Lecture #4

Practical Mobile Application Security

SMA 2025

Today's Agenda

- **Part 1: The Attacker's Playbook** - Introduction to the OWASP Mobile Top 10 (2024).
- **Part 2: The Security Tester's Toolkit** - SAST, DAST, and IAST.
- **Part 3: Under the Hood** - Reverse Engineering Android (APK) & iOS (IPA) files.
- **Part 4: Writing Bulletproof Code** - Secure Coding Best Practices.
- **Part 5: Live Demo** - A mini vulnerability assessment.

Recap & The Goal

- **Lecture 1:** We learned about the CIA triad and the mobile threat landscape.
- **Lecture 2:** We saw how social engineering targets users and how to manage permissions and local data securely.
- **Lecture 3:** We explored the app stores, the vetting process, and the dangers of sideloading.

Today's Goal: To think like an attacker, find vulnerabilities, and then put on our developer hat to fix them.

Part 1: The Attacker's Playbook

An Introduction to the OWASP Mobile Top 10

What is OWASP?

The Open Worldwide Application Security Project

- A worldwide non-profit organization focused on improving the security of software.
- All of their resources are free, open, and created by a community of experts.
- They are most famous for their "Top 10" lists, which raise awareness of the most critical security risks.

The OWASP Mobile Top 10 (2024)

This is the "cheat sheet" for mobile security assessment. It tells you the 10 most common and critical risk categories.

- **M1:** Improper Credential Usage
- **M2:** Inadequate Supply Chain Security
- **M3:** Insecure Authentication/Authorization
- **M4:** Insufficient Input/Output Validation
- **M5:** Insecure Communication
- **M6:** Inadequate Privacy Controls
- **M7:** Insufficient Binary Protections
- **M8:** Security Misconfiguration
- **M9:** Insecure Data Storage
- **M10:** Insufficient Cryptography

M1: Improper Credential Usage

This is the new #1 risk. It focuses on how your app handles secrets (keys, tokens, passwords).

- **Examples:Hardcoding API keys** or passwords in the source code (we'll find this later!).
- Storing sensitive session tokens in insecure locations like or .
- Improperly using or managing keys in the Android Keystore or iOS Keychain.

M2: Inadequate Supply Chain Security

This is a new category about the risks from third-party code.

- **Goes beyond just your code:** Are you using an old, vulnerable analytics or advertising SDK?
- Are you vetting the open-source libraries you add to your or ?
- A vulnerability in a library you use is a vulnerability in your app.

M3: Insecure Authentication/Authorization

This merges the old M4 and M6. It's about who you are (Authentication) and what you can do (Authorization).

- Using weak password policies.
- Poor session management (e.g., tokens that never expire).
- Not protecting against brute-force attacks on a login.
- **Critical Flaw:** Performing authorization checks on the client-side instead of the server-side (e.g., an flag in the app's code).

The Rest of the 2024 List

We'll see examples of these throughout the lecture:

- **M4 (Insufficient Input/Output Validation):** (Old M7) Flaws like local SQL injection. We'll cover this in Part 4.
- **M5 (Insecure Communication):** (Old M3) Using HTTP, no certificate pinning. We'll cover this in Part 4.
- **M7 (Insufficient Binary Protections):** (Old M8+M9) This combines Code Tampering and Reverse Engineering. This is what we'll discuss in Part 3.
- **M8 (Security Misconfiguration):** (Old M10) Renamed. Leaving debug code, backdoors, or developer menus.
- **M9 (Insecure Data Storage):** (Old M2) Storing sensitive data insecurely on the device.
- **M10 (Insufficient Cryptography):** (Old M5) Using broken algorithms or "rolling your own" crypto.

Part 2: The Security Tester's Toolkit

SAST, DAST, and IAST

The Three Pillars of AppSec Testing

- **Static Application Security Testing (SAST)**
- “White-box” testing. You have the source code.
- "Black-box" testing. You only have the running application.
- "Gray-box" testing. A hybrid of the two.

What is SAST?

Static Application Security Testing

- Analyzes an application's source code (or compiled code) from the inside out.
- It does this **without** executing the application.
- Think of it as a super-powered linter that is focused exclusively on security flaws.

How SAST Works

It builds a model of your application and looks for dangerous patterns.

- **Control-Flow Analysis:** How does the application execute?
- **Data-Flow Analysis (Taint Analysis):** How does data move through the application? Does user input ever reach a dangerous function without being cleaned? (As seen in Lecture 3).

SAST Tools for Mobile

- **Android Studio / Xcode:** The built-in code inspectors can find basic security issues.
- **Semgrep:** A powerful, open-source, and fast SAST tool that uses simple rules.
- **Mobile Security Framework (MobSF):** An all-in-one open-source tool that can perform static analysis on compiled APKs and IPAs.
- **Commercial Tools:** Veracode, Checkmarx, SonarQube.

SAST Example: Finding a Hardcoded Key

Let's say we want to find developers who have hardcoded an AWS access key.

[Code Snippet: Kotlin]

```kotlin
// A developer accidentally commits a secret key
  class ApiClient {
     private val awsAccessKey = "AKIAIOSFODNN7EXAMPLE" // Bad! (M1)
     // ...
  }
```

[Code Snippet: YAML (Semgrep Rule)] see: https://github.com/semgrep/semgrep

 # A simplified Semgrep rule

```yaml
  rules:
  - id: aws-access-key
    pattern: '"AKIA[0-9A-Z]{16}"'
    message: "An AWS access key has been hardcoded in the source."
    languages: [java, kotlin, swift]
    severity: ERROR
```

What is DAST?

Dynamic Application Security Testing

- Analyzes a **running** application from the outside in.
- It has no knowledge of the source code. It behaves like a real-world attacker.
- It focuses on the application's inputs and outputs, especially network traffic.

How DAST Works

- **Proxying Traffic:** The core of mobile DAST is intercepting the communication between the app and its backend servers.
- **Fuzzing:** Sending unexpected or malicious data to the app's inputs (e.g., login forms, API parameters) to see if it crashes or behaves unexpectedly.
- **Monitoring:** Watching for data leaks, crashes, or error messages that reveal information.

The DAST Setup: The Intercepting Proxy

The most important tool for mobile DAST is an intercepting proxy.

- **Popular Tools:Burp Suite:** The industry standard (commercial, with a free edition).
- **OWASP ZAP:** The best open-source alternative.

How a Proxy Works

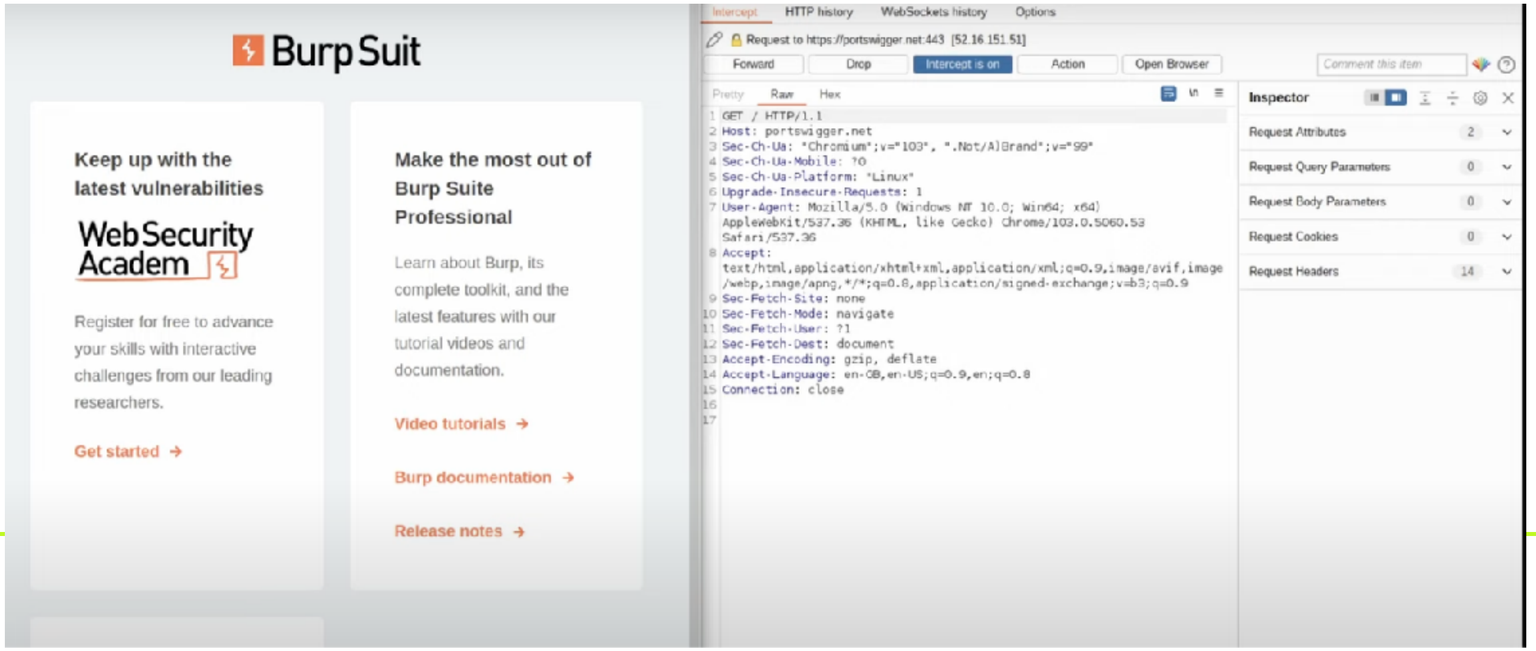
Setting Up a Proxy for Mobile

This is a critical skill for any mobile security tester.

- **Configure the Proxy:** Set up Burp Suite or ZAP to listen on your computer's IP address.
- **Configure the Mobile Device:** Go to your phone's Wi-Fi settings and manually configure the proxy to point to your computer's IP and port.
- **Install the Root Certificate:** Open the phone's browser and navigate to the proxy's address (e.g.,). Download and install the proxy's custom CA certificate. **This is the step that allows the proxy to decrypt HTTPS traffic.**

DAST Example: Finding an Insecure API Call

- **Vulnerabilities Found:M5: Insecure Communication:** The app is using HTTP, not HTTPS. The password is sent in cleartext.
- **M3: Insecure Authentication/Authorization:** The password "password123" was accepted by the server, indicating a weak password policy.



What is IAST?

Interactive Application Security Testing

- The "gray-box" hybrid approach.
- It uses an "agent" instrumented inside the running application to combine SAST-like code analysis with DAST-like observation of real-time behavior.
- It's like having a SAST tool that can see exactly what's happening during a DAST scan.

How IAST Works

- An IAST agent is added to the application, often as a a library.
- A QA tester or DAST tool interacts with the application.
- The agent, running inside the app, watches the code execute.
- If it sees tainted data (like user input from a DAST scan) reach a dangerous sink (like a database query), it reports a vulnerability in real-time.

SAST vs. DAST vs. IAST

Feature	SAST (White-box)	DAST (Black-box)	IAST (Gray-box)
Require Source?	Yes	No	Yes (for instrumentation)
Finds...	Code quality issues, hardcoded secrets	Runtime issues, server misconfigurations	Both, with context
Accuracy	Can have many false positives	High, confirms exploitability	Highest, low false positives
When to use	Early in development (CI/CD)	Later in testing, on running app	During QA Integration testing

Part 3: Under the Hood

Reverse Engineering Android (APK) & iOS (IPA) files

Why Reverse Engineer? (Enabling M7)

- To understand how an app works without the source code.
- To find vulnerabilities that SAST might miss, like **M1 (Hardcoded Secrets)**.
- To enable **M7 (Insufficient Binary Protections)** - has a legitimate app been repackaged with malware (i.e., Code Tampering)?
- To bypass client-side security controls (**M3**).

The Android Package (APK)

An APK is just a ZIP file. You can rename to and extract it.

- **AndroidManifest.xml**: Declares permissions, components, etc. In binary XML format.
- **classes.dex**: The application's code, compiled into Dalvik Executable format. This is what we target.
- **res/**: Application resources (images, layouts).
- **lib/**: Native libraries (C/C++ code).
- **META-INF/**: Contains the app's signature.

Tools for Decompiling APKs

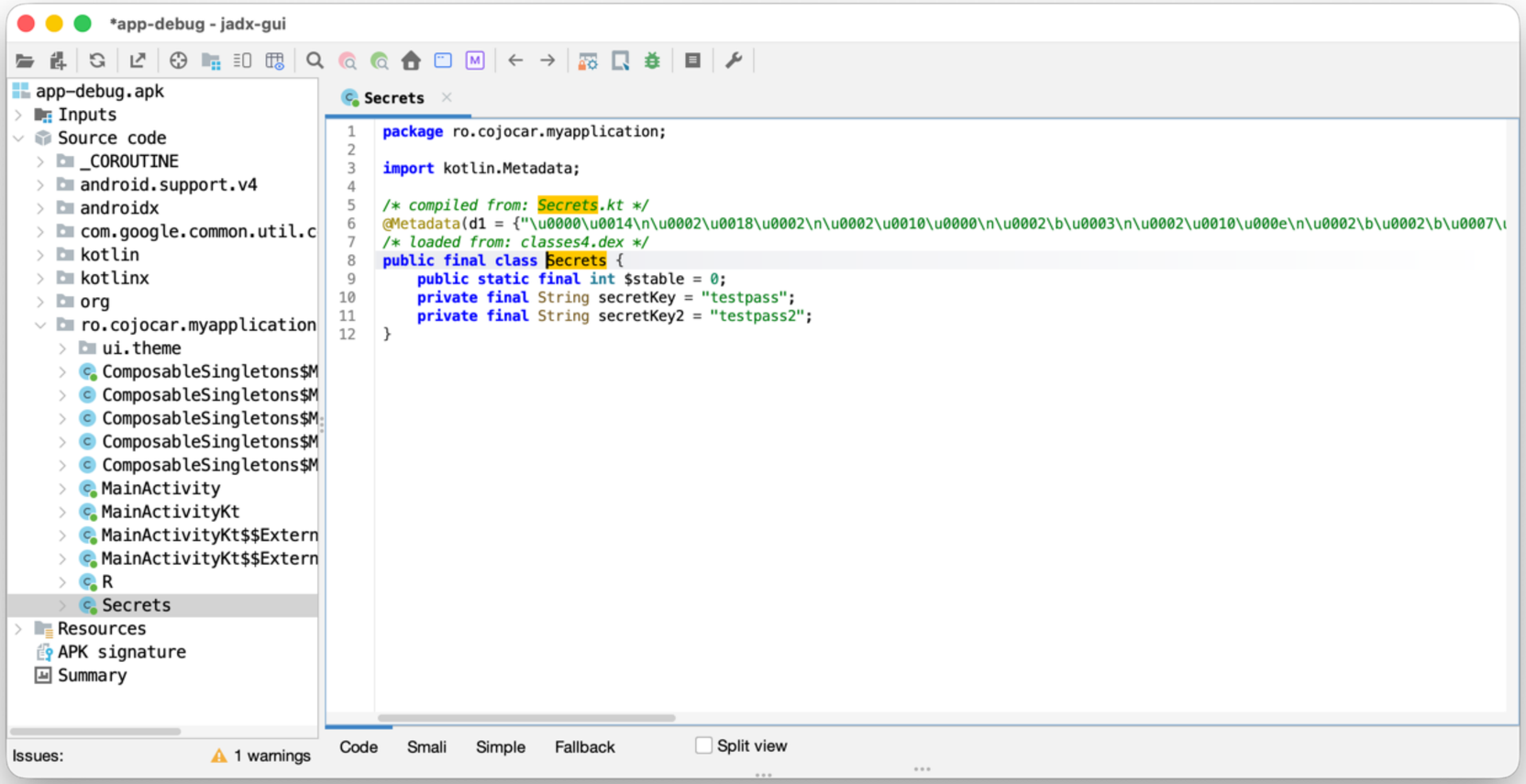
- **apktool:** A command-line tool that can decode an APK's resources (like) and disassemble its DEX files into a human-readable format called .
- **JAXD:** An amazing tool that can decompile DEX files directly back into mostly-readable Java code. It has both a command-line and GUI version.

Live Demo Plan: Decompiling an APK

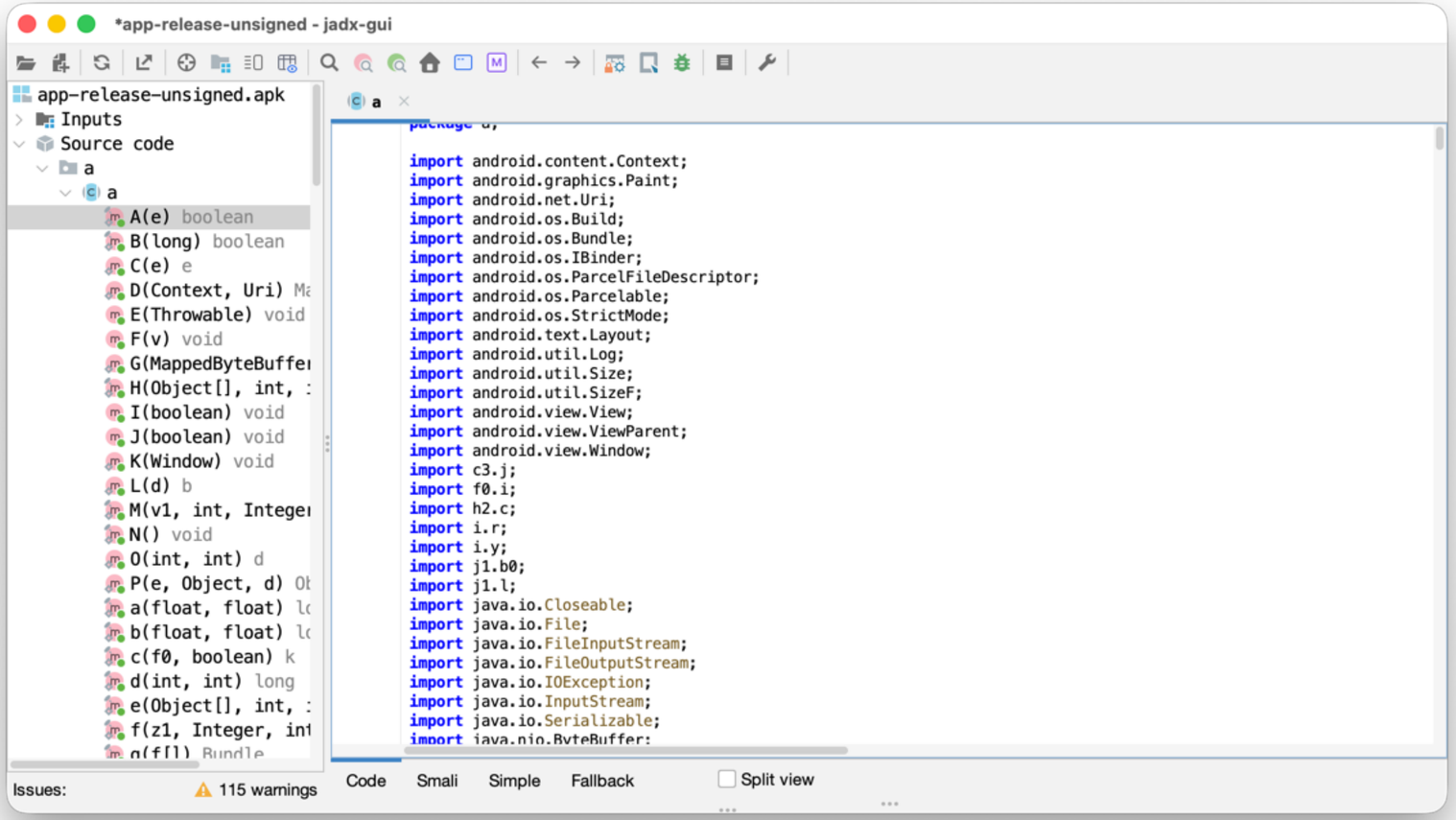
Goal: Find a hardcoded API key in a vulnerable app.

- Obtain the .**apk** file of our target application.
- Open the **.apk** file directly with **JADX**.
- JADX will decompile the **classes.dex** file automatically.
- Use the search function within JADX to search for common keywords like "api_key", "token", "password", etc.

JADX-GUI in Action



JADX-GUI in Action



What to Look For in Decompiled Code

- **Hardcoded Secrets (M1):** API keys, passwords, encryption keys.
- **Hidden Endpoints (M8):** URLs to staging servers or hidden developer APIs.
- **Logic Flaws (M3):** Is the "isPremiumUser" check done on the client-side? An attacker can patch this to get premium for free.
- **Disabled Security Features (M8):** Code that is commented out or a boolean flag that disables certificate pinning in a debug build.

The iOS Package (IPA)

An IPA is also just a ZIP file.

Payload/: This directory contains the main app bundle.

Payload/YourApp.app/: Inside the bundle, you'll find:

Info.plist: The app's metadata.

YourApp` (the binary): The compiled ARM machine code. This is our target.

Resources (images, storyboards).

Frameworks/: Embedded dynamic libraries.

The Challenge of iOS Reverse Engineering

Unlike Android's DEX files, the main binary in an IPA is a fully compiled ARM executable.

- You can't decompile it back to Swift or Objective-C easily.
- You must work with **assembly language (ARM64)**.
- This requires more advanced tools and a much steeper learning curve.

Tools for Analyzing iOS Binaries

- **otool/class-dump:** Command-line tools to extract information about the binary's structure and class interfaces.
- **Hopper/Ghidra/IDA Pro:** Professional disassemblers and decompilers that can analyze the ARM binary and provide pseudo-code.
- **Mobile Security Framework (MobSF):** Can perform automated static analysis on the IPA to extract strings, check for security settings, and identify basic flaws.

Live Demo Plan: Analyzing an IPA

Goal: Find sensitive URLs in an iOS app.

- Obtain the **.ipa** file (this is harder than Android, often requiring a jailbroken device to extract).
- Upload the **.ipa** to MobSF for automated analysis.
- MobSF will run its static analysis rules.
- We will check the "Strings" section of the report for interesting values, like URLs or keywords.

Defense: Obfuscation (M7)

How do we make reverse engineering harder? **Obfuscation.**

- **Goal:** To make the compiled code much more difficult for a human to understand, even after decompilation.
- **Techniques:Renaming:** Changing class, method, and variable names to meaningless characters (**a,b,c**).
- **String Encryption:** Encrypting string literals in the binary and only decrypting them in memory at runtime.
- **Control Flow Obfuscation:** Inserting junk code and complex jumps to make the logic hard to follow.

Android Obfuscation with R8/ProGuard

Android has a built-in tool called R8 (which incorporates ProGuard) that provides obfuscation.

Before R8:

```
class UserProfile {
    fun checkSubscriptionStatus() { ... }
}
```

After R8 (decompiled):

```
// The original class and method names are gone
public class a {
    public void a() { ... }
}
```

iOS Obfuscation

- **Less Common:** The compiled nature of iOS binaries makes them harder to reverse by default, so fewer developers use obfuscation.
- **Swift & Name Mangling:** The Swift compiler performs "name mangling" which already makes function names hard to read, providing a small level of natural obfuscation.
- **Third-Party Tools:** Tools like iXGuard or Obfuscator-LLVM can provide much stronger, commercial-grade obfuscation for iOS apps if needed.

Part 4: Writing Bulletproof Code

Secure Coding Practices in Action

From Offense to Defense

- We found hardcoded keys (SAST/Reversing). **Fix:** Don't hardcode them.
- We found insecure API calls (DAST). **Fix:** Use secure communication protocols.
- We found logic flaws (Reversing). **Fix:** Write better, more robust code.

Secure Coding: Input Validation (M4)

Principle: Never trust data coming from the client or any external source. Always sanitize and validate it.

- This is the primary defense against **injection attacks**.
- While mobile apps are less prone to classic SQL injection than web apps, it's still possible in local SQLite databases or if the app constructs backend queries from user input.

Input Validation: Android

Vulnerable Code (Local SQL Injection):

```
// User input is directly concatenated into the query
fun searchNotes(db: SQLiteDatabase, query: String) {
    val cursor = db.rawQuery("SELECT * FROM notes WHERE content = '$query'", null)
    // ...
}
// If query = "' OR '1'='1", the query becomes:
// SELECT * FROM notes WHERE content = '' OR '1'='1'
// This returns all notes, bypassing the intended logic.
```

Secure Code (Parameterization):

```
// Use '?' as a placeholder and provide the arguments separately.
fun searchNotes(db: SQLiteDatabase, query: String) {
    val cursor = db.rawQuery("SELECT * FROM notes WHERE content = ?", arrayOf(query))
    // The database driver handles safe substitution.
}
```

Input Validation: iOS

The same principle applies on iOS with Core Data.

Vulnerable Code (Injection in):

```
// User input is directly formatted into the predicate string
let userInput = "' OR 1==1"
let predicate = NSPredicate(format: "noteBody == \(userInput)")
// This can lead to unexpected behavior or data leakage.
```

Secure Code (Argument Substitution):

```
// Use %K for keys and %@ for values
let userInput = "My secret note"
let predicate = NSPredicate(format: "noteBody == %@", userInput)
// The framework handles safe substitution.
```

Secure Coding: Proper Cryptography (M10)

The Golden Rule: DO NOT ROLL YOUR OWN CRYPTO.

- Cryptography is extremely difficult to get right.
- Use well-vetted, high-level, modern libraries for any cryptographic operations.
- Never use old, broken algorithms like MD5 or SHA1 for anything security-related.

Modern Cryptography: AEAD

For most symmetric encryption needs, you should use an **Authenticated Encryption with Associated Data (AEAD)** cipher.

- **What it provides:**

  Confidentiality: The data is encrypted.
- **Integrity & Authenticity:** The data is signed with a MAC (Message Authentication Code). This prevents an attacker from tampering with the ciphertext.

Crypto Example: Android

Google's **Tink** library is the recommended way to do crypto on Android.

See: https://developers.google.com/tink/what-is

```
import com.google.crypto.tink.Aead
import com.google.crypto.tink.aead.AeadKeyTemplates
import com.google.crypto.tink.integration.android.TinkAndroid

// 1. Initialize Tink
TinkAndroid.init(applicationContext)

// 2. Generate a new key
val keyHandle = KeystoreHandle.generateNew(AeadKeyTemplates.AES256_GCM)

// 3. Get the AEAD primitive
val aead: Aead = keyHandle.getPrimitive(Aead::class.java)

// 4. Encrypt
val plaintext = "some sensitive data".toByteArray(UTF_8)
val associatedData = "my_associated_data".toByteArray(UTF_8)
val ciphertext: ByteArray = aead.encrypt(plaintext, associatedData)

// 5. Decrypt
val decrypted: ByteArray = aead.decrypt(ciphertext, associatedData)
```

Crypto Example: iOS

Apple's **CryptoKit** framework is the modern, Swifty way to do crypto on iOS.

See: https://developer.apple.com/documentation/cryptokit/

```
import CryptoKit

// 1. Generate a new symmetric key
let key = SymmetricKey(size: .bits256)

// 2. Data to be encrypted
let plaintext = "some sensitive data".data(using: .utf8)!
let associatedData = "my_associated_data".data(using: .utf8)!

// 3. Encrypt using AES-GCM (an AEAD cipher)
let sealedBox = try! AES.GCM.seal(plaintext, using: key, authenticating: associatedData)

// `sealedBox` contains the ciphertext, nonce, and authentication tag
```

```
// 4. Decrypt
let decryptedData = try! AES.GCM.open(sealedBox, using: key, authenticating: associatedData)
```

Secure Coding: Secure Communication (M5)

We saw in the DAST example how an attacker on our Wi-Fi can intercept HTTP traffic. HTTPS is the first step, but it's not enough.

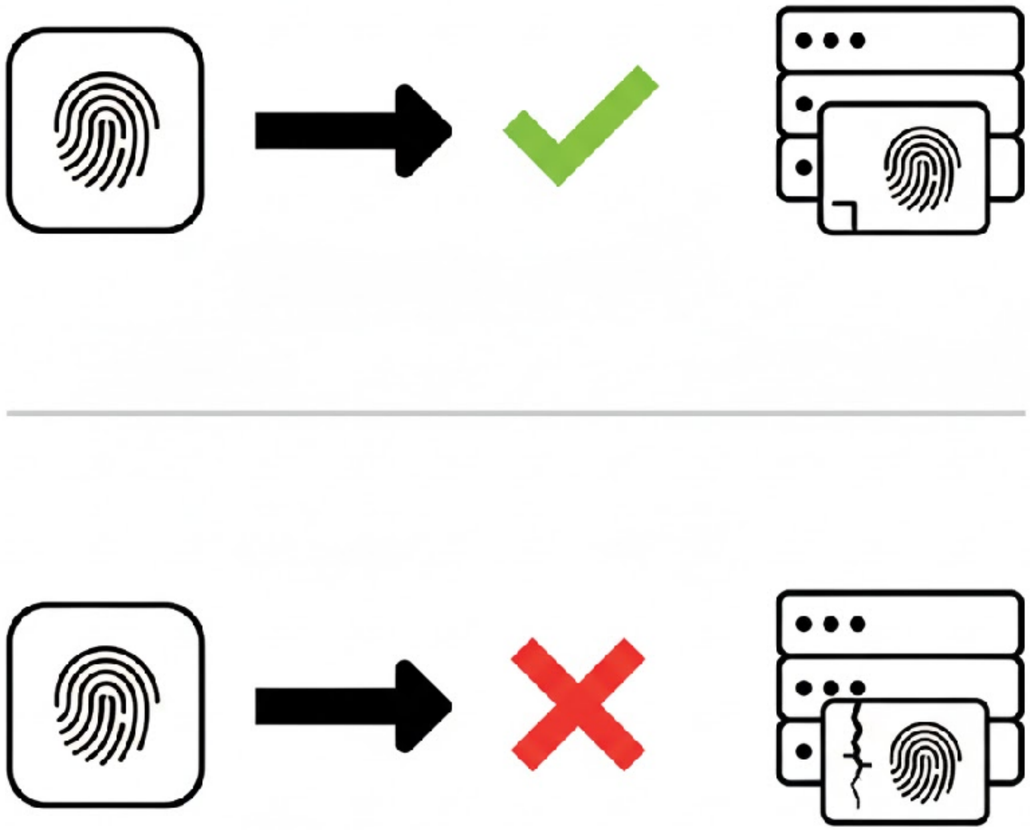
The Problem: An attacker can still perform a Man-in-the-Middle attack by tricking your phone into trusting a malicious root certificate (like we did with our Burp Suite setup).

The Solution: SSL/TLS Certificate Pinning

Certificate Pinning is the act of associating a host with their expected X.509 certificate or public key.

- **In simple terms:** You bake a fingerprint of the server's real certificate into your mobile app.
- When the app connects to the server, it compares the server's certificate to the fingerprint it has stored.
- If they don't match, the app knows something is wrong (a MitM attack!) and immediately terminates the connection.

How Pinning Works



Pinning Example: Android

OkHttp, the most popular networking library for Android, has built-in support for pinning.

```
import okhttp3.CertificatePinner
import okhttp3.OkHttpClient

val hostname = "publicobject.com"

// Create a CertificatePinner that specifies the SHA-256 hash of the server's public key.
val certificatePinner = CertificatePinner.Builder()
    .add(hostname, "sha256/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=")
    .build()

val client = OkHttpClient.Builder()
    .certificatePinner(certificatePinner)
    .build()

// All requests made with this client will now enforce the pin.
```

Pinning Example: iOS

On iOS, you can implement pinning by using and implementing a custom .

```
class PinningDelegate: NSObject, URLSessionDelegate {

    // The pinned public key hash
    private let pinnedHash = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA="

    func urlSession(_ session: URLSession,
                    didReceive challenge: URLAuthenticationChallenge,
                    completionHandler: @escaping (URLSession.AuthChallengeDisposition, URLCredential?) -> Void) {

        guard let trust = challenge.protectionSpace.serverTrust,
              SecTrustGetCertificateCount(trust) > 0 else {
            completionHandler(.cancelAuthenticationChallenge, nil)
            return
        }

        // Get the public key from the leaf certificate
        guard let certificate = SecTrustGetCertificateAtIndex(trust, 0),
              let publicKey = SecCertificateCopyKey(certificate),
```

```
class PinningDelegate: NSObject, URLSessionDelegate {

    // The pinned public key hash
    private let pinnedHash = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA="

    func urlSession(_ session: URLSession,
                    didReceive challenge: URLAuthenticationChallenge,
                    completionHandler: @escaping (URLSession.AuthChallengeDisposition, URLCredential?) -> Void) {

        guard let trust = challenge.protectionSpace.serverTrust,
              SecTrustGetCertificateCount(trust) > 0 else {
            completionHandler(.cancelAuthenticationChallenge, nil)
            return
        }

        // Get the public key from the leaf certificate
        guard let certificate = SecTrustGetCertificateAtIndex(trust, 0),
              let publicKey = SecCertificateCopyKey(certificate),
              let publicKeyData = SecKeyCopyExternalRepresentation(publicKey, nil) as? Data else {
            completionHandler(.cancelAuthenticationChallenge, nil)
            return
        }

        // Hash the public key and compare to our pinned hash
        let remoteHash = sha256(data: publicKeyData)

        if remoteHash == pinnedHash {
            // Success!
```

```
            return
        }

        // Get the public key from the leaf certificate
        guard let certificate = SecTrustGetCertificateAtIndex(trust, 0),
              let publicKey = SecCertificateCopyKey(certificate),
              let publicKeyData = SecKeyCopyExternalRepresentation(publicKey, nil) as? Data else {
            completionHandler(.cancelAuthenticationChallenge, nil)
            return
        }

        // Hash the public key and compare to our pinned hash
        let remoteHash = sha256(data: publicKeyData)

        if remoteHash == pinnedHash {
            // Success!
            completionHandler(.useCredential, URLCredential(trust: trust))
        } else {
            // Pin mismatch! Fail the connection.
            completionHandler(.cancelAuthenticationChallenge, nil)
        }
    }

    private func sha256(data: Data) -> String {
        // ... implementation of SHA-256 hashing ...
        return "..."
    }
}
```

The Pros and Cons of Pinning

- **Pros:**The most effective defense against network MitM attacks.
- **Brittleness:** If your server's certificate expires and you have to get a new one, your app will stop working until you release an update with the new pin. This can be a major operational headache.
- **Management:** You need a solid process for managing and rotating your pinned keys.

Q&A

Questions?