Mobile Programming

Mobile Security & MobSF

Lecturer: Ali Najimi
Faculty of Computer Engineering
Sharif University of Technology
Fall 2027

OWASP Top 10 | MobSF | Threat Modeling | Secure SDLC



Table of Contents

1. Why Mobile Security Matters
2. OWASP Mobile Top 10
3. Threat Modeling for Mobile Apps
4. Secure Storage & Key Management
5. Network Security & Certificate Pinning
6. Authentication & Session Security
7. Reverse Engineering & Code Protection
8. MobSF — Mobile Security Framework
9. MobSF — Static Analysis
10. MobSF — Dynamic Analysis
11. Mini Lab: Security Audit with MobSF
12. Summary & Key Takeaways

Why Mobile Security Matters

Mobile apps handle **sensitive data** — passwords, tokens, personal info, financial data.

The Threat Landscape:

- **80%+ of mobile apps** have at least one security vulnerability (OWASP).
- **Reverse engineering** an APK/IPA takes minutes with free tools.
- **Data at rest** can be extracted from rooted/jailbroken devices.
- **Data in transit** can be intercepted on public WiFi.
- **Malware** can be injected into repackaged apps.

Your Responsibility as a Developer:

1. Protect user data at rest and in transit.
2. Prevent unauthorized access to app functionality.
3. Make reverse engineering as difficult as possible.
4. Follow secure coding practices throughout the SDLC.

OWASP Mobile Top 10 (2024)

The **Open Worldwide Application Security Project** (OWASP) defines the top 10 mobile security risks.

#	Risk	Description
M1	Improper Credential Usage	Hardcoded credentials, API keys in source code
M2	Inadequate Supply Chain Security	3rd-party libraries with known vulnerabilities
M3	Insecure Authentication/Authorization	Weak login, missing session validation
M4	Insufficient Input/Output Validation	SQL injection, XSS in WebViews
M5	Insecure Communication	HTTP instead of HTTPS, no pinning
M6	Inadequate Privacy Controls	Excessive data collection, no consent

OWASP Mobile Top 10 (2024) — Continued

#	Risk	Description
M7	Insufficient Binary Protections	No obfuscation, debuggable builds
M8	Security Misconfiguration	Default settings, verbose logging
M9	Insecure Data Storage	Sensitive data in SharedPreferences, logs
M10	Insufficient Cryptography	Weak algorithms (MD5, DES), hardcoded keys

Key Takeaway: Most vulnerabilities come from **lazy defaults** and **rushed development.** Security must be intentional.

Threat Modeling for Mobile Apps

Threat modeling identifies potential threats before writing a single line of code.

STRIDE Framework:

Threat	Example in Mobile
Spoofing	Fake login screen, credential theft
Tampering	Repackaged APK with malware
Repudiation	Denying a transaction was made
Information Disclosure	Logs leaking sensitive data
Denial of Service	Battery drain via excessive network calls
Elevation of Privilege	Bypassing payment or admin checks

Steps:

1. **Identify Assets:** What data is valuable? (tokens, PII, payment info)
2. **Define Trust Boundaries:** Where does trusted data meet untrusted input?
3. **Identify Threats:** Use STRIDE to brainstorm.

Secure Storage & Key Management

Never store sensitive data in plain text.

Android Options

- **Keystore:** Hardware-backed key storage.
- **EncryptedSharedPreferences:** AES-256 encrypted KV store.
- **SQLCipher:** Encrypted SQLite.

iOS Options

- **Keychain Services:** Secure enclave–backed storage.
- **Data Protection:** File-level encryption (NSFileProtectionComplete).

```
// Flutter — Always use flutter_secure_storage
await storage.write(
  key: 'auth_token',
  value: token,
  iOptions: IOSOptions(
    accessibility:
      KeychainAccessibility
        .first_unlock_this_device,
  ),
  aOptions: AndroidOptions(
    encryptedSharedPreferences: true,
  ),
);
```

> **Rule:** If data can be used to impersonate a user or access their account, it goes in secure storage — nowhere else.

Network Security & Certificate Pinning

Protect data in transit against Man-in-the-Middle (MITM) attacks.

Defense Layers:

1. **HTTPS Only** — Never use plain HTTP.
2. **Certificate Pinning** — Embed known certificate hash.
3. **Network Security Config** (Android) — Enforce TLS in XML.
4. **App Transport Security** (iOS) — Enforce TLS by default.

```
// Flutter certificate pinning with Dio
final dio = Dio();
dio.httpClientAdapter = IOHttpClientAdapter(
  createHttpClient: () {
    final client = HttpClient();
    client.badCertificateCallback = (cert, host, port) {
      final sha256 = cert.sha256.toString();
      return sha256 == 'expected_hash_here';
    };
    return client;
  },
);
```

Authentication & Session Security

Common Vulnerabilities:

- Tokens stored in insecure locations.
- No token expiration or refresh mechanism.
- Session not invalidated on logout.
- Missing device binding.

Best Practices:

1. **Short-lived access tokens** (15–30 minutes).
2. **Refresh tokens** with rotation (one-time use).
3. **Secure storage** for all tokens.
4. **Invalidate on logout** — Clear tokens from storage AND server-side.
5. **Device binding** — Associate session with device fingerprint.
6. **Biometric authentication** — Use `local_auth` for sensitive operations.

```
// Proper logout
```

Reverse Engineering & Code Protection

How easy is it to reverse-engineer a mobile app?

- **Android APK:** `jadx` decompiles to readable Java/Kotlin in seconds.
- **iOS IPA:** `class-dump`, `Hopper`, `IDA Pro` reveal class structure.
- **Flutter:** `reFlutter` can dump Dart snapshot and analyze function calls.

Protection Strategies:

Technique	Android	iOS	Flutter
Code Obfuscation	ProGuard/R8	Swift optimization	`--obfuscate` flag
Root/Jailbreak Detection	✅ SafetyNet/Play Integrity	✅ DeviceCheck	✅ Packages available
Anti-Debugging	✅	✅	✅
App Signing Verification	✅	✅ (automatic)	✅
String Encryption	✅	✅	Manual

Flutter-Specific Security

```
# Build with obfuscation
flutter build apk --obfuscate --split-debug-info=/<project-name>/symbols
flutter build ios --obfuscate --split-debug-info=/<project-name>/symbols
```

```
// Root/jailbreak detection
import 'package:flutter_jailbreak_detection/flutter_jailbreak_detection.dart';

Future<bool> isDeviceCompromised() async {
  final isJailbroken = await FlutterJailbreakDetection.jailbroken;
  final isDeveloperMode = await FlutterJailbreakDetection.developerMode;
  return isJailbroken || isDeveloperMode;
}

// Disable screenshots (sensitive screens)
import 'package:flutter_windowsecure/flutter_windowsecure.dart';

// In your sensitive screen
FlutterWindowSecure.setSecureScreen(true); // Android only
```


Important: Obfuscation is **not encryption**. It raises the bar but doesn't make reverse

MobSF — Mobile Security Framework

MobSF is an open-source, automated mobile app security testing framework.

What MobSF Does:

- **Static Analysis (SAST):** Analyzes APK/IPA without running them.
- **Dynamic Analysis (DAST):** Runs the app in an emulator and monitors behavior.
- **Malware Analysis:** Detects known malware patterns.
- **Web API Testing:** Tests connected APIs.

Why MobSF?

- **Free & open-source** (Apache 2.0 license).
- **Supports Android, iOS, and Windows** apps.
- **One-click analysis** — Upload APK/IPA, get a full report.
- **CI/CD integration** — Run as part of your build pipeline.
- **REST API** — Automate security testing.

Installation:

MobSF — Static Analysis

MobSF performs deep static analysis on your APK/IPA without executing the app.

What it checks:

Android (APK/AAB)

- Manifest analysis (permissions, exported components)
- Hardcoded secrets (API keys, passwords)
- Insecure cryptography (MD5, DES, ECB)
- Certificate analysis (signing, validity)
- Malware analysis (YARA rules)
- Binary analysis (NX, PIE, RPATH, RELRO)
- Trackers and libraries detection

iOS (IPA)

- Info.plist analysis
- Binary protection checks
- Insecure data storage
- Keychain usage analysis
- URL scheme validation
- Entitlements review
- ATS (App Transport Security) config

MobSF — Static Analysis Report

Key sections of a MobSF report:

1. **Security Score:** Overall risk rating (0–100).
2. **Permissions:** All requested permissions with risk levels.
3. **Findings:** Categorized as High / Warning / Info.
4. **Code Analysis:** Scans for insecure patterns in source code.
5. **Binary Analysis:** Checks binary hardening (PIE, NX, Stack Canary).
6. **Malware Analysis:** YARA rule matches.
7. **Trackers:** Detected analytics/tracking SDKs.
8. **Network Security:** TLS configuration, certificate analysis.

> **Typical Finding:** "App allows cleartext traffic" or "Hardcoded API key found in source code."

MobSF — Dynamic Analysis

MobSF runs your app in an Android emulator and monitors runtime behavior.

Dynamic Analysis