" + cameraIds.joinToString("\n")
    }
}
```

Formats and sets the text of a `TextView` (`R.id.camera_list`) to display the camera IDs.



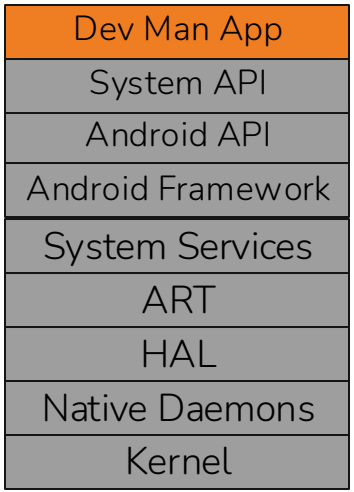
System API

+ Only available to partners and OEMs



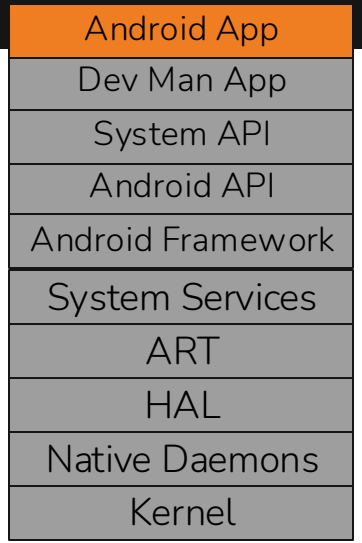
Device Manufacturer App

- Apps created using:
  - Android API
  - System API
  - Direct access to the Android Framework
- Apps are preinstalled by manufacturer
- Similar to “privileged apps”
  - Created using Android and System API

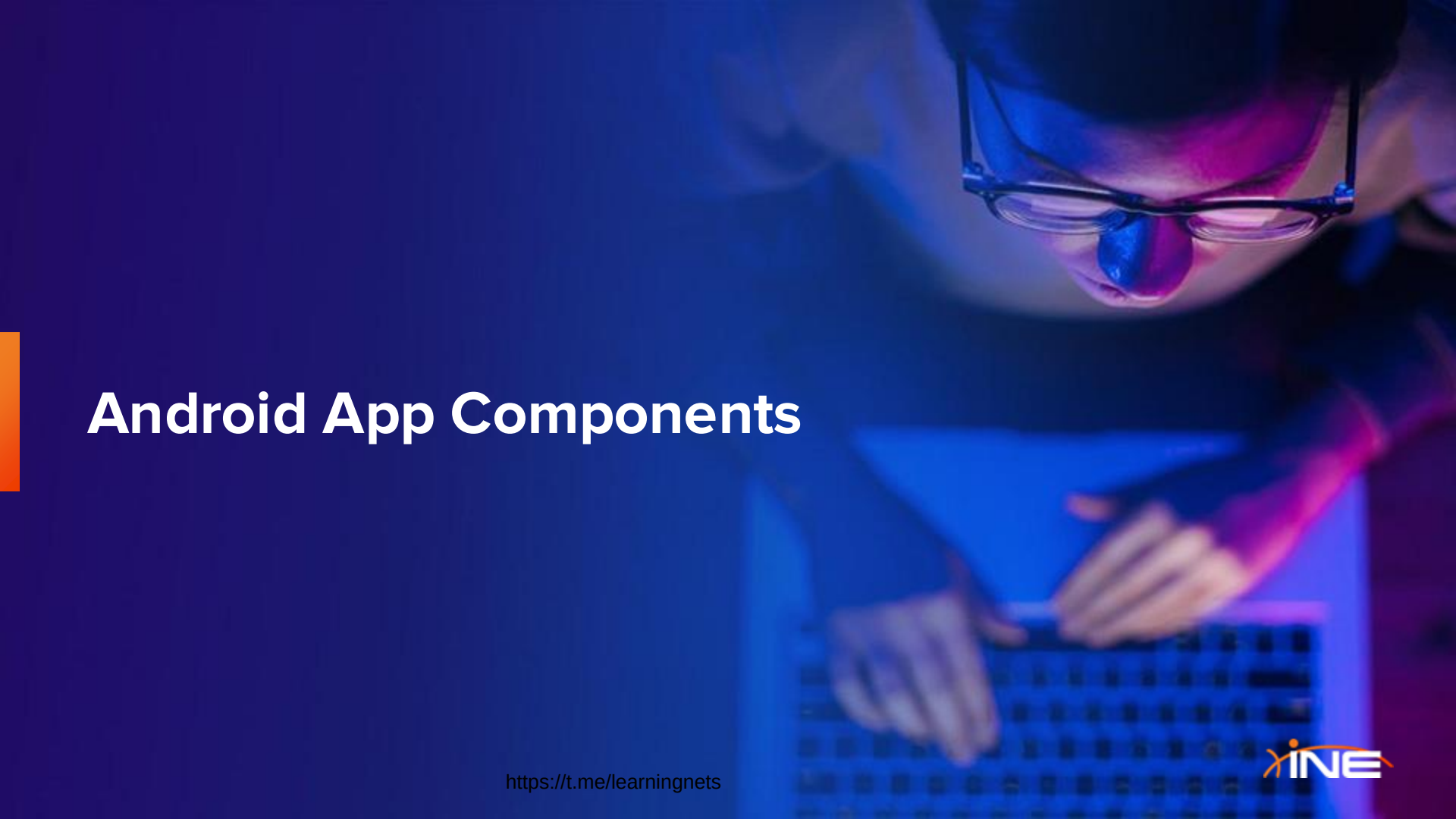


Android App

- Apps created using the Android API
- Usually you'll be testing these



Android App Components



Android Security Features

- Defense-in-depth approach
- Four domains of Android security:
  - System-wide security
  - Software isolation
  - Network security
  - Anti-exploitation

Android Security Features

- System-wide security:
  - Device encryption:
    - Full disk encryption: Android 5.0 and above
    - File-based encryption: Different files can be encrypted using different keys. Usually supports Direct Boot (e.g. alarms work without unlocking the device)

Android Security Features

- System-wide security:
  - Device encryption
  - Trusted Execution Environment (TEE):
    - Securely generate and protect keys

Android Security Features

- System-wide security:
  - Device encryption
  - Trusted Execution Environment (TEE)
  - Verified boot:
    - Ensures all executed code comes from a trusted source
    - Each stage of the booting process verifies the integrity and authenticity of the next stage before handing over execution.

Android Security Features

- Software isolation:
  - Android is Linux but handles user accounts differently
  - Multi-user support is used for application sandboxing
  - Each app runs as a different user

```
/* This is the master Users and Groups config for the platform.
 * DO NOT EVER RENUMBER
 */

#define AID_ROOT 0 /* traditional unix root user */
/* The following are for LTP and should only be used for testing */
#define AID_DAEMON 1 /* traditional unix daemon owner */
#define AID_BIN 2    /* traditional unix binaries owner */

#define AID_SYSTEM 1000 /* system server */

#define AID_RADIO 1001          /* telephony subsystem, RIL */
#define AID_BLUETOOTH 1002      /* bluetooth subsystem */
#define AID_GRAPHICS 1003       /* graphics devices */
#define AID_INPUT 1004          /* input devices */
#define AID_AUDIO 1005          /* audio devices */
#define AID_CAMERA 1006         /* camera devices */
#define AID_LOG 1007            /* log devices */
#define AID_COMPASS 1008        /* compass device */
#define AID_MOUNT 1009          /* mountd socket */
#define AID_WIFI 1010           /* wifi subsystem */
#define AID_ADB 1011            /* android debug bridge (adbd) */
#define AID_INSTALL 1012        /* group for installing packages */
#define AID_MEDIA 1013          /* mediaserver process */
#define AID_DHCP 1014           /* dhcp client */
#define AID_SDCARD_RW 1015      /* external storage write access */
#define AID_VPN 1016            /* vpn system */
#define AID_KEYSTORE 1017       /* keystore subsystem */
#define AID_USB 1018            /* USB devices */
#define AID_DRM 1019            /* DRM server */
#define AID_MDNSR 1020          /* MulticastDNSResponder (service discovery) */
#define AID_GPS 1021            /* GPS daemon */
#define AID_UNUSED1 1022        /* deprecated, DO NOT USE */
#define AID_MEDIA_RW 1023       /* internal media storage write access */
#define AID_MTP 1024            /* MTP USB driver access */
#define AID_UNUSED2 1025        /* deprecated, DO NOT USE */
#define AID_DRMRPC 1026         /* group for drm rpc */
#define AID_NFC 1027            /* nfc subsystem */
```

http://androidxref.com/9.0.0_r3/xref/system/core/include/private/android_filesystem_config.h

Sandbox

- Apps run in sandboxes
  - Each app's data and executable is isolated from others
- Developers can share resources between apps
  - If signed by the same certificate
  - If *sharedUserId* is added to the *AndroidManifest.xml* file

Sandbox

- Users are added to groups when permissions are granted

```
<!-- The following tags are associating low-level group IDs with
     permission names.  By specifying such a mapping, you are saying
     that any application process granted the given permission will
     also be running with the given group ID attached to its process,
     so it can perform any filesystem (read, write, execute) operations
     allowed for that group. -->

<permission name="android.permission.BLUETOOTH_ADMIN" >
    <group gid="net_bt_admin" />
</permission>

<permission name="android.permission.BLUETOOTH" >
    <group gid="net_bt" />
</permission>

<permission name="android.permission.BLUETOOTH_STACK" >
    <group gid="bluetooth" />
    <group gid="wakelock" />
    <group gid="uhid" />
</permission>

<permission name="android.permission.NET_TUNNELING" >
    <group gid="vpn" />
</permission>

<permission name="android.permission.INTERNET" >
    <group gid="inet" />
</permission>
```

http://androidxref.com/9.0.0_r3/xref/frameworks/base/data/etc/platform.xml

How Does An App Run?

1. User taps on icon: Android's Activity Manager (part of the System Server) handles the request
2. Android System Checks if the App is Already Running
   + If the app is already running, the existing process is brought to the foreground.
   + If the app is not running, a new process needs to be created.
3. Zygote does its thing (!)
4. The Activity Manager Service (AMS) in the System Server coordinates the app launch.
5. If memory is low, the Android OS may kill background processes to free up resources.
6. If the user closes the app, the process may remain in memory for faster relaunch.

Zygote

- Zygote is a pre-initialized process that serves as the "parent" of all Android app processes.
- It is started during device boot-up and loads essential libraries, system classes, and the Dalvik/ART runtime.
- The idea is to fork new app processes from Zygote instead of starting from scratch, which significantly speeds up app launches.

Zygote

- Zygote is Started at Boot Time
  - It loads common framework classes (android.app.*, android.os.*, android.view.*).
  - It preloads shared libraries (like graphics, networking, media, etc.).
  - It initializes a virtual machine (ART or Dalvik).

Zygote

- Forks a New App Process
  - When a new app is launched, Zygote forks itself to create a new child process.
  - The child process inherits the preloaded classes and resources.
  - This reduces startup time and memory usage because many components are shared (copy-on-write mechanism).

Application States

- A Linux process is created when an app component is started
- Android may kill a process to free resources
- The process can be in one of these states:
  - Foreground process: Running on the screen
  - Visible process: Killing it may impact user experience
  - Service process: Not directly seen by the user but relevant
  - Cached process: Not currently needed

Android Security Features

- Software isolation:
  - Users
  - SELinux:
    - Android uses Security-Enhanced Linux (SELinux)
    - Enforce mandatory access control (MAC) over all processes, even processes running with root/superuser privileges
    - Works on a least privilege principle

Android Security Features

- Software isolation:
  - Users
  - SELinux
  - Permissions:
    - Install-time permissions
    - Normal permissions
    - Runtime permissions
    - Special permissions
    - Signature permissions

Android Security Features

- Network security
  - TLS by default
  - DNS over TLS

Android Security Features

- Anti-exploitation
  - ASLR: Address Space Layout Randomization
  - KASLR: Kernel Address Space Layout Randomization
  - PIE: Position Independent Executable
  - DEP: Data Execution Prevention

Android Security Features

- Anti-exploitation
  - SECCOMP: Secure Computing
  - Filters system calls made by user applications

How are Apps Shipped?

- APK: Android Package Kit
- AAB: Android App Bundle
  - Includes all the compiled code and resources
  - Google Play uses AAB to generate optimized APKs

Android Manifest

- Named AndroidManifest.xml
- Located in the root directory of the APK file
- Describes:
  - The components of the app
  - The permissions that the app needs
  - The hardware and software features the app requires

App Components

- Activities
- Services
- Broadcast Receivers
- Content Providers

Activities

- Represents a single screen with a user interface
  - Keeps track of what the user cares about
  - Keeps track of recently used processes
  - User can returned to activities with the previous state restored

Services

- General-purpose entry point for keeping an app running in the background
  - Playing music in the background while user uses another activity
- Types of services:
  - Started services
  - Bound services

Services

- Started services:
  - Tells the system to keep this running until work is completed
  - Music playing in the background: System won't kill this process
  - Regular background services can be killed and restarted without the user noticing it

Services

- Bound services:
  - Provides API to another process
  - The system keeps track of bound services so it doesn't kill a needed service

Broadcast Receivers

- Lets the system deliver events to the app
- The system can deliver broadcasts to apps that aren't running
- E.g. alarms
  - You don't need to keep the alarm or clock app open until the alarm time
- Apps can also initiate broadcasts

Content Providers

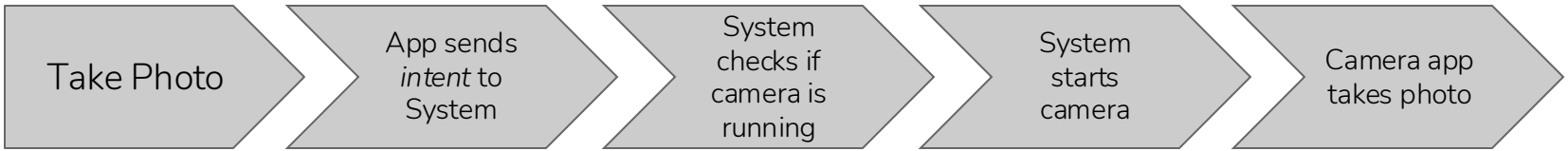
- Manages a shared set of app data
- E.g. contact information is a content provider. Any app with permission can access the contacts list.

Activate Components

- Intents are asynchronous messages
- Intents can activate: activities, services, broadcast receivers
- Content providers are activated with `ContentResolver`

Apps Can Start Other App's Components

+ E.g. taking a photo using a social media app



How it works?

```
    captureButton.setOnClickListener {
        val takePictureIntent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
        if (takePictureIntent.resolveActivity(packageManager) != null) {
            startActivityForResult(takePictureIntent, REQUEST_IMAGE_CAPTURE)
        }
    }
}

override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == Activity.RESULT_OK) {
        val imageBitmap = data?.extras?.get("data") as Bitmap
        imageView.setImageBitmap(imageBitmap)
    }
}
```

How it works?

```
    captureButton.setOnClickListener {
        val takePictureIntent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
        if (takePictureIntent.resolveActivity(packageManager) != null) {
            startActivityForResult(takePictureIntent, REQUEST_IMAGE_CAPTURE)
        }
    }
}

override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == Activity.RESULT_OK) {
        val imageBitmap = data?.extras?.get("data") as Bitmap
        imageView.setImageBitmap(imageBitmap)
    }
}
```

`Intent(MediaStore.ACTION_IMAGE_CAPTURE)` asks the system to open the camera app.

How it works?

```
    captureButton.setOnClickListener {
        val takePictureIntent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
        if (takePictureIntent.resolveActivity(packageManager) != null) {
            startActivityForResult(takePictureIntent, REQUEST_IMAGE_CAPTURE)
        }
    }
}

override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == Activity.RESULT_OK) {
        val imageBitmap = data?.extras?.get("data") as Bitmap
        imageView.setImageBitmap(imageBitmap)
    }
}
```

Ensures there’s an available camera app before starting

How it works?

```
    captureButton.setOnClickListener {
        val takePictureIntent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
        if (takePictureIntent.resolveActivity(packageManager) != null) {
            startActivityForResult(takePictureIntent, REQUEST_IMAGE_CAPTURE)
        }
    }
}

override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == Activity.RESULT_OK) {
        val imageBitmap = data?.extras?.get("data") as Bitmap
        imageView.setImageBitmap(imageBitmap)
    }
}
```

Launches the camera and waits for a result

How it works?

```
    captureButton.setOnClickListener {
        val takePictureIntent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
        if (takePictureIntent.resolveActivity(packageManager) != null) {
            startActivityForResult(takePictureIntent, REQUEST_IMAGE_CAPTURE)
        }
    }
}

override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == Activity.RESULT_OK) {
        val imageBitmap = data?.extras?.get("data") as Bitmap
        imageView.setImageBitmap(imageBitmap)
    }
}
```

Retrieves the image and displays it

AndroidManifest.xml

- Identifies any user permissions the app requires
- Declares the minimum API level required by the app
- Declares hardware and software features used or required by the app
- Declares API libraries the app needs to be linked against (e.g. the Google Maps library).

AndroidManifest.xml

- <activity> for each subclass of Activity
- <service> for each subclass of Service
- <receiver> for each subclass of BroadcastReceiver
- <provider> for each subclass of ContentProvider
- etc..

"Manifesting the manifesto"

Let's Read a Manifest File

Manifest Analysis Methodology

- Permissions
- Exported components
- Debuggable apps
- Backup enabled
- Cleartext traffic
- Grants URI use

Risky Permissions

- Device Administration & Accessibility Exploits
  - BIND_DEVICE_ADMIN: Grants Device Admin rights, which can be abused to lock the user out, wipe data, or prevent uninstallation.
  - BIND_ACCESSIBILITY_SERVICE: Grants full screen reading & input control—often abused by malware (e.g., keyloggers, banking trojans).

Risky Permissions

- Data Theft & Privacy Risks:
  - READ_CONTACTS
  - READ_SMS / RECEIVE_SMS / SEND_SMS
  - READ_CALL_LOG / WRITE_CALL_LOG
  - RECORD_AUDIO
  - READ_PHONE_STATE
  - ACCESS_FINE_LOCATION / ACCESS_COARSE_LOCATION

Risky Permissions

- File System & Storage Exploits:
  - READ_EXTERNAL_STORAGE: Allows an app to access all user files, including sensitive documents and images.
  - WRITE_EXTERNAL_STORAGE: Lets an app modify or delete files, including altering system logs or malware injection.

Risky Permissions

- Background & Hidden Activities:
  - SYSTEM_ALERT_WINDOW: Allows an app to display overlays, which is often used in clickjacking/phishing attacks.
  - BIND_NOTIFICATION_LISTENER_SERVICE: Lets an app intercept notifications, often used by spyware.
  - FOREGROUND_SERVICE: Allows running background services without user awareness, commonly used in spyware.
  - REQUEST_IGNORE_BATTERY_OPTIMIZATIONS: Lets an app bypass Doze Mode, keeping it running in the background indefinitely.

Risky Permissions

- Network & Remote Exploitation:
  - INTERNET: Grants full network access—used for data exfiltration and C2 communication.
  - ACCESS_NETWORK_STATE : Allows an app to monitor network status, used by malware to detect active connections.
  - CHANGE_WIFI_STATE: Lets an app enable/disable Wi-Fi.

Risky Permissions

- Critical System Permissions:
  - WAKE_LOCK: Allows an app to prevent sleep mode, often used in battery-draining malware.
  - REBOOT: Lets an app force restart the device, used in ransomware attacks.
  - CHANGE_CONFIGURATION: Allows changing system settings like keyboard.
  - INSTALL_PACKAGES: Allows an app to silently install other apps, often used in malware droppers.
  - DELETE_PACKAGES: Allows an app to uninstall other apps, often used in targeted attacks.

Risky Permissions

- Read: “Uraniborg’s Device Preloaded App Risks Scoring Metric” for a more detailed list.
- https://www.android-device-security.org/publications/2020-lau-uraniborg/Lau_2020_Uraniborg_Scoring_Whitepaper_20200827.pdf

Permission Category	Weight	Permission Name
ASTRONOMICAL	100	android.permission.INSTALL_PACKAGES
CRITICAL	10	android.permission.COPY_PROTECTED_DATA android.permission.WRITE_SECURE_SETTINGS android.permission.READ_FRAME_BUFFER android.permission.MANAGE_CA_CERTIFICATES android.permission.MANAGE_APP_OPS_MODES android.permission.GRANT_RUNTIME_PERMISSIONS android.permission.DUMP android.permission.CAMERA

Exported Components

- android:exported="true"
- When an app component is exported, it can be accessed by other apps, including potentially malicious ones.
- Malicious apps can invoke or exploit exported components to steal data, manipulate app behavior, or escalate privileges.

Exported Components

Component	Risk	Mitigation
Exported Activities	UI hijacking, intent injection	Set android:exported="false" unless necessary
Exported Services	Remote code execution	Restrict with permissions
Exported Broadcast Receivers	Intent hijacking	Use explicit broadcasts or restrict exported="false"
Exported Content Providers	Data leakage	Restrict access with exported="false" or permissions

Debuggable Apps

- Allows an attacker to attach a debugger (adb) and inspect runtime memory, methods, and sensitive data.
- Can be exploited to modify execution flow, bypass authentication, or steal data.

```
<application android:name=".MyApp"
    android:debuggable="true">
```

Backup Enabled

- Allows an attacker to extract sensitive data using:

```
adb backup -f backup.ab com.example.myapp
```

- Even if data is stored securely, an attacker can restore it on another device.

```
<application android:allowBackup="true">
```

Cleartext Traffic Allowed

+ Allows unencrypted HTTP communication, which can be intercepted.
+ Attackers can modify network traffic, inject malware, or steal user data.

```
<application android:usesCleartextTraffic="true">
```

URI Permissions

- Other apps can access sensitive files.
- Attackers may modify files leading to data manipulation or injection attacks.

```
<provider android:authorities="com.example.myapp.fileprovider"
    android:grantUriPermissions="true">
```

URI Permissions

- URI permissions allow temporary access to a file or content provider without exposing full storage access.
- Granted using FLAG_GRANT_READ_URI_PERMISSION and FLAG_GRANT_WRITE_URI_PERMISSION
- If misconfigured, any app can access or modify the shared file.

URI Permissions

- Data Theft:
- File Tampering
- Privilege Escalation
- Bypassing App Sandboxing
- Arbitrary Code Execution

URI Permissions

- Data theft via open content providers: A publicly exported provider (android:exported="true") can expose private app data.

```
<provider android:name=".UserProvider"
    android:authorities="com.example.app.provider"
    android:grantUriPermissions="true"
    android:exported="true"/>
```

```
Cursor cursor = getContentResolver().query(
    Uri.parse("content://com.example.app.provider/users"),
    null, null, null, null);
```

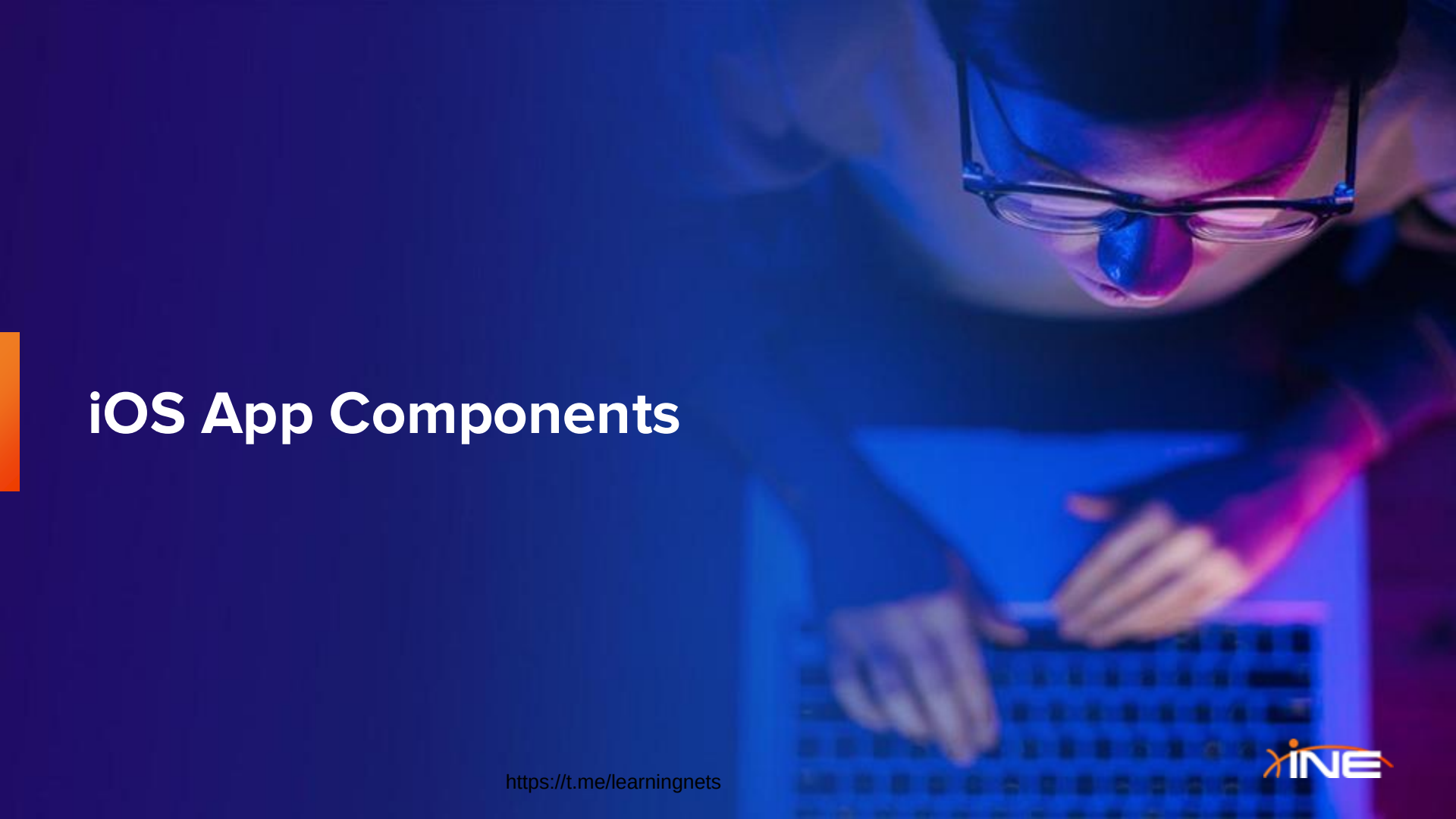
URI Permissions

- File tampering with open file providers: Using FileProvider with grantUriPermissions="true" allows modification of shared files.

```
<provider android:name="androidx.core.content.FileProvider"
    android:authorities="com.example.app.fileprovider"
    android:exported="true"
    android:grantUriPermissions="true">
</provider>
```

```
getContentResolver().delete(
    Uri.parse("content://com.example.app.fileprovider/shared.txt"),
    null, null);
```

iOS App Components



iOS

- iOS apps run in a more restricted environment
- iOS apps are isolated
- System API access limited
- Offers few IPC (Inter Process Communication)

Apple Platform Security

- Hardware security and biometrics
- System security
- Encryption and Data Protection
- App security
- Services security
- Network security
- Developer kit security

iOS Security Architecture

- Many security features are etch on the silicon
- Boot ROM is read-only and ensures Low Level Bootloader security
- If any signature check fails, the boot process stops
- Secure boot chain ensures boot integrity by verifying that only Apple-signed software runs on the device

Code Signing

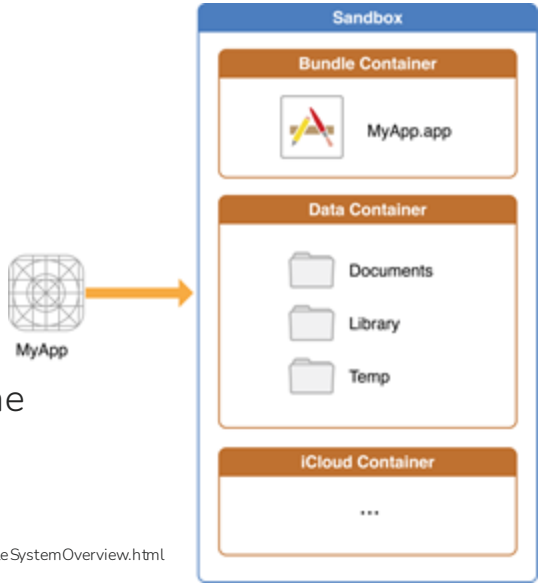
- Only Apple-approved code runs on devices
- Developers can compile and deploy apps on a single device via sideloading

Encryption and Data Protection

- A private - public key pair is created for every Apple user ID
- Apple has dedicated hardware cryptographic engine
- File-based encryption using device's UID (burned into the chip) and passcode

Sandbox

- iOS app's interactions with the file system are limited to the directories inside the app's sandbox directory
- An app is generally prohibited from accessing or creating files outside its container directories
- One exception to this rule is when an app uses public system interfaces to access things such as the user's contacts or music



https://developer.apple.com/library/archive/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html

Sandbox

- Sandbox is enforced at the kernel level
- All third party apps run under the same user
- Processes are isolated from each other
- The app process is restricted to its own directory
- Hardware drivers can only be accessed through public frameworks

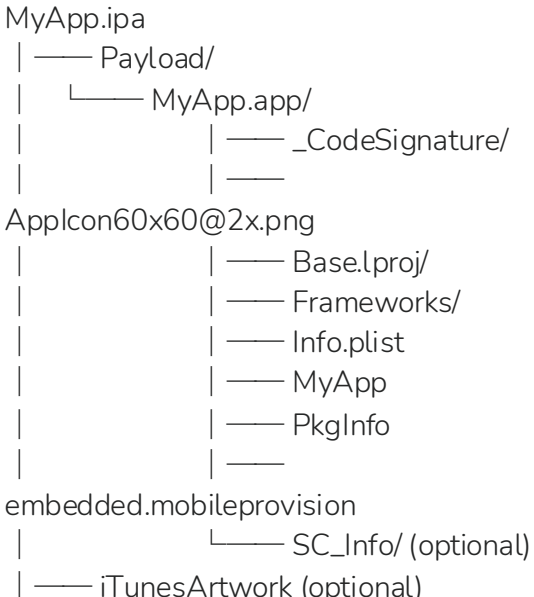
Exploit Mitigations

- ASLR
- eXecute Never (XN) bit: iOS never allows an app to execute from writeable memory

iOS Apps

- Apps are distributed in IPA (iOS App Store Package) format
- IPA = Compressed zip file
- Contains all code and resources required by the app

IPA File Structure



- Payload/ : Main Application Directory
- MyApp.app/ : Main Application Bundle
- _CodeSignature/ : Contains cryptographic signatures of the app
- Info.plist : The main configuration file for the app
- MyApp : Executable Binary
- Embedded.mobileprovision : Links the app to a specific Apple Developer account.
- Frameworks/ : Stores dynamic frameworks (shared libraries) used by the app
- Base.lproj/ : Stores UI elements

iOS App Permissions

- User asked to grant permission at runtime
- “Privacy Settings” gives control over app permissions
  - Location Services, Contacts, Calendars, Reminders, Photos, Bluetooth, Local Network, Microphone, Camera, Files and Folders, etc.
- DeviceCheck services:
  - Can help identify devices that have already redeemed a coupon
  - App Attest service: Adds hardware supported checks that can be used to detect unauthorized features like game cheats, ad removal, or access to premium content

"Infoing the info.plist"

Let's Read an info.plist File

Info.plist Analysis Methodology

- Extracting info.plist
- Debugging & Security Flags
- Network & Data Security Risks
- Privacy & Data Exposure Risks
- App Execution & Inter-Process Communication Risks
- Third-Party SDKs & Data Sharing Risks

Info.plist: Debugging & Security Flags

Setting	Risk
UIFileSharingEnabled	If true, allows access to app files via iTunes File Sharing, enabling data theft.
LSSupportsOpeningDocumentsInPlace	Allows an app to modify files in external locations (e.g., iCloud, other apps). Attackers can overwrite or read sensitive data.
UIApplicationExitsOnSuspend	If true, prevents background execution, improving privacy. If false, app data remains in memory when suspended.
UIBackgroundModes	If audio, fetch, location, or remote-notification is enabled, the app runs in background mode, increasing risk of data leaks.
ITSAppUsesNonExemptEncryption	If false, the app may use weak or no encryption, leading to compliance issues.
UIApplicationSupportsIndirectInputEvents	If true, the app may process inputs from external devices, leading to keylogging risks.

Info.plist: Network & Data Security Risks

Setting	Risk
NSAllowsArbitraryLoads (ATS - App Transport Security)	If true, allows insecure HTTP connections, leading to MITM attacks.
NSExceptionDomains	Specifies exceptions to ATS. If NSIncludesSubdomains = true, subdomains are also exposed.
NSAllowsLocalNetworking	If true, allows communication with local network devices (e.g., smart home devices). Could be exploited for data exfiltration.
NSAppTransportSecurity	Overrides ATS for specific hosts. If improperly configured, allows unencrypted traffic.
NSContactsUsageDescription	If the app accesses contacts, an attacker might abuse it for social engineering or data leaks.

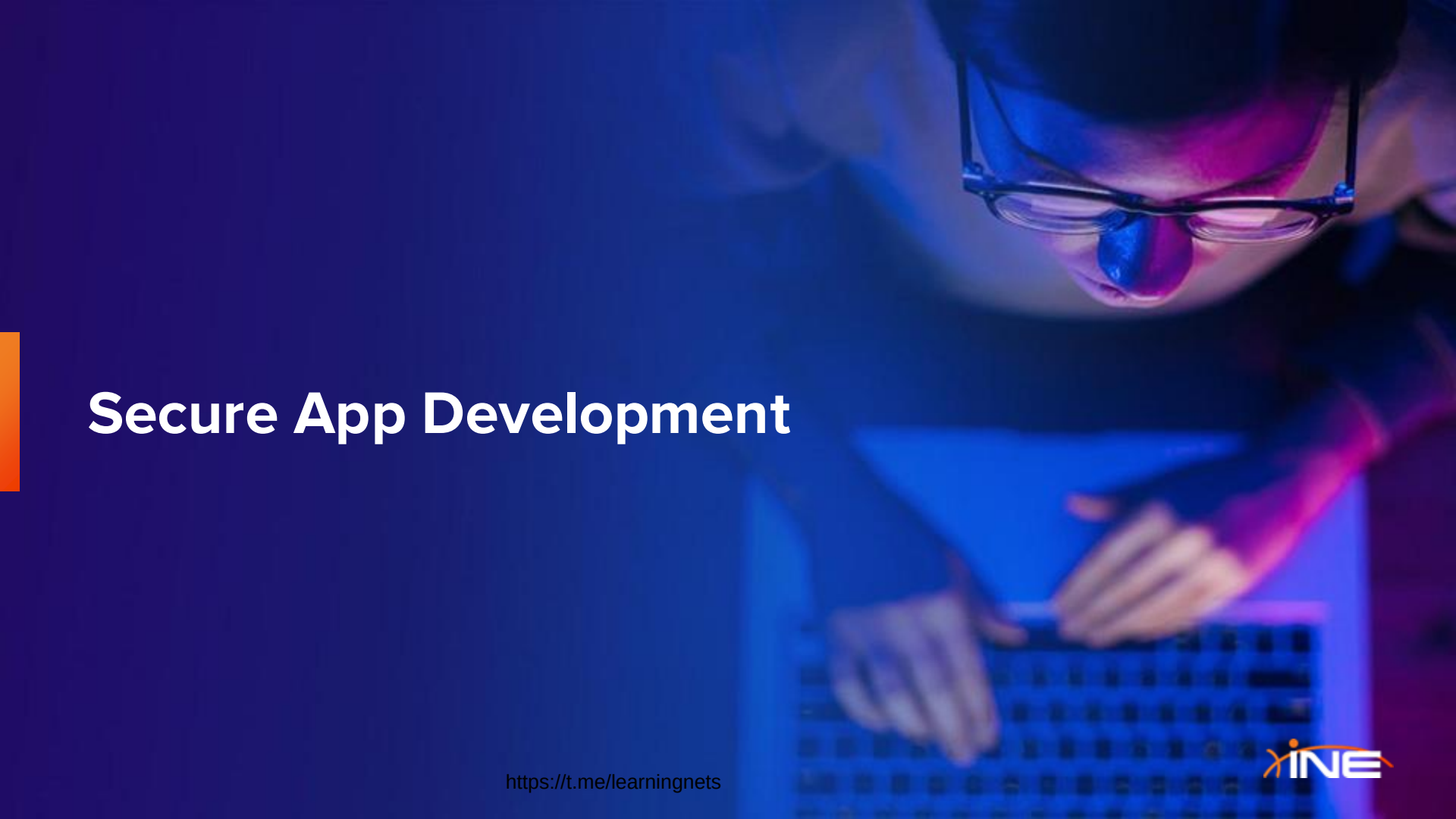
Info.plist: Privacy & Data Exposure Risks

Setting	Risk
NSCameraUsageDescription	If present, app requests camera access. A malicious app might spy on users.
NSMicrophoneUsageDescription	If present, app requests microphone access, leading to covert eavesdropping risks.
NSPhotoLibraryUsageDescription	Allows app to read user photos, which could be misused for data extraction.
NSLocationAlwaysUsageDescription	Allows continuous location tracking, which can leak sensitive user movements.
NSUserTrackingUsageDescription	Required for tracking across apps & websites. If present, the app tracks user behavior.

Info.plist: App Execution & IPC Risks

Setting	Risk
CFBundleURLTypes (Deep Linking)	If the app registers a custom URL scheme (e.g., myapp://), it can be exploited for deep link hijacking.
CFBundleExecutable	Defines the binary name of the app. Attackers can replace the binary in tampering attacks.
CFBundleIdentifier	The app's unique identifier. Attackers might clone the app for phishing attacks.
UIRequiresFullScreen	Forces the app into full-screen mode, hiding security warnings (e.g., during biometric authentication).
LSApplicationQueriesSchemes	Allows the app to query other installed apps, which can be abused for fingerprinting or tracking users.

Secure App Development



Mobile Application Development Process

1. Idea and Planning
2. Design
3. Development
4. Testing
5. Deployment

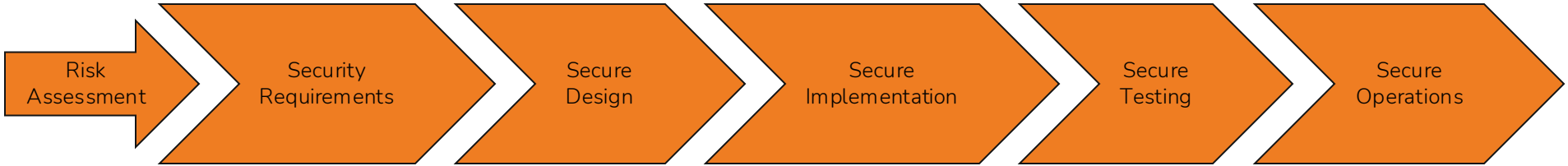
Mobile Application Development Process

Mobile Application Development Process

Secure SDLC

1. Perform a risk assessment
2. List security requirements
3. Threat modeling
4. Develop securely
5. Security testing

Secure SDLC



Mobile Application Development Process

1. Idea and Planning => Developing a security plan
2. Design => Secure design (least privilege, security roles, etc.)
3. Development => Secure coding
4. Testing => Security testing
5. Deployment => Secure deployment (package security, code signing, etc.)

App Store Review and Publishing

- App store requirements
- Submission
- Review
- Approval

App Store Review and Publishing

- App store requirements
  - Google: Bigger focus on functionality, design, and policy compliance
  - Apple: Bigger focus on security, privacy, and design consistency
- Submission process:
  - Create developer account
  - Upload app store assets (screenshots, description, etc.)
  - App submission

App Store Review and Publishing

- Review process:
  - Google: Mostly automated checks
  - Apple: Manual review
- Approval and deployment:
  - App goes live
  - Analytics and user feedbacks shared with developers

Mobile Application Maintenance

+ Regular updates
+ Monitoring and analytics
+ User feedback

Tools and Software for Mobile Penetration Testing

Tools != Techniques

+ Know what you are trying to accomplish
+ Know how the target works
+ Know how the tool works

Static Analysis Tools

- What is static analysis?
  - Analyzing an application's source code, binary, or intermediate code **without executing it**

Tool Categories

- Static analysis
- Dynamic analysis
- Network analysis
- Reverse engineering

Static Analysis Tools

- What do we expect from static analysis?
  - Sensitive information (usernames, passwords, etc.)
  - Application endpoints (APIs, URLs, etc.)
  - A better understanding of the application
  - An overview of security practices

Static Analysis Tools

- Command line tools (strings, grep, etc.)
- IDE
- MobSF
- JADX-GUI
- Apktool
- Otool

Dynamic Analysis Tools

- What is dynamic analysis?
  - Analyzing an application by **executing the app binary**

Dynamic Analysis Tools

- What do we expect from dynamic analysis?
  - Business logic flaws
  - Input validation weaknesses

Dynamic Analysis Tools

- Emulator / Simulator
- MobSF
- Drozer
- Frida
- Proxy tools (Burp Suite, Zed Attack Proxy, etc.)
- Network tools (Wireshark, tcpdump, etc.)

Do We Need Both?

- Static Analysis:
  - Reading the code may be difficult
  - The app may have additional functions when running
- Dynamic Analysis:
  - Allows to see the app's behaviour

One More Thing

- Most “enterprise-grade” mobile application tests will require some level of reverse engineering
  - Reverse engineering: Understand the app, retrieve information from the compiled app
  - Tampering: Modify the app to change its behavior
    - Binary patching
    - Code injection

Introduction to Threat Modeling

What is Threat Modeling?

- Threat modeling is a structured process to identify, analyze, and mitigate security risks in a system.
  - Identify potential threats and vulnerabilities early in the development lifecycle.
  - Design countermeasures to reduce risk.

The Importance of Threat Modeling

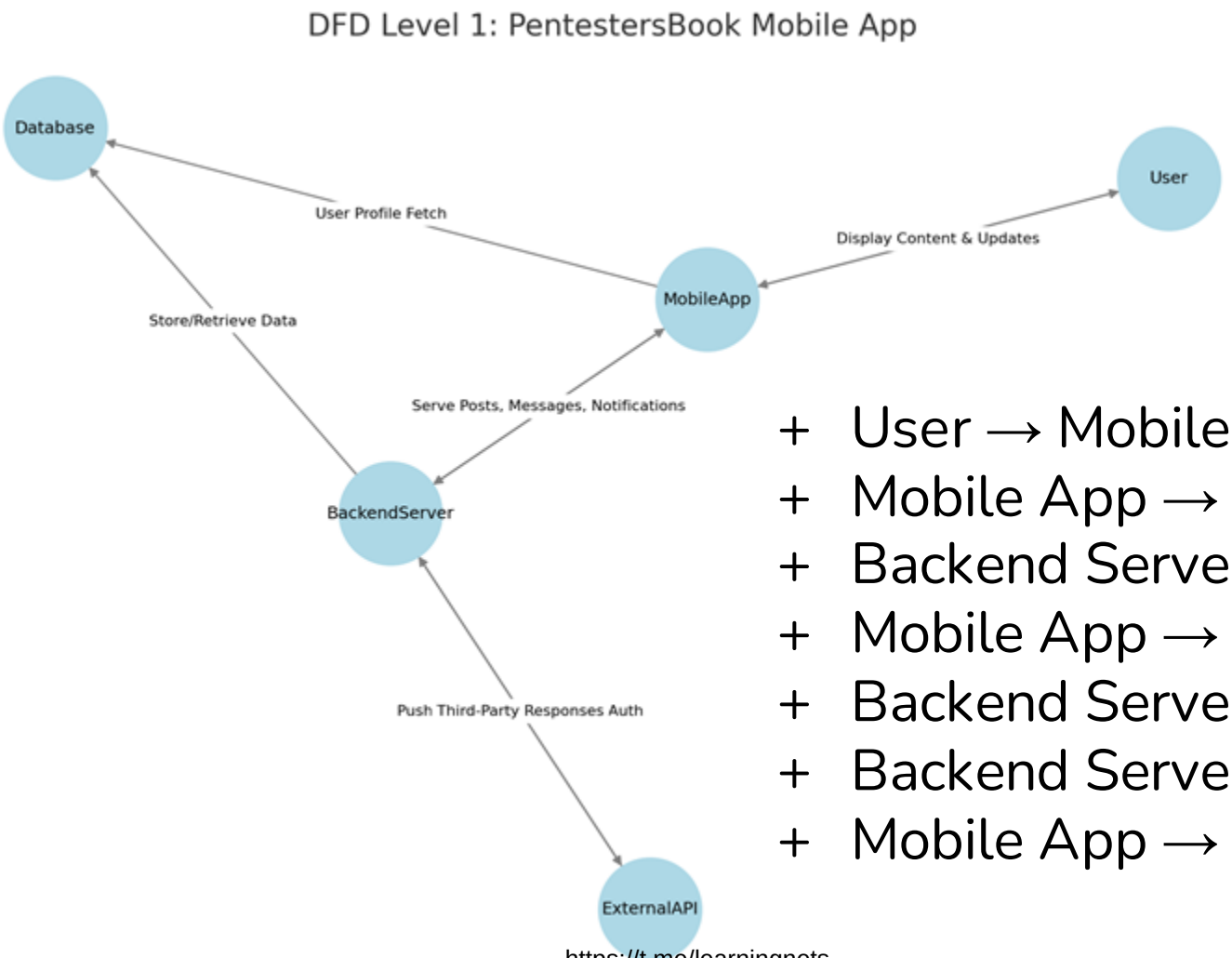
+ Helps prioritize security efforts.
+ Proactively reduces the attack surface.
+ Enhances the overall security posture.

Key Questions in Threat Modeling

- What are we building? (*Understand the system architecture*)
- What can go wrong? (*Identify threats and vulnerabilities*)
- What are we doing to protect it? (*Mitigation strategies*)
- Did we do a good job? (*Validation and testing*)

The Threat Modeling Process

- Understand the System:
  - Create a Data Flow Diagram (DFD).
  - Map out components, data flows, and trust boundaries.
- Identify Threats:
  - Use frameworks like STRIDE (Spoofing, Tampering, etc.).
  - Consider potential attack vectors.



+ User → Mobile App
+ Mobile App → Backend Server
+ Backend Server → Database
+ Mobile App → Database
+ Backend Server → External APIs
+ Backend Server → Mobile App
+ Mobile App → User

Level 1 Data Flow Diagram (DFD) for PentestersBook

- **User → Mobile App**: The user submits login credentials, posts content, and messages.
- **Mobile App → Backend Server**: The app sends encrypted data (authentication, posts, messages).
- **Backend Server → Database**: Stores and retrieves user profiles, posts, and messages.
- **Mobile App → Database**: Direct profile data fetching for quick access.
- **Backend Server → External APIs**: Communicates with third-party services for notifications, authentication, and ML-based content moderation.
- **Backend Server → Mobile App**: Sends updates like new messages, posts, and notifications.
- **Mobile App → User**: Displays content, notifications, and messages.

Trust Boundaries

- A trust boundary is a conceptual line within a system where the level of trust changes. This is where:
  - Different security policies apply.
  - Data integrity and confidentiality risks emerge.
  - Untrusted or semi-trusted data is processed.

Trust Boundaries in Everyday Systems

- A web browser vs. the internet (e.g., Cross-Site Scripting attacks)
- A bank ATM vs. the bank's internal network
- A mobile app vs. backend API server
- A user-controlled mobile device vs. corporate-managed devices

Where Do Trust Boundaries Exist in Mobile Applications?

- **Between the User and the Mobile App**
  - Risk: Users can manipulate inputs, reverse engineer the app, or install modified versions.
  - Mitigation: Input validation, obfuscation, anti-reverse engineering techniques.

Where Do Trust Boundaries Exist in Mobile Applications?

- **Between the Mobile App and the Backend API**
  - Risk: Man-in-the-Middle (MITM) attacks, token theft, API abuse.
  - Mitigation: TLS enforcement, authentication tokens, rate limiting.

Where Do Trust Boundaries Exist in Mobile Applications?

- **Between the Mobile App and Local Device Storage**
  - Risk: Malware, rooted/jailbroken devices accessing sensitive files.
  - Mitigation: Encrypted storage, secure enclave usage (Android Keystore/iOS Keychain).

Where Do Trust Boundaries Exist in Mobile Applications?

- **Between the Backend API and Third-Party Services**
  - Risk: API keys being exposed, weak authentication mechanisms.
  - Mitigation: Token expiration, IP whitelisting, zero-trust principles.

Where Do Trust Boundaries Exist in Mobile Applications?

- **Between Different User Roles (Standard User vs. Admin)**
  - Risk: Privilege escalation attacks, unauthorized data access.
  - Mitigation: Role-based access control (RBAC), least privilege enforcement.

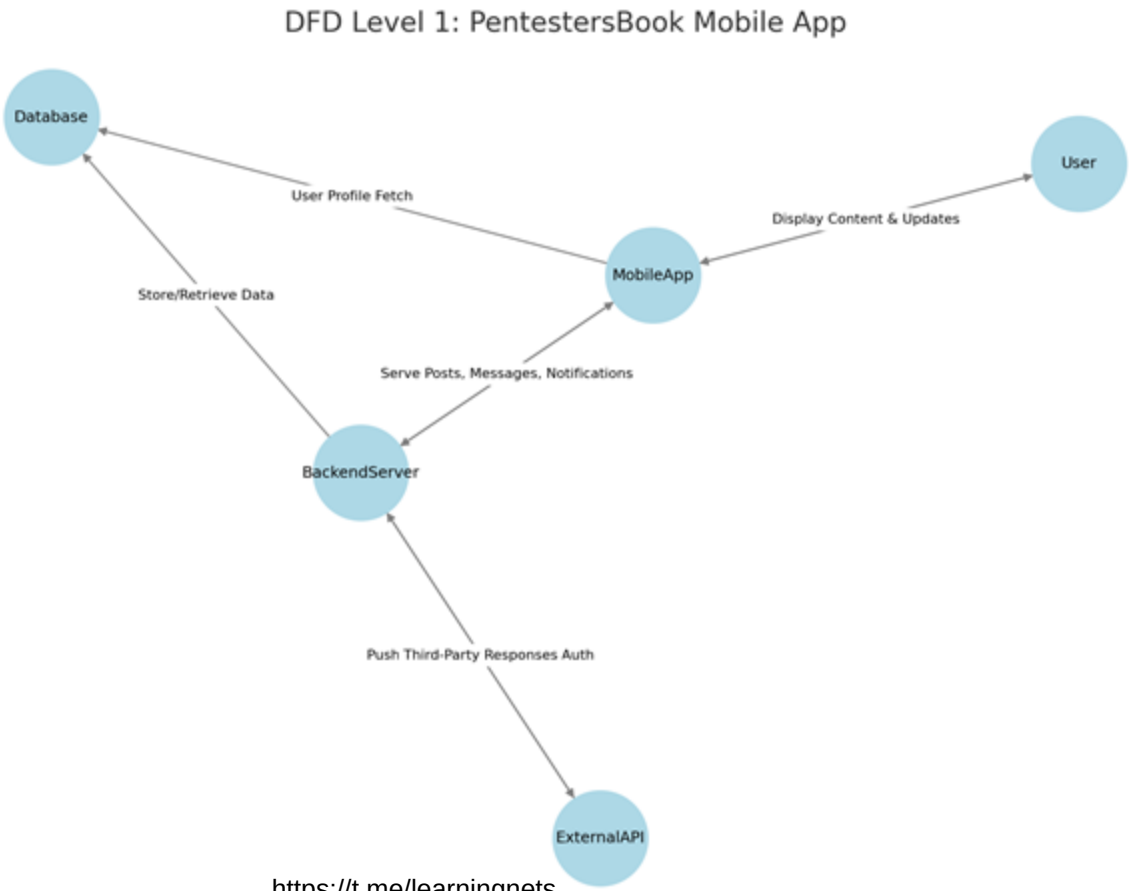
Case Study 1: Uber API Token Exposure (2016)

- Issue: Uber's hardcoded API keys were found inside the mobile app.
- Trust Boundary Violated: Mobile App ↔ Backend API.
- Attack Vector: Attackers extracted the API keys from the mobile app and gained unauthorized access.
- Mitigation: Secure API key handling, restricting API keys to server-side environments.

Case Study 2: WhatsApp GIF Vulnerability (2019)

- Issue: A crafted GIF file could execute malicious code via an Android memory corruption bug.
- Trust Boundary Violated: User ↔ Mobile App.
- Attack Vector: Sending malicious GIFs to exploit WhatsApp.
- Mitigation: Secure memory management, strict input sanitization.

Trust Boundaries



The Threat Modeling Process

- Analyze Risks:
  - Assess likelihood and impact.
  - Prioritize based on risk levels.
- Implement Countermeasures:
  - Mitigate threats with security controls.
  - Validate effectiveness through testing.

Understand the System

- Components: Mobile app, backend server, APIs, third-party SDKs, etc.
- Data flow: How user data travels through the app.
- Entry points: Login pages, APIs, file uploads, etc.

Identifying Assets

- Identifying key assets: User data, API keys, payment info, authentication tokens.
- Understanding data flow: How sensitive data moves in a mobile app.
- Common threat actors: Malicious insiders, hackers, compromised devices.
- Uber’s API keys were found hardcoded in the app, leading to a massive data leak.

Analyzing Mobile Attack Surfaces

- Entry points and attack vectors:
  - User inputs (forms, authentication).
  - API endpoints and data transmission.
  - Device storage, logs, and shared preferences.
  - External libraries and third-party integrations.

Common Mobile Security Risks

- Broken Authentication (session hijacking, insecure token storage).
- Insecure Data Storage (sensitive info in plaintext).
- Insecure Communication (MITM attacks, improper TLS implementation).
- Reverse Engineering & Code Tampering (static & dynamic analysis).
- Supply Chain Attacks (malicious SDKs).

Identify Threats

- Spoofing (*e.g., bypassing login*).
- Tampering (*e.g., altering stored app data*).
- Repudiation (*e.g., lack of audit logs*).
- Information disclosure (*e.g., sensitive data in logs*).
- Denial of service (*e.g., app crashes on malformed input*).
- Elevation of privilege (*e.g., executing admin functions as a regular user*).

Applying STRIDE to a Mobile Application

- Spoofing: Fake login pages, OAuth token abuse.
- Tampering: Modifying APKs/IPA files, manipulating app data.
- Repudiation: No proper logging/auditing.
- Information Disclosure: API misconfigurations, exposing logs.
- Denial of Service: Resource exhaustion, excessive API calls.
- Elevation of Privilege: Exploiting misconfigured permissions.

Analyze Risks

- Assess threat impact and likelihood:
- Use frameworks like DREAD:
  - Damage potential.
  - Reproducibility.
  - Exploitability.
  - Affected users.
  - Discoverability.
- Prioritize threats: High, Medium, Low.

Implement Countermeasures

+ Spoofing: Enforce strong authentication (*e.g.*, *MFA*).
+ Tampering: Secure sensitive data (*e.g.*, *encryption*, *secure storage*).
+ Information Disclosure: Use secure protocols (*e.g.*, *TLS*) and remove sensitive logs.
+ Denial of Service: Implement rate limiting and input validation.

Overview of Mobile Application Vulnerabilities

Categories of Mobile App Vulnerabilities

- Storage
- Crypto
- Auth
- Network
- Platform
- Code
- Resilience
- Privacy

OWASP mobile Top 10 (1/3)

- M1: Improper Credential Usage
- M2: Inadequate Supply Chain Security
- M3: Insecure Authentication/Authorization
- M4: Insufficient Input/Output Validation
- M5: Insecure Communication

OWASP mobile Top 10 (2/3)

- M6: Inadequate Privacy Controls
- M7: Insufficient Binary Protections
- M8: Security Misconfiguration
- M9: Insecure Data Storage
- M10: Insufficient Cryptography

Which Vulnerability is More Important?

- Vulnerabilities are scored
  - Threat agent
  - Ease of exploitability
  - Prevalence
  - Detectability
  - Technical impact
  - Business impact

M1: Improper Credential Usage (Causes)

+ Hardcoded Credentials
+ Insecure Credential Transmission
+ Insecure Credential Storage
+ Weak User Authentication

M1: Improper Credential Usage (Impact)

+ Reputation Damage
+ Information Theft
+ Fraud
+ Unauthorized Access to Data

M1: Improper Credential Usage (Attack Scenario 1)

- Hardcoded API Keys in Mobile Application
  - A banking app stores API keys and credentials directly in its source code for server authentication. An attacker decompiles the app (using tools like JADX or apktool), extracts the hardcoded API keys, and uses them to impersonate legitimate users or interact with backend APIs. This can allow unauthorized transactions or data exfiltration.

M1: Improper Credential Usage (Attack Scenario 1)

- Attack Steps:
  - Download the mobile app APK file.
  - Decompile the app and locate hardcoded API keys in the code.
  - Use the extracted credentials with tools like Postman to call backend APIs without authentication.

M1: Improper Credential Usage (Attack Scenario 2)

- Weak Password Policy Enforcement
  - A fitness app allows users to set weak passwords, such as "12345" or "password". An attacker creates a script for a credential stuffing or brute-force attack using a list of common passwords. Because the app doesn't enforce account lockout or rate-limiting, the attacker gains unauthorized access to multiple accounts.

M1: Improper Credential Usage (Attack Scenario 2)

- Attack Steps:
  - Collect leaked or commonly used passwords from previous breaches.
  - Automate login attempts against the app using tools like Hydra or Burp Suite.
  - Gain access to users' accounts and personal data.

M1: Improper Credential Usage (Attack Scenario 3)

- Unencrypted Credentials in Local Storage
  - A social media app stores user credentials (username and password) in plaintext in local storage for "remember me" functionality. An attacker gains physical or remote access to the device, retrieves the stored credentials, and logs into the user’s account.

M1: Improper Credential Usage (Attack Scenario 3)

- Attack Steps:
  - Gain access to the user's mobile device (e.g., via phishing or malware).
  - Browse the app's local storage or shared preferences folder.
  - Extract the plaintext credentials and use them to authenticate to the app's backend.

M1: Improper Credential Usage (Attack Scenario 4)

- Insecure OAuth Implementation
  - An e-commerce app implements OAuth 2.0 but improperly uses the Implicit Grant Flow instead of a secure flow like the Authorization Code Flow. The app leaks the access token via the URL in a GET request. An attacker intercepts this token using a malicious app or a MitM attack and uses it to impersonate the victim.

M1: Improper Credential Usage (Attack Scenario 4)

- Attack Steps:
  - Perform a man-in-the-middle attack using a tool like mitmproxy.
  - Capture the access token in the app's HTTP traffic.
  - Replay the token to gain unauthorized access to protected resources.

M1: Improper Credential Usage (Attack Scenario 5)

- Credential Leakage Through Improper Logging
  - A ride-hailing app logs sensitive information such as usernames and passwords during the login process for debugging purposes. An attacker gains access to the device logs using tools like adb (Android Debug Bridge) or jailbreaks the device to read log files, retrieving user credentials.

M1: Improper Credential Usage (Attack Scenario 5)

- Attack Steps:
  - Access the target device and extract logs using `adb logcat` or similar tools.
  - Identify sensitive credentials or authentication tokens within the logs.
  - Use the credentials to log in to the victim's account.

M1: Improper Credential Usage (Mitigation)

- Avoid Using Hardcoded Credentials
- Properly Handle User Credentials
  - Encrypt credentials during transmission
  - Do not store user credentials on the device
  - Implement strong user authentication protocols
  - Regularly update and rotate any used API keys or tokens

M2: Inadequate Supply Chain Security (Causes)

+ Lack of Security in Third-Party Components
+ Malicious Insider Threats
+ Inadequate Testing and Validation
+ Lack of Security Awareness

M2: Inadequate Supply Chain Security (Impact)

+ Data Breach
+ Malware Infection
+ Unauthorized Access
+ System Compromise

M2: Inadequate Supply Chain Security (Attack Scenario 1)

- Malicious Third-Party SDK
  - A fitness tracking app integrates a third-party analytics SDK to track user behavior. The SDK, however, includes malicious code that collects sensitive user data (e.g., location, health metrics, or login credentials) and transmits it to an attacker-controlled server. The app developers did not properly review the SDK before integration.

M2: Inadequate Supply Chain Security (Attack Scenario 1)

- Attack Steps:
  - The attacker injects malicious code into the third-party SDK.
  - The app integrates the SDK without auditing its security.
  - Sensitive user data is exfiltrated when the app runs, compromising privacy and security.

M2: Inadequate Supply Chain Security (Attack Scenario 2)

- Compromised Build Server
  - A gaming app's build server is improperly secured, allowing attackers to access and inject malicious code into the app during the build process. The compromised app is then distributed through legitimate app stores, infecting end-user devices.

M2: Inadequate Supply Chain Security (Attack Scenario 2)

- Attack Steps:
  - Gain access to the app's build server via weak credentials or vulnerabilities.
  - Inject malicious code or a backdoor into the app during compilation.
  - Distribute the compromised app through app stores to infect user devices.

M2: Inadequate Supply Chain Security (Attack Scenario 3)

- Tampered Open-Source Libraries
  - A banking app uses an open-source library for cryptographic functions. The attacker compromises the library's repository, introducing vulnerabilities that allow them to decrypt sensitive user data. The app developers unknowingly use the tampered library in their app.

M2: Inadequate Supply Chain Security (Attack Scenario 3)

- Attack Steps:
  - The attacker compromises the library's GitHub repository or package manager (e.g., npm, Maven, PyPI).
  - The app integrates the malicious version of the library.
  - Encrypted user data is intercepted and decrypted by the attacker.

M2: Inadequate Supply Chain Security (Attack Scenario 4)

- Dependency Confusion Attack
  - An attacker registers a malicious package with the same name as a private internal library used by an e-commerce app. The package is uploaded to a public package repository (e.g., npm or PyPI), and the app's build system mistakenly pulls the malicious package instead of the private one.

M2: Inadequate Supply Chain Security (Attack Scenario 4)

- Attack Steps:
  - The attacker creates a malicious package with the same name as a private dependency.
  - The app's build system downloads the malicious package from the public repository.
  - The malicious code executes on user devices, compromising sensitive data or enabling remote control.

M2: Inadequate Supply Chain Security (Attack Scenario 5)

- Unverified App Updates
  - A productivity app allows users to update the app directly without verifying the integrity of the update package. An attacker performs a man-in-the-middle (MitM) attack and injects malicious updates, replacing the legitimate app with a trojanized version.

M2: Inadequate Supply Chain Security (Attack Scenario 5)

- Attack Steps:
  - The attacker intercepts the update request using a MitM attack (e.g., ARP spoofing).
  - The malicious update package is delivered to the user's device.
  - The compromised app runs on the device, allowing the attacker to steal sensitive data or execute further attacks.

M2: Inadequate Supply Chain Security (Mitigation)

+ Implement secure coding practices, code review, and testing
+ Ensure secure app signing and distribution processes
+ Use only trusted and validated third-party libraries or components
+ Establish security controls for app updates, patches, and releases
+ Monitor and detect supply chain security incidents

M3: Insecure Authentication/Authorization (Causes)

- Insecure Direct Object Reference (IDOR) vulnerabilities
- Hidden Endpoints
- User Role or Permission Transmissions
- Anonymous Backend API Execution
- Local Storage of Passwords or Shared Secrets
- Weak Password Policy

M3: Insecure Authentication/Authorization (Impact)

+ Reputation Damage
+ Information Theft
+ Fraud
+ Unauthorized Access to Data

M3: Insecure Authentication/Authorization (Attack Scenario 1)

- Bypassing Authentication via Debug Code
  - A retail app includes hidden debug functionality that allows bypassing login screens for testing purposes. The functionality is not removed in the production build. An attacker decompiles the app, identifies the debug mode, and uses it to gain access to any account without authentication.

M3: Insecure Authentication/Authorization (Attack Scenario 1)

- Attack Steps:
  - Decompile the app using tools like JADX or apktool.
  - Discover hidden debug methods or flags in the code (e.g., a debug=true parameter).
  - Use the debug feature to bypass authentication and gain full access to app features.

M3: Insecure Authentication/Authorization (Attack Scenario 2)

- Token Replay Attack
  - An online banking app uses insecure session tokens that don't expire or lack proper validation mechanisms. An attacker intercepts a session token via a man-in-the-middle (MitM) attack or by extracting it from local storage, then reuses the token to impersonate the victim.

M3: Insecure Authentication/Authorization (Attack Scenario 2)

- Attack Steps:
  - Capture the session token via tools like Burp Suite or a malware-infected device.
  - Replay the token in API requests to gain unauthorized access to the victim's account.
  - Perform actions such as transferring funds or viewing sensitive data.

M3: Insecure Authentication/Authorization (Attack Scenario 3)

- Weak Biometric Authentication
  - A healthcare app uses facial recognition for login but does not implement liveness detection. An attacker takes a high-quality photo of the victim from their social media account and uses it to bypass facial authentication.

M3: Insecure Authentication/Authorization (Attack Scenario 3)

- Attack Steps:
  - Obtain a clear photo of the victim.
  - Use the photo to bypass the facial recognition mechanism.
  - Access sensitive health records and personal data.

M3: Insecure Authentication/Authorization (Attack Scenario 4)

- Lack of Role-Based Access Control (RBAC)
  - A corporate communication app fails to enforce proper authorization controls. An attacker, logged in as a regular user, modifies API requests to escalate privileges and access administrative features such as deleting user accounts or viewing sensitive company information.

M3: Insecure Authentication/Authorization (Attack Scenario 4)

- Attack Steps:
  - Login as a regular user and monitor API traffic using a proxy tool like Burp Suite.
  - Modify the user_role parameter in API requests (e.g., change user to admin).
  - Exploit the lack of server-side authorization checks to perform administrative actions.

M3: Insecure Authentication/Authorization (Attack Scenario 5)

- Credential Leak in Debug Logs
  - A ride-sharing app improperly logs sensitive authentication details, such as usernames and passwords, in plaintext during the login process for debugging purposes. An attacker with access to the device retrieves the logs and uses the leaked credentials to log into the victim's account.

M3: Insecure Authentication/Authorization (Attack Scenario 5)

- Attack Steps:
  - Access the device and extract logs using adb logcat or other tools.
  - Locate authentication logs containing usernames and passwords.
  - Use the credentials to log into the victim's account and exploit their ride-sharing credits or payment details.

M3: Insecure Authentication/Authorization (Mitigation)

+ Avoid weak authentication patterns
+ Reinforce authentication

M4: Insufficient Input/Output Validation (Causes)

- Lack of Input Validation
- Inadequate Output Sanitization
- Context-Specific Validation Neglect
- Insufficient Data Integrity Checks
- Poor Secure Coding Practices

M4: Insufficient Input/Output Validation (Impact)

- Code Execution
- Data Breaches
- System Disruptions
- Data Integrity Issues

M4: Insufficient Input/Output Validation (Attack Scenario 1)

- SQL Injection via Search Functionality
  - An e-commerce app has a search feature that directly passes user input to a backend SQL database without proper sanitization. An attacker crafts malicious input to execute unauthorized SQL commands, exposing or modifying sensitive data.

M4: Insufficient Input/Output Validation (Attack Scenario 1)

- Attack Steps:
  - Enter malicious input, such as '; DROP TABLE users; --, into the app's search bar.
  - The backend executes the input as part of an SQL query without sanitization.
  - Sensitive user data is exposed, or the database is manipulated.

M4: Insufficient Input/Output Validation (Attack Scenario 2)

- Cross-Site Scripting (XSS) in Chat Application
  - A social networking app allows users to send messages to each other but fails to sanitize or validate the input. An attacker sends a malicious script in a chat message, which is executed when the recipient views the message.

M4: Insufficient Input/Output Validation (Attack Scenario 2)

- Attack Steps:
  - Send a message containing a malicious payload, e.g., <script>alert('Hacked');</script>.
  - The app fails to sanitize the message and renders the malicious script.
  - The script executes on the recipient's device, stealing session tokens or redirecting them to phishing sites.

M4: Insufficient Input/Output Validation (Attack Scenario 3)

- Remote Code Execution via File Upload
  - A document management app allows users to upload files but doesn't validate file types or content. An attacker uploads a malicious .exe or .php file disguised as a document, which the backend server executes, leading to a remote code execution vulnerability.

M4: Insufficient Input/Output Validation (Attack Scenario 3)

- Attack Steps:
  - Upload a malicious file named invoice.pdf.php.
  - The backend fails to validate the file type and allows execution.
  - The attacker gains control of the server or access to sensitive data.

M4: Insufficient Input/Output Validation (Attack Scenario 4)

- Buffer Overflow via User Input
  - A fitness app processes user-provided biometric data but doesn't properly validate input lengths. An attacker sends an unusually long input string, causing a buffer overflow and allowing arbitrary code execution on the mobile device.

M4: Insufficient Input/Output Validation (Attack Scenario 4)

- Attack Steps:
  - Provide oversized input, such as a string with thousands of characters, to the biometric input field.
  - The app fails to handle the input safely, causing a buffer overflow.
  - Execute arbitrary code or crash the app to exploit further vulnerabilities.

M4: Insufficient Input/Output Validation (Attack Scenario 5)

- Directory Traversal in File Download
  - A document-sharing app allows users to download files by specifying a file path in the URL, but it fails to validate the input. An attacker manipulates the input to access unauthorized files on the server.

M4: Insufficient Input/Output Validation (Attack Scenario 5)

- Attack Steps:
  - Access the file download URL, e.g., https://example.com/download?file=../../etc/passwd.
  - The app doesn't validate or sanitize the file path.
  - Unauthorized files, such as configuration files or sensitive data, are exposed.

M4: Insufficient Input/Output Validation (Mitigation)

- Input Validation
- Output Sanitization
- Context-Specific Validation
- Data Integrity Checks
- Secure Coding Practices
- Regular Security Testing

M5: Insecure Communication (Causes)

- All communications should be encrypted
  - WiFi, TCP/IP, NFC, GSM, etc.
- Lack of certificate inspection
- Weak handshake negotiation
- Privacy information leakage
- Credential information leakage
- Two-Factor authentication bypass

M5: Insecure Communication (Impact)

+ Identity theft
+ Fraud
+ Reputational Damage

M5: Insecure Communication (Attack Scenario 1)

- Man-in-the-Middle (MitM) Attack on Unencrypted Traffic
  - A weather app transmits sensitive user data, such as location and preferences, over HTTP instead of HTTPS. An attacker intercepts the unencrypted traffic on a public Wi-Fi network and captures sensitive information.

M5: Insecure Communication (Attack Scenario 1)

- Attack Steps:
  - The attacker sets up a rogue access point or uses ARP spoofing on a public Wi-Fi network.
  - The user connects to the app, and unencrypted data is sent via HTTP.
  - The attacker intercepts and reads sensitive information, such as location data or user preferences.

M5: Insecure Communication (Attack Scenario 2)

- Exploiting Weak TLS Configuration
  - A banking app uses HTTPS but supports outdated TLS versions (e.g., TLS 1.0 or 1.1) and weak cipher suites. An attacker exploits these weaknesses to decrypt the communication and access sensitive information, such as session tokens or login credentials.

M5: Insecure Communication (Attack Scenario 2)

- Attack Steps:
  - Perform a downgrade attack to force the app to use an older, vulnerable TLS version.
  - Intercept and decrypt the communication using tools like SSLStrip or mitmproxy.
  - Extract sensitive data from the decrypted traffic.

M5: Insecure Communication (Attack Scenario 3)

- Credential Theft via Certificate Pinning Bypass
  - A social media app attempts to secure communication using SSL certificate pinning but implements it incorrectly. An attacker bypasses pinning by modifying the app or injecting their own certificate, intercepting user credentials during the login process.

M5: Insecure Communication (Attack Scenario 3)

- Attack Steps:
  - Use tools like Frida or Xposed to disable or bypass certificate pinning in the app.
  - Perform a man-in-the-middle attack with a custom certificate to intercept HTTPS traffic.
  - Capture usernames and passwords submitted during login.

M5: Insecure Communication (Attack Scenario 4)

- Sensitive Data Leakage via Push Notifications
  - A healthcare app sends sensitive medical information (e.g., test results) via unencrypted push notifications. An attacker monitors network traffic or gains access to the device to view the sensitive data in the notifications.

M5: Insecure Communication (Attack Scenario 4)

- Attack Steps:
  - The attacker monitors unencrypted notification traffic using a network sniffer.
  - Sensitive information, such as test results or appointments, is exposed in plaintext.
  - The attacker uses this information for phishing or blackmail.

M5: Insecure Communication (Attack Scenario 5)

- Exploiting Lack of End-to-End Encryption
  - A messaging app claims to offer secure communication but doesn't implement proper end-to-end encryption. An attacker intercepts messages in transit between the sender and receiver by compromising the app's backend server or network communication.

M5: Insecure Communication (Attack Scenario 5)

- Attack Steps:
  - The attacker intercepts messages during transit using a MitM attack or server compromise.
  - Since the messages are only encrypted at the transport level and decrypted at the server, the attacker gains access to their plaintext content.
  - Sensitive user communications are exposed, leading to privacy breaches.

M5: Insecure Communication (Mitigation)

- Apply SSL/TLS to transport channels
- Use strong, industry standard cipher suites
- Use certificates signed by a trusted CA provider
- Never allow bad certificates
- Alert users through the UI if the mobile app detects an invalid certificate
- Do not send sensitive data over alternate channels

M6: Inadequate Privacy Controls (Causes)

+ Insecure data storage and communication
+ Data access with insecure authentication and authorization
+ Insider attacks on the app's sandbox

M6: Inadequate Privacy Controls (Impact)

+ Violation of legal regulations
+ Financial damage due to victims' lawsuits
+ Reputational damage

M6: Inadequate Privacy Controls (Attack Scenario 1)

- Unauthorized Access to Location Data
  - A navigation app collects and stores users' location data without proper user consent or encryption. An attacker compromises the app’s backend or intercepts the API calls and gains access to sensitive location history, which can be used to track the victim's movements.

M6: Inadequate Privacy Controls (Attack Scenario 1)

- Attack Steps:
  - The attacker intercepts API calls using tools like Burp Suite.
  - Location data is transmitted unencrypted or poorly secured.
  - The attacker collects sensitive location information and uses it to monitor the victim's behavior or physical movements.

M6: Inadequate Privacy Controls (Attack Scenario 2)

- Unnecessary Access to Contacts
  - A social media app requests access to users' contact lists but does not disclose how the data will be used. The app uploads all contacts to its servers without anonymization. An attacker breaches the server and accesses the entire user contact database, which can be used for spam campaigns or phishing attacks.

M6: Inadequate Privacy Controls (Attack Scenario 2)

- Attack Steps:
  - The app requests contact list permissions without clear justification.
  - All contact data is uploaded to the backend server.
  - The attacker breaches the server or intercepts the data to steal sensitive personal information.

M6: Inadequate Privacy Controls (Attack Scenario 3)

- Logging Sensitive Information
  - A fitness app logs user credentials, biometric data (e.g., heart rate, calorie intake), and session tokens in plaintext to local storage for debugging purposes. An attacker gains access to the mobile device or extracts logs remotely, exposing sensitive health information.

M6: Inadequate Privacy Controls (Attack Scenario 3)

- Attack Steps:
  - Access the mobile device or use malware to extract log files.
  - Locate sensitive information, such as biometric data or credentials, in plaintext logs.
  - Use the information to compromise user accounts or invade their privacy.

M6: Inadequate Privacy Controls (Attack Scenario 4)

- Excessive Data Collection
  - A shopping app collects data unrelated to its functionality, such as browsing habits, microphone input, or keystroke patterns, without informing the user. This data is sold to third-party advertisers without anonymization. An attacker buys or steals this data to profile and target users with malicious ads or scams.

M6: Inadequate Privacy Controls (Attack Scenario 4)

- Attack Steps:
  - The app collects data unrelated to shopping activities without user consent.
  - Data is stored in an insecure database or shared with third parties.
  - An attacker steals or buys the data and uses it for targeted social engineering attacks.

M6: Inadequate Privacy Controls (Attack Scenario 5)

- Exposing Sensitive Files via Misconfigured File Storage
  - A document management app stores sensitive files (e.g., ID cards, bank statements) in an insecure cloud bucket with publicly accessible permissions. An attacker finds the misconfigured storage using tools like Shodan or Google Dorks and downloads sensitive user files.

M6: Inadequate Privacy Controls (Attack Scenario 5)

- Attack Steps:
  - The attacker scans for misconfigured cloud storage using automated tools.
  - Locate publicly accessible files uploaded by the app.
  - Download sensitive files and use them for identity theft or fraud.

M6: Inadequate Privacy Controls (Mitigation)

- Is all PII processed really necessary?
- Can some of the PII be replaced by less critical information?
- Can some of the PII be reduced?
- Can some of the PII be anonymized?
- Can some of the PII be deleted after a predefined period?
- Can users consent to optional PII usage?

M7: Insufficient Binary Protection (Causes)

+ No threat modeling analysis
+ Not using obfuscation tools

M7: Insufficient Binary Protection (Impact)

+ Loss of API keys
+ Disabled payment and/or licence
+ Loss of IP

M7: Insufficient Binary Protection (Attack Scenario 1)

- Reverse Engineering to Extract API Keys
  - A payment app embeds sensitive API keys and secrets in the binary without obfuscation. An attacker decompiles the app using tools like JADX or apktool, extracts the keys, and uses them to make unauthorized API calls.

M7: Insufficient Binary Protection (Attack Scenario 1)

- Attack Steps:
  - Download the app's APK or IPA file.
  - Decompile the app to view its source code or resources.
  - Extract hardcoded API keys and secrets and use them to access backend services or perform unauthorized transactions.

M7: Insufficient Binary Protection (Attack Scenario 2)

- Dynamic Analysis to Bypass Authentication
  - An attacker uses a dynamic analysis tool like Frida or Xposed to hook into a mobile app's runtime and bypass authentication checks by modifying the app's logic in memory.

M7: Insufficient Binary Protection (Attack Scenario 2)

- Attack Steps:
  - Attach a dynamic analysis tool to the app during runtime.
  - Hook authentication methods to force a return value indicating successful login.
  - Gain unauthorized access to the app without valid credentials.

M7: Insufficient Binary Protection (Attack Scenario 3)

- Malware Injection via Unprotected Code
  - A gaming app doesn't implement integrity checks or binary signing. An attacker modifies the app's binary to include malicious code (e.g., spyware) and distributes the modified app through unofficial app stores or phishing campaigns.

M7: Insufficient Binary Protection (Attack Scenario 3)

- Attack Steps:
  - Decompile and modify the app's binary to inject malicious code.
  - Repack the modified app and sign it with a self-signed certificate.
  - Distribute the malicious app to unsuspecting users, gaining control over their devices or stealing sensitive data.

M7: Insufficient Binary Protection (Attack Scenario 4)

- Debugging to Extract User Credentials
  - A health tracking app allows debugging in production builds. An attacker uses a debugger (e.g., IDA Pro or Ghidra) to analyze memory during runtime and extract sensitive information like usernames, passwords, or session tokens.

M7: Insufficient Binary Protection (Attack Scenario 4)

- Attack Steps:
  - Attach a debugger to the app running on a rooted or jailbroken device.
  - Monitor the app's memory during sensitive operations like login.
  - Extract user credentials or tokens and use them for unauthorized access.

M7: Insufficient Binary Protection (Attack Scenario 5)

- Licensing Mechanism Circumvention
  - A premium music streaming app uses a weak licensing mechanism to enforce subscription-based access. An attacker reverse engineers the app to identify and patch the code responsible for license validation, enabling free access to premium features.

M7: Insufficient Binary Protection (Attack Scenario 5)

- Attack Steps:
  - Decompile the app binary and analyze its license validation logic.
  - Modify the binary to bypass or disable license checks.
  - Repack and run the modified app to access premium features without paying.

M7: Insufficient Binary Protection (Mitigation)

+ Prevent reverse engineering
+ Obfuscate
+ Perform integrity checks

M8: Security Misconfiguration (Causes)

- Default settings not reviewed
- Lack of secure communication
- Weak or absent access controls
- Failure to update or patch
- Improper storage of sensitive data
- Insecure file provider path settings
- Exported activities

M8: Security Misconfiguration (Impact)

- Unauthorized access to sensitive data
- Account hijacking or impersonation
- Data breaches
- Compromise of backend systems

M8: Security Misconfiguration (Attack Scenario 1)

- Exposed Debugging Features in Production
  - A mobile banking app is released with debugging features enabled in the production build. These features allow attackers to see detailed logs, bypass certain security checks, and interact with hidden testing endpoints.

M8: Security Misconfiguration (Attack Scenario 1)

- Attack Steps:
  - The attacker uses tools like Logcat (for Android) or Console.app (for iOS) to view sensitive debug logs.
  - The logs reveal API endpoints, hardcoded keys, or internal application logic.
  - The attacker exploits this information to perform unauthorized actions or access sensitive data.

M8: Security Misconfiguration (Attack Scenario 2)

- Misconfigured Permissions
  - A mobile game app requests excessive permissions, such as access to the camera, microphone, and contacts, without any legitimate need. An attacker exploits this by installing malware that leverages these permissions to eavesdrop on conversations or steal contact information.

M8: Security Misconfiguration (Attack Scenario 2)

- Attack Steps:
  - Install the app on a compromised device.
  - Use malicious code to access the over-permissioned features.
  - Record audio, capture images, or exfiltrate contact data without the user's knowledge.

M8: Security Misconfiguration (Attack Scenario 3)

- Unrestricted Access to Sensitive Files
  - A file-sharing app stores user-uploaded files in a public cloud bucket without proper access controls. An attacker scans for misconfigured storage using tools like Shodan or AWS S3 Scanner and downloads private files.

M8: Security Misconfiguration (Attack Scenario 3)

- Attack Steps:
  - Use automated tools to identify publicly accessible cloud storage buckets.
  - Access files such as personal photos, documents, or credentials stored in the bucket.
  - Exploit the stolen files for identity theft or blackmail.

M8: Security Misconfiguration (Attack Scenario 4)

- Weak Default Configuration
  - A smart home app uses default admin credentials ("admin/admin") for connecting to IoT devices. An attacker connects to the devices using these default credentials to take control, manipulate settings, or spy on users.

M8: Security Misconfiguration (Attack Scenario 4)

- Attack Steps:
  - Scan the network to identify IoT devices configured with default credentials.
  - Log in using the default username and password.
  - Take control of the smart devices, such as cameras, alarms, or thermostats.

M8: Security Misconfiguration (Attack Scenario 5)

- Improper API Endpoint Exposure
  - A fitness app's backend exposes internal API endpoints that are not intended for public use. These endpoints allow attackers to retrieve sensitive data or perform unauthorized actions, such as modifying user profiles.

M8: Security Misconfiguration (Attack Scenario 5)

- Attack Steps:
  - Intercept app traffic using a proxy tool like Burp Suite or mitmproxy.
  - Identify hidden or undocumented API endpoints by analyzing API calls.
  - Exploit these endpoints to retrieve sensitive information, such as other users' workout data, or modify account details.

M8: Security Misconfiguration (Mitigation)

+ Secure default configurations
+ Default credentials
+ Insecure permissions
+ Least privilege
+ Secure network configuration
+ Disable Debugging
+ Disable backup mode (Android)
+ Limit application attack surface

M9: Insecure Data Storage (Causes)

- Lack of Access Controls
- Inadequate Encryption
- Unintentional Data Exposure
- Poor Session Management
- Insufficient Input Validation
- Cloud Storage Misconfigurations
- Third-Party Library Vulnerabilities
- Unintended Data Sharing

M9: Insecure Data Storage (Impact)

- Data breaches
- Compromised user accounts
- Data tampering and integrity issues
- Unauthorised access to application resources
- Reputation and trust damage
- Compliance violations

M9: Insecure Data Storage (Attack Scenario 1)

- Storing Sensitive Data in Plaintext
  - A health tracking app stores sensitive user information, such as passwords, medical history, and location data, in plaintext within the device's local storage. An attacker gains physical or remote access to the device and extracts this data.

M9: Insecure Data Storage (Attack Scenario 1)

- Attack Steps:
  - Gain access to the user's mobile device through theft, malware, or remote control tools.
  - Navigate to the app's local storage directory or shared preferences.
  - Extract sensitive information stored in plaintext, such as passwords or health records.

M9: Insecure Data Storage (Attack Scenario 2)

- Database Files Accessible Without Encryption
  - A financial app stores transaction history and account details in a SQLite database without encryption. An attacker extracts the .db file from a rooted or jailbroken device and retrieves sensitive information.

M9: Insecure Data Storage (Attack Scenario 2)

- Attack Steps:
  - Root or jailbreak the target device.
  - Locate the app’s SQLite database file in the app's data directory.
  - Open the database using tools like DB Browser for SQLite to view transaction data and account details.

M9: Insecure Data Storage (Attack Scenario 3)

- Caching Sensitive Data in Temporary Files
  - An e-commerce app temporarily stores credit card details and purchase information in cache files for performance optimization. These files are not cleared after a session, and an attacker extracts them using file system access.

M9: Insecure Data Storage (Attack Scenario 3)

- Attack Steps:
  - Gain access to the file system through malware or by rooting/jailbreaking the device.
  - Identify cached files containing sensitive data in temporary directories.
  - Read the files to extract credit card details and other private information.

M9: Insecure Data Storage (Attack Scenario 4)

- Insecure Backup Mechanism
  - A messaging app includes an automatic backup feature that saves chat history and attachments in the cloud. However, the backups are not encrypted and are accessible to anyone who gains access to the cloud storage.

M9: Insecure Data Storage (Attack Scenario 4)

- Attack Steps:
  - The attacker gains access to the victim's cloud storage account via credential stuffing or social engineering.
  - Locate unencrypted backup files uploaded by the app.
  - Download and analyze the files to read chat history and attachments.

M9: Insecure Data Storage (Attack Scenario 5)

- Sensitive Data in Logs
  - A mobile banking app logs sensitive information, such as session tokens and account numbers, during API interactions for debugging purposes. An attacker retrieves the logs from the device to steal sensitive data.

M9: Insecure Data Storage (Attack Scenario 5)

- Attack Steps:
  - Use tools like adb logcat (Android) or console access (iOS) to extract logs from the device.
  - Locate sensitive data such as session tokens or account details within the logs.
  - Use the stolen information to impersonate the user or perform unauthorized actions.

M9: Insecure Data Storage (Mitigation)

- Use Strong Encryption
- Secure Data Transmission
- Implement Secure Storage Mechanisms
- Employ Proper Access Controls
- Validate Input and Sanitize Data
- Apply Secure Session Management
- Regularly Update and Patch Dependencies

M10: Insufficient Cryptography (Causes)

- Weak Encryption Algorithms
- Insufficient Key Length
- Improper Key Management
- Flawed Encryption Implementation
- Insecure Storage of Data/Encryption Keys
- Lack of Secure Transport Layer
- Insufficient Validation and Authentication
- Lack of Salting

M10: Insufficient Cryptography (Impact)

- Data Breach
- Loss of Intellectual Property
- Financial Losses
- Compliance and Legal Consequences

M10: Insufficient Cryptography (Attack Scenario 1)

- Using Weak Encryption Algorithms
  - A financial app encrypts sensitive user data, such as credit card details, using a weak algorithm like DES (Data Encryption Standard) or custom-built encryption. An attacker brute-forces or exploits known vulnerabilities in the algorithm to decrypt the data.

M10: Insufficient Cryptography (Attack Scenario 1)

- Attack Steps:
  - Extract the encrypted data (e.g., from local storage or intercepted network traffic).
  - Identify the encryption algorithm (e.g., via static analysis or known patterns).
  - Use brute-force or tools like Hashcat to decrypt the sensitive information.

M10: Insufficient Cryptography (Attack Scenario 2)

- Hardcoded Cryptographic Keys
  - A health app hardcodes cryptographic keys into its source code for encrypting user health records. An attacker decompiles the app, retrieves the hardcoded keys, and uses them to decrypt sensitive user data.

M10: Insufficient Cryptography (Attack Scenario 2)

- Attack Steps:
  - Decompile the app using tools like JADX or apktool.
  - Search for hardcoded keys in the source code or resources.
  - Use the keys to decrypt sensitive data stored on the device or transmitted over the network.

M10: Insufficient Cryptography (Attack Scenario 3)

- Lack of Encryption for Sensitive Data in Transit
  - An e-commerce app transmits sensitive payment details (e.g., credit card numbers) over HTTP instead of HTTPS. An attacker performs a man-in-the-middle (MitM) attack to intercept and read the unencrypted data.

M10: Insufficient Cryptography (Attack Scenario 3)

- Attack Steps:
  - Set up a MitM attack using tools like mitmproxy or Wireshark.
  - Intercept the HTTP traffic containing sensitive payment data.
  - Extract credit card details and use them for fraud.

M10: Insufficient Cryptography (Attack Scenario 4)

- Insecure Key Management
  - A messaging app generates encryption keys for secure communication but stores the keys in local storage without additional protection. An attacker gains access to the device and retrieves the keys, enabling them to decrypt private messages.

M10: Insufficient Cryptography (Attack Scenario 4)

- Attack Steps:
  - Gain physical or remote access to the user's mobile device.
  - Extract the encryption keys stored in local storage.
  - Use the keys to decrypt stored messages or intercept future communications.

M10: Insufficient Cryptography (Attack Scenario 5)

- Failure to Implement Proper Padding
  - A mobile banking app uses AES encryption in ECB (Electronic Codebook) mode with improper padding. An attacker exploits the predictable nature of ECB mode to reconstruct encrypted data, such as transaction details, without the encryption key.

M10: Insufficient Cryptography (Attack Scenario 5)

- Attack Steps:
  - Intercept encrypted traffic or extract encrypted data from storage.
  - Analyze the ciphertext patterns caused by ECB mode's lack of randomness.
  - Reconstruct the original plaintext data, exposing sensitive transaction information.

M10: Insufficient Cryptography (Mitigation)

+ Use Strong Encryption Algorithms
+ Ensure Sufficient Key Length
+ Follow Secure Key Management Practices
+ Implement Encryption Correctly
+ Secure Storage of Encryption Keys
+ Employ Secure Transport Layer
+ Use Strong Hash Functions
+ Use Salting

Recon: Understanding and Mapping the Application

Why Recon Matters

+ Helps identify potential vulnerabilities early.
+ Guides the rest of the penetration testing process.

When Recon Fails

- Missed Endpoint Discovery
- Overlooked Hardcoded Credentials
- Lack of File Storage Analysis
- Missed App Store Metadata Recon

Step 1: Understanding the Application

- Analyze the App's Functionality
- Identify Data Flows
- Gather Documentation

Step 2: Static Analysis Recon

- Extract the App's Binary
- Analyze the Binary
- Review Manifest/Plist Files

Step 3: Dynamic Analysis Recon

+ Intercept Traffic
+ Monitor Runtime Behavior

Step 4: External Recon

+ App Store Recon
+ Third-Party Integrations
+ Search Engine Dorks

Step 5: Mapping the Attack Surface

+ Review Findings
+ Create an Attack Surface Map
+ Document Observations

Static Analysis of Android Apps

Why Static Analysis Matters

- Early detection of hardcoded secrets, weak configurations, and insecure code practices.
- Complements dynamic analysis by revealing hidden flaws.

Common Vulnerabilities to Look for

- HardcodedSecrets - API keys, passwords, and sensitive URLs.
- InsecureLogging - Logging sensitive user data.
- InsecureStorage - Storing credentials in SharedPreferences and external storage.
- WeakCrypto - Hardcoded encryption keys and weak encryption algorithms.

Common Vulnerabilities to Look for

+ UnprotectedComponents - Exported activities, services, and content providers.
+ WebViewVulnerabilities - Untrusted URL loading and JavaScript enabled.
+ InsecureNetwork - Use of HTTP and ignoring SSL certificate validation.
+ ReflectionExecution - Dynamic code execution with reflection.

Introduction to Hardcoded Secrets

- Hardcoded secrets are API keys, credentials, encryption keys, and sensitive URLs stored directly in the source code.
- They expose critical data to attackers, leading to data breaches, service abuse, and unauthorized access.

Introduction to Hardcoded Secrets

- Common examples include:
  - API keys for third-party services (Google Maps, AWS, Firebase, etc.).
  - Hardcoded usernames and passwords.
  - Database connection strings.
  - Private cryptographic keys.

Impact of Hardcoded Secrets

- Easy Target for Attackers:
  - Attackers use automated tools to scan repositories for hardcoded secrets.
  - Even if removed, they may still be present in Git history.
- Leads to Unauthorized Access:
  - Compromised API keys can be used to abuse services or steal user data.
- Regulatory and Compliance Risks:
  - Violates compliance requirements like GDPR, PCI-DSS, and ISO 27001.

How to Find Hardcoded Secrets?

- Manual Code Review:
  - Look for suspicious string literals in codebases.
- Automated Scanning Tools:
  - GitLeaks - Scans Git repositories for secrets.
  - TruffleHog - Searches commit history for sensitive information.
  - GitHub Secret Scanning - Detects exposed credentials in public repositories.

Java Example: Hardcoded API Key

```
public class ApiClient {
          private static final String API_KEY = "1234567890abcdef";
          private static final String DB_PASSWORD = "supersecret";

          public static void connect() {
          System.out.println("Using API Key: " + API_KEY);
          }
}
```

Kotlin Example: Hardcoded Database Credentials

```
object DatabaseConfig {
        const val DB_USER = "admin"
        const val DB_PASSWORD = "password123"
        const val API_URL = "http://example.com/api"
}
```

How to Find Hardcoded Secrets?

- Example regex to detect API keys:

```
(?:api_key|apikey|secret|token|password|key)\\s*[:=]\\s*[""]?[0-9a-zA-Z]{32,}[""]?
```

Introduction to Insecure Logging

- Logging is essential for debugging and monitoring but can be a security risk.
- Insecure logging occurs when:
  - User credentials, authentication tokens, payment details, or PII (Personally Identifiable Information) are logged.
  - Logs are stored unencrypted in files accessible to unauthorized users.
  - Verbose logging is left enabled in production.

Impact of Insecure Logging

- Data Leakage Risk:
  - Usernames, passwords, session tokens, and API keys can be exposed in logs.
  - Attackers with access to logs can bypass authentication.
- Regulatory Violations:
  - GDPR, HIPAA, PCI-DSS, and ISO 27001 prohibit storing sensitive data in logs.
- Exposes Internal Application Logic:
  - Logging full request/response cycles may expose backend structure to

Impact of Insecure Logging

- Example Log Leak Scenario:
  - If an application logs requests containing passwords:

```
INFO: User Login Attempt - Username: admin, Password: SuperSecret123
```

Java Example: Logging Sensitive User Data

```
import java.util.logging.Logger;

public class InsecureLogging {
          private static final Logger LOGGER =
Logger.getLogger(InsecureLogging.class.getName());

          public void login(String username, String password) {
          LOGGER.info("User login attempt - Username: " + username + ", Password: " +
password);
          }
}
```

Kotlin Example: Logging Session Tokens

```
import android.util.Log

class SecureSessionManager {
        fun authenticate(user: String, sessionToken: String) {
        Log.d("Auth", "User $user authenticated with token:
$sessionToken")
        }
}
```

Introduction to Insecure Storage

- What is insecure storage?
  - Storing credentials, API keys, or sensitive user data in unencrypted storage locations.
  - Examples: SharedPreferences, external storage (SD card), SQLite databases, app private directories.
- Why is it a problem?
  - Attackers can extract sensitive data from app storage using root access, malware, or forensic tools.

Introduction to Insecure Storage

- Common insecure storage locations in Android apps:
  - SharedPreferences (unencrypted)
  - External storage (SD card, public directories)
  - Local SQLite databases (without encryption)

Impact of Insecure Storage

- Credentials & PII Theft
  - Attackers can extract usernames, passwords, and tokens from storage.
- Account Takeovers & Session Hijacking
  - If session tokens are stored insecurely, attackers can reuse them for unauthorized access.
- Regulatory & Compliance Violations
  - Storing sensitive data in plaintext may violate GDPR, PCI-DSS, HIPAA, and ISO 27001.

Java Example: Storing Passwords in SharedPreferences

```
import android.content.Context;
import android.content.SharedPreferences;

public class InsecureStorage {
            public void saveCredentials(Context context, String username, String password) {
            SharedPreferences prefs = context.getSharedPreferences("UserPrefs", Context.MODE_PRIVATE);
            SharedPreferences.Editor editor = prefs.edit();
            editor.putString("username", username);
            editor.putString("password", password);
            editor.apply();
            }
}
```

SharedPreferences is not encrypted and is accessible by rooted devices or malware.

Kotlin Example: Saving API Keys in External Storage

```
import android.content.Context
import android.os.Environment
import java.io.File

class InsecureFileStorage {
          fun saveAPIKey(context: Context, apiKey: String) {
          val file = File(Environment.getExternalStorageDirectory(), "api_key.txt")
          file.writeText(apiKey)
          }
}
```

External storage is world-readable, meaning any app can access the API key.

Introduction to Weak Cryptography

- What is Weak Cryptography?
  - Using hardcoded encryption keys in source code.
  - Using outdated, weak, or easily breakable encryption algorithms.
- Why is it dangerous?
  - Attackers can extract hardcoded keys from source code or reverse-engineered APKs.
  - Weak encryption (e.g., MD5, DES) can be cracked using modern tools.
  - If encryption is not implemented securely, data can still be exposed.

Introduction to Weak Cryptography

- Examples of Weak Crypto Practices:
  - Hardcoded keys inside source code
  - Using weak algorithms like MD5, DES, RC4
  - Not using proper encryption modes (ECB instead of CBC or GCM)

Impact of Weak Cryptography

- Data Exposure & Unauthorized Decryption
  - Attackers can decrypt sensitive data using exposed or weak keys.
  - Example: If an API key is encrypted with a hardcoded key, an attacker can extract and decrypt it.
- Regulatory & Compliance Risks
  - Weak cryptography violates security standards like GDPR, PCI-DSS, NIST, ISO 27001.
  - Payment processors & banking apps must use strong encryption (AES-256, TLS 1.3).

How to Find Weak Cryptography in Code?

- Look for hardcoded encryption keys, weak algorithms, and ECB mode usage.
  - Search for AES/DES/RSA keys inside code
  - Search for MD5 or SHA-1 usage (weak hashes)

```
(AES|DES|RSA)\\s*\\(\\s*\"([0-9A-Fa-f]{16,})\"

MessageDigest.getInstance\\(\"(MD5|SHA-1)\"\\)
```

Java Example: Hardcoded AES Key

```
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;

public class WeakCrypto {
            private static final String AES_KEY = "1234567812345678";

            public static String encrypt(String data) throws Exception {
            Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
            SecretKeySpec keySpec = new SecretKeySpec(AES_KEY.getBytes(), "AES");
            cipher.init(Cipher.ENCRYPT_MODE, keySpec);
            return Base64.getEncoder().encodeToString(cipher.doFinal(data.getBytes()));
            }
}
```

Kotlin Example: Using Weak MD5 Hashing

```
import java.security.MessageDigest

class WeakHashing {
        fun hashPassword(password: String): String {
        val md = MessageDigest.getInstance("MD5")
        return md.digest(password.toByteArray()).joinToString("") {
"%02x".format(it) }
        }
}
```

Introduction to WebView Vulnerabilities

- What is WebView?
  - WebView is an Android component that allows apps to display web content.
  - It renders HTML, executes JavaScript, and loads external URLs within the app.
- Why is it dangerous?
  - If an app loads untrusted URLs or enables JavaScript, attackers can:
  - Inject malicious scripts into WebView (Cross-Site Scripting - XSS).
  - Steal session cookies, credentials, and sensitive user data.
  - Exploit Android WebView flaws to gain control over the app.

Impact of WebView Vulnerabilities

- Cross-Site Scripting (XSS) Attacks
  - If an app loads user-controlled URLs, attackers can inject malicious JavaScript.
- Data Theft & Phishing
  - Attackers can load fake login pages in WebView to steal credentials.
- Arbitrary Code Execution via JavaScriptInterface
  - Exposing Java methods to JavaScript allows attackers to execute arbitrary code.

How to Detect WebView Vulnerabilities?

- Manual Code Review
  - Search for setJavaScriptEnabled(true) in WebView configurations.
  - Look for loadUrl() calls that accept user input or external sources.

Java Example: Unrestricted URL Loading

```
import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebView;

public class InsecureWebViewActivity extends Activity {
           @Override
           protected void onCreate(Bundle savedInstanceState) {
           super.onCreate(savedInstanceState);
           WebView webView = new WebView(this);
           setContentView(webView);

           String url = getIntent().getStringExtra("url");
           webView.loadUrl(url);
           }
}
```

Kotlin Example: JavaScript Execution & JavaScriptInterface

```
import android.content.Context
import android.webkit.JavascriptInterface
import android.webkit.WebView

class InsecureWebView(context: Context) {
                private val webView: WebView = WebView(context)

                init {
                webView.settings.javaScriptEnabled = true
                webView.addJavascriptInterface(this, "Android")
                webView.loadUrl("http://example.com")
                }
                @JavascriptInterface
                fun showToast(message: String) {
                println("JavaScript called: $message")
                }
}
```

Introduction to Insecure Network Communication

- What is insecure network communication?
  - Using HTTP instead of HTTPS to transmit sensitive data.
  - Ignoring SSL certificate validation, allowing man-in-the-middle (MITM) attacks.
  - Using self-signed or expired certificates without verification.

Introduction to Insecure Network Communication

- Why is it dangerous?
  - Attackers can intercept and modify data sent over HTTP.
  - Unvalidated SSL certificates allow MITM attacks where attackers can spoof legitimate servers.
  - Leads to data leaks, credential theft, and session hijacking.

Impact of Insecure Network Communication

- Man-in-the-Middle (MITM) Attacks
  - Attackers can intercept HTTP traffic and steal credentials or modify responses.
  - Example: A login request sent via HTTP can be hijacked to capture passwords.
- Session Hijacking & Data Theft
  - Attackers can capture authentication tokens, credit card details, and personal data.
- Regulatory & Compliance Violations

How to Find Insecure Network Communication?

- Manual Code Review
  - Look for HTTP URLs used in networking libraries.
  - Check for custom SSL handling that ignores verification.

Java Example: Using HTTP Instead of HTTPS

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class InsecureHttpClient {
              public static void sendRequest() throws Exception {
              URL url = new URL("http://example.com/api");
              HttpURLConnection connection = (HttpURLConnection) url.openConnection();
              connection.setRequestMethod("GET");

              BufferedReader reader = new BufferedReader(new InputStreamReader(connection.getInputStream()));
              String response;
              while ((response = reader.readLine()) != null) {
              System.out.println(response);
              }
              reader.close();
              }
}
```

Kotlin Example: Ignoring SSL Certificate Validation

```
import javax.net.ssl.*

class InsecureSSLClient {
            fun createInsecureClient(): SSLSocketFactory {
            val trustAllCerts = arrayOf<TrustManager>(object : X509TrustManager {
            override fun checkClientTrusted(chain: Array<java.security.cert.X509Certificate>, authType: String) {}
            override fun checkServerTrusted(chain: Array<java.security.cert.X509Certificate>, authType: String) {}
            override fun getAcceptedIssuers(): Array<java.security.cert.X509Certificate>? = null
            })

            val sslContext = SSLContext.getInstance("TLS")
            sslContext.init(null, trustAllCerts, java.security.SecureRandom())
            return sslContext.socketFactory
            }
}
```

Introduction to Reflection & Dynamic Code Execution

- What is Reflection?
  - Reflection allows programs to inspect and modify code at runtime dynamically.
  - Used for loading classes, invoking methods, and modifying objects without compile-time knowledge.

Introduction to Reflection & Dynamic Code Execution

- What is Dynamic Code Execution?
  - Dynamically loading external or runtime-generated code using:
  - Reflection (`Class.forName()`, `Method.invoke()`)
  - Dynamic class loading (`DexClassLoader`, `PathClassLoader`)
  - Runtime script execution (`JavaScriptInterface`, `Runtime.exec()`)

Introduction to Reflection & Dynamic Code Execution

- Why is this dangerous?
  - Attackers can inject malicious code and execute it at runtime.
  - Can be used for code obfuscation by malware to evade detection.
  - Enables loading and executing untrusted code dynamically.

Impact of Reflection & Dynamic Execution Vulnerabilities

- Arbitrary Code Execution
  - Attackers can use reflection to invoke restricted methods, bypassing security controls.
- Bypassing Security Restrictions
  - Private methods and fields can be accessed and modified, leading to unauthorized actions.
- Loading & Executing Malicious Code
  - Dynamic code execution allows malware to inject and run harmful payloads.

How to Find Reflection & Dynamic Execution Vulnerabilities?

- Manual Code Review
  - Look for `Class.forName()`, `Method.invoke()`, and dynamic class loaders.
  - Check for runtime execution methods like `Runtime.exec()`.

Java Example: Executing Private Methods via Reflection

```
import java.lang.reflect.Method;

public class ReflectionExample {
          public static void main(String[] args) throws Exception {
          Class<?> cls = Class.forName("com.example.SecureClass");
     Method method = cls.getDeclaredMethod("getSecretData");
          method.setAccessible(true);
          String data = (String) method.invoke(cls.newInstance());
          System.out.println("Sensitive Data: " + data);
          }
}
```

Kotlin Example: Dynamic Code Loading with DexClassLoader

```
import dalvik.system.DexClassLoader
import java.io.File

class DynamicCodeLoader {
          fun loadMaliciousCode(apkPath: String, context: android.content.Context) {
          val optimizedDir = context.cacheDir.absolutePath
          val dexClassLoader = DexClassLoader(apkPath, optimizedDir, null, context.classLoader)
          val loadedClass = dexClassLoader.loadClass("com.malicious.Backdoor")
          val instance = loadedClass.getDeclaredConstructor().newInstance()
          val method = loadedClass.getDeclaredMethod("executePayload")
          method.invoke(instance)
          }
}
```

Static Analysis of iOS Apps

Common Vulnerabilities to Look for

- Hardcoded API Keys and Secrets – Finding credentials in the binary.
- Insecure Data Storage – Hardcoded sensitive data in the app.
- Weak Cryptography – Improper encryption methods.
- URL Scheme Hijacking – Unsafe URL handling.
- Insecure Logging – Sensitive data in logs.

Common Vulnerabilities to Look for

+ Insecure Keychain Storage – Improper usage of Keychain.
+ Jailbreak Detection Bypass – Easily bypassable jailbreak detection.
+ Unsafe HTTP Communication – Using HTTP instead of HTTPS.
+ Dynamic Library Injection – Weak code that can be exploited by runtime modifications.
+ Reverse Engineering Protections – Simple techniques to detect reverse engineering.

Introduction to Hardcoded API Keys and Secrets

- API keys and secrets are often embedded in mobile applications to authenticate API requests.
- Hardcoding these secrets into the binary exposes them to attackers who can extract and misuse them.
- A common vulnerability in iOS applications that can lead to unauthorized access to backend services.

Impact of Hardcoded API Keys

- Unauthorized Access – Attackers can extract API keys and abuse backend services.
- Service Abuse – API rate limits may be bypassed, leading to account misuse.
- Financial Loss – Stolen credentials could be used to incur billing charges on cloud services.
- Regulatory Compliance Risks – Violates best practices like OWASP Mobile Top 10 (M1: Improper Credential Storage).

How Attackers Find Hardcoded Secrets

```
+ strings Pentesterbook | grep "api"

+ class-dump -H Pentesterbook.app/Pentesterbook -o headers/
  + grep -r "API_KEY" headers/
```

Example of Hidden API Keys in Code (Swift)

- Swift
  - `let apiKey = "sk_live_1234567890abcdef"`
  - `let authToken = "Bearer 123456abcdef"`
- Objective-C
  - `NSString *apiKey = @"sk_test_ABCDEFG1234567";`

Introduction to Insecure Data Storage

- Mobile apps often store sensitive data such as user credentials, session tokens, API keys, encryption keys, and personal information.
- If stored insecurely, this data can be extracted by attackers from the app bundle, logs, or even memory.
- Improper storage practices violate OWASP Mobile Top 10 (M2: Insecure Data Storage).

Impact of Insecure Data Storage

- Credential Theft – Attackers can extract stored passwords or session tokens.
- Data Breaches – Exposure of personally identifiable information (PII).
- Regulatory Non-Compliance – Violates GDPR, ISO27001, PCI DSS security standards.
- Insecure App Modifications – Attackers can alter stored settings to bypass security mechanisms (e.g., disabling 2FA).
- Session Hijacking – Attackers can reuse stored tokens to impersonate users.

How Attackers Find Sensitive Data in the App?

- Static Analysis
  - `grep -r "password"`
- Extracting Data from UserDefaults (Jailbroken Device)
  - `plutil -p /var/mobile/Containers/Data/Application/{App_ID}/Library/Preferences/com.pentesterbook.plist`

How Attackers Find Sensitive Data in the App?

- Extracting Sensitive SQLite Data
  - `sqlite3 /var/mobile/Containers/Data/Application/{App_ID}/Documents/userdata.db`
  - `select * from users;`
- Reading Logs for Leaked Data
  - `log stream | grep "password"`

- Swift - Logging Sensitive Data
  - `let userToken = "123456789abcdef"`
  - `print("User logged in with token: \(userToken)")`
- Objective-C - Logging Sensitive Data
  - `NSString *sessionToken = @"session_abcdef12345";`
  - `NSLog(@"User session: %@", sessionToken);`

Introduction to Weak Cryptography

- Common weak encryption methods include:
  - Base64 Encoding instead of Encryption
  - Using Hardcoded Encryption Keys
  - Using Deprecated or Weak Algorithms (e.g., MD5, DES)
  - Lack of Key Management (storing keys in source code)
- Weak cryptography violates OWASP Mobile Top 10 (M5: Insecure Cryptography).

Introduction to Weak Cryptography

- Swift
  - `let password = "SuperSecret123"`
  - `let encoded = Data(password.utf8).base64EncodedString()`
- Objective-C
  - `NSString *password = @"SuperSecret123";`
  - `NSData *data = [password dataUsingEncoding:NSUTF8StringEncoding];`
  - `NSString *encoded = [data base64EncodedStringWithOptions:0];`

Impact of Weak Cryptography

- Data Exposure: Weak encryption makes data easily reversible.
- Regulatory Compliance Issues: Fails security standards like GDPR, PCI DSS, HIPAA.
- Credential Theft: Attackers can decode stored credentials easily.
- Man-in-the-Middle Attacks (MITM): Weak cryptographic methods allow attackers to intercept and modify data.
- Ransomware & Data Manipulation: Weak or improperly implemented encryption allows data tampering.

How Attackers Find Weak Encryption?

```
+ strings Pentesterbook | grep "key"

+ class-dump -H Pentesterbook.app/Pentesterbook -o headers/
   +  grep -r "AES128" headers/

+ grep -E "MD5|DES|Base64" -r .
```

Swift - MD5

```
import CommonCrypto

func weakMD5Hash(input: String) -> String {
        let data = Data(input.utf8)
        var digest = [UInt8](repeating: 0, count: Int(CC_MD5_DIGEST_LENGTH))
        data.withUnsafeBytes { bytes in
        _ = CC_MD5(bytes.baseAddress, CC_LONG(data.count), &digest)
        }
        return digest.map { String(format: "%02x", $0) }.joined()
}
```

Objective-C - Using DES

```
#import <CommonCrypto/CommonCryptor.h>

void encryptWithDES(NSString *data) {
        char key[] = "weakkey1";
        size_t numBytesEncrypted = 0;
        uint8_t buffer[1024];

        CCCrypt(kCCEncrypt, kCCAlgorithmDES, kCCOptionECBMode,
        key, kCCKeySizeDES,
        NULL, [data UTF8String], [data length],
        buffer, 1024, &numBytesEncrypted);
}
```

Introduction to URL Scheme Hijacking

- iOS applications can define custom URL schemes (e.g., myapp://) to handle deep linking.
- URL schemes allow inter-app communication but can be hijacked by malicious apps if improperly implemented.
- Attackers can exploit unsafe URL handling to steal data, trigger unintended actions, or execute malicious code.
- This vulnerability is covered under OWASP Mobile Top 10 (M3: Insecure Communication).

Introduction to URL Scheme Hijacking

- Swift
  - `UIApplication.shared.open(URL(string: "myapp://login?token=abc123")!)`
- Objective-C
  - `[[UIApplication sharedApplication] openURL:[NSURL URLWithString:@"myapp://login?token=abc123"]];`

Impact of URL Scheme Hijacking

- Data Theft: Malicious apps can register the same URL scheme and intercept sensitive data (e.g., authentication tokens).
- Phishing Attacks: Attackers can trick users into opening rogue apps.
- Session Hijacking: If authentication tokens are passed via URL parameters, an attacker can capture and reuse them.
- Remote Code Execution: Unvalidated URL input can lead to command execution vulnerabilities.

How Attackers Exploit URL Scheme Hijacking?

+ `plutil -p Payload/MyApp.app/Info.plist | grep CFBundleURLSchemes`

+ Simulate a Malicious App Hijacking a URL Scheme
  + `xcrun simctl openurl booted myapp://auth?token=attack`

Swift - Unsafe URL Handling

```
func openPaymentPage() {
        let url = URL(string:
"myapp://payment?amount=100&user=admin")!
        UIApplication.shared.open(url)
}
```

Objective-C - Unsafe URL Handling

```
NSURL *url = [NSURL URLWithString:@"myapp://payment?amount=100&user=admin"];
[[UIApplication sharedApplication] openURL:url];
```

Swift - Missing Validation

```
func application(_ app: UIApplication, open url: URL, options:
[UIApplication.OpenURLOptionsKey : Any] = [:]) -> Bool {
        let urlString = url.absoluteString
        print("Opened via URL: \(urlString)")
        return true
}
```

Objective-C - Missing Validation

```
(BOOL)application:(UIApplication *)app openURL:(NSURL *)url
options:(NSDictionary<UIApplicationOpenURLOptionsKey,id>
*)options {
        NSLog(@"Opened via URL: %@", url.absoluteString);
        return YES;
```

Introduction to Insecure Logging

- Logging is essential for debugging, but sensitive data should never be logged.
- Insecure logging occurs when sensitive data such as passwords, API keys, session tokens, or PII (Personally Identifiable Information) is written to logs.

Introduction to Insecure Logging

- Attackers can extract logged sensitive information from:
  - System logs (log stream)
  - App-specific logs (NSLog, print)
  - Crash reports (sysdiagnose dumps)
  - Log files stored in the app sandbox

Impact of Insecure Logging

- Exposure of Sensitive Data: Attackers or malware can access logs containing passwords or session tokens.
- Regulatory Compliance Risks: Violates GDPR, PCI DSS, HIPAA, leading to penalties and fines.
- Reverse Engineering Risks: Analyzing app logs can reveal API endpoints, user data, or internal security mechanisms.
- Persistence of Data: Logs may persist after app termination or be included in crash reports, increasing the attack surface.

How Attackers Find Sensitive Data in Logs

- Searching Logs for Sensitive Data (Device Logs)
  - `log stream --level debug --predicate 'eventMessage contains "password"'`
- Extracting Logs from an iOS Device (Jailbroken)
  - `cat /private/var/mobile/Containers/Data/Application/{App_ID}/Library/Logs/app.log`
- Static Analysis for Logging Statements
  - `grep -E "NSLog|print" -r .`

Swift - Logging Sensitive API Responses

```
let response = "{ \"token\": \"abcdef123456\" }"
print("API Response: \(response)")
```

Objective-C - Logging Session Data

```
NSString *sessionToken = @"abcdef123456";
NSLog(@"Session Token: %@", sessionToken);
```

Introduction to Insecure Keychain Storage

- The iOS Keychain is a secure storage system for sensitive data like passwords, API keys, and tokens.
- However, misconfigurations and improper usage can lead to vulnerabilities.
- Common insecure Keychain storage practices include:
  - Storing data without encryption
  - Using weak access control policies
  - Storing secrets that should not persist across device backups

Impact of Insecure Keychain Storage

- Unauthorized Access: If access controls are misconfigured, other apps or attackers can read stored data.
- Data Persistence Across Backups: If sensitive information is stored without restricting backups, it could be extracted from iCloud or iTunes backups.
- Credential Theft: Storing API keys, passwords, or tokens insecurely can lead to account takeovers.
- Regulatory Compliance Risks: Violates GDPR, PCI DSS, HIPAA, leading to penalties.

How Attackers Find Insecure Keychain Storage?

- Look for missing access controls in `kSecAttrAccessible`

```
+  grep -r "kSecClassGenericPassword" .
```

Swift - Storing Data with Weak Access Controls

```
let keychainQuery: [String: Any] = [
        kSecClass as String: kSecClassGenericPassword,
        kSecAttrAccount as String: "admin",
        kSecValueData as String: "SuperSecretKey".data(using: .utf8)!,
        kSecAttrAccessible as String: kSecAttrAccessibleAlways
]
SecItemAdd(keychainQuery as CFDictionary, nil)
```

Objective-C - Storing Data without Access Control

```
NSDictionary *keychainQuery = @{
        (__bridge id)kSecClass: (__bridge id)kSecClassGenericPassword,
        (__bridge id)kSecAttrAccount: @"admin",
        (__bridge id)kSecValueData: [@"SuperSecretKey"
dataUsingEncoding:NSUTF8StringEncoding],
        (__bridge id)kSecAttrAccessible: (__bridge id)kSecAttrAccessibleAlways
};
SecItemAdd((__bridge CFDictionaryRef)keychainQuery, NULL);
```

Introduction to Jailbreak Detection Bypass

- Jailbreaking removes iOS restrictions, allowing:
  - Installation of unauthorized apps
  - Modification of system files
  - Disabling of security mechanisms
- Apps (especially banking & corporate apps) implement jailbreak detection to prevent risks.
- However, many implementations are easily bypassed by attackers.

Impact of Bypassing Jailbreak Detection

- Attackers can modify app behavior (e.g., bypass authentication, inject malicious code).
- App data is exposed to other apps.
- Banking apps can be tricked into processing unauthorized transactions.
- Jailbroken devices weaken Keychain security, making credential theft easier.
- Dynamic libraries (e.g., Frida, Cydia Substrate) can manipulate app behavior.

How Attackers Bypass Jailbreak Detection?

- Check jailbreak detection: `strings MyApp | grep "Cydia"`
- Modifying the App's File Checks
  - `mv /Applications/Cydia.app /Applications/HiddenCydia.app`
- Attackers override jailbreak detection functions at runtime using Frida.
- Patching the Binary (Hex Editing): Locate jailbreak check return values and modify them

Introduction to Unsafe HTTP Communication

- HTTP (Hypertext Transfer Protocol) transmits data unencrypted, making it vulnerable to interception.
- HTTPS (Hypertext Transfer Protocol Secure) encrypts data using TLS (Transport Layer Security).

Impact of Using HTTP Instead of HTTPS

- Data Interception (e.g., passwords, API keys).
- Man-in-the-Middle (MITM) Attacks
- Session Hijacking
- Data Manipulation
- Regulatory Compliance Issues: Violates security policies like GDPR, PCI DSS, HIPAA.

How Attackers Exploit HTTP Communication?

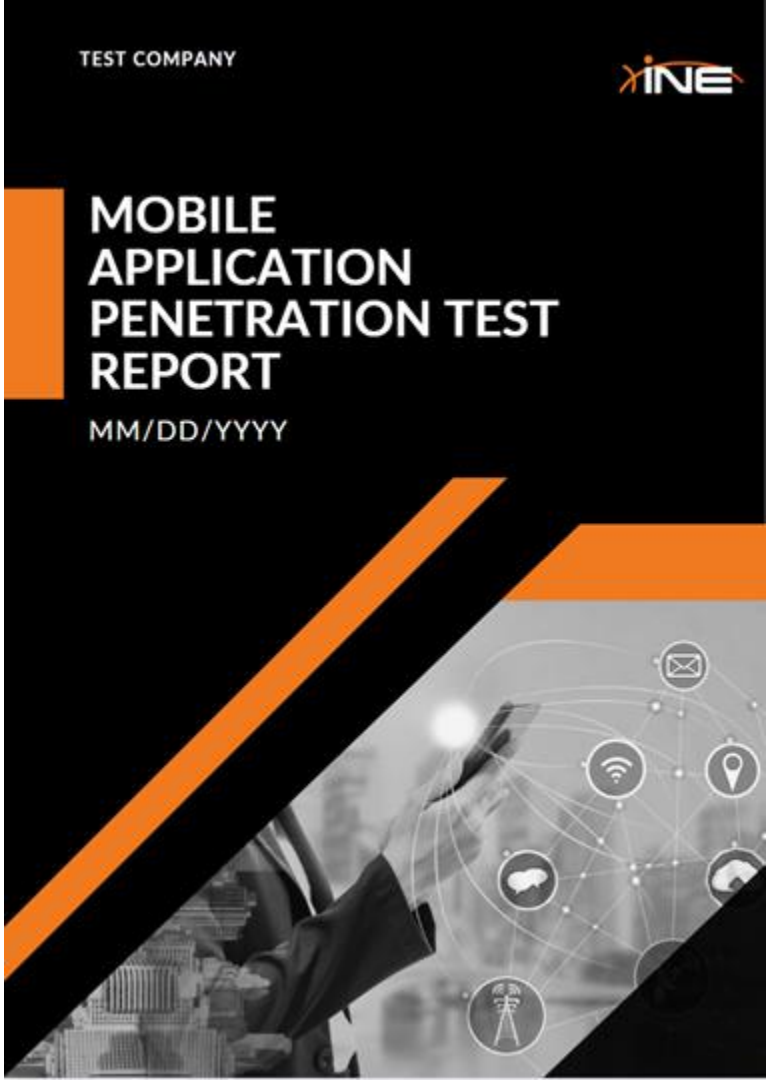
+ Wireshark or `tcpdump -i en0 port 80 -A`
+ Checking iOS App Network Configuration: inspecting the Info.plist file
+ `grep -r "http://" .`

Swift - Using HTTP Instead of HTTPS

```
let url = URL(string: "http://example.com/api")!
let task = URLSession.shared.dataTask(with: url) { data,
response, error in
        print("Received data: \(String(data: data!, encoding:
.utf8) ?? "")")
}
task.resume()
```

Reporting

Everything Leads to



Purpose of a Penetration Test Report

- Documents security weaknesses found in the assessment.
- Provides evidence to support findings.
- Offers recommendations to fix vulnerabilities.

Stakeholders Who Rely on the Report

+ Developers and IT teams.
+ Executives and decision-makers.
+ Compliance officers and auditors.

Structuring a Professional Penetration Test Report

- Executive Summary
- Assessment Scope and Methodology
- Findings and Risk Ratings
- Technical Details of Vulnerabilities
- Remediation Recommendations
- Appendices (Raw Data, Logs, Tools Used)

Writing the Executive Summary and Scope

- Executive Summary:
  - Should be concise, non-technical, and address key business risks.
  - Includes an overview of vulnerabilities found and their impact.

Writing the Scope

- Defining Scope and Methodology:
  - Clearly state what was tested (e.g., platform, versions, APIs, backend services).
  - Tools and techniques used (static analysis, dynamic testing, reverse engineering).
- This can keep you out of trouble!

Documenting Findings

- How to Present Findings Clearly:
  - Use CWE (Common Weakness Enumeration) and OWASP Mobile Top 10 references.
  - Provide technical details with proof (screenshots, logs, PoC scripts).
  - Explain business impact and potential attack scenarios.

Risk Assessment and Prioritization

- Use frameworks like CVSS (Common Vulnerability Scoring System).
- Categorize vulnerabilities as Critical, High, Medium, or Low.

Vulnerability	CVSS Score	Risk Level	Impact
SQL Injection	9.8	Critical	Data Breach

Writing Remediation Steps and Actionable Recommendations

- Effective Remediation Steps:
  - Be clear and concise.
  - Provide both short-term (quick fixes) and long-term (best practices) solutions.
  - Use secure coding guidelines as references.

Finalizing the Report

- Polishing and Reviewing the Report:
  - Proofreading and clarity checks.
  - Ensuring all technical findings are supported with evidence.

Remember

- Your test is only as good as your report
- Keep language professional and neutral.
- Avoid blaming developers or teams.
- Make sure recommendations are feasible and realistic.

Summary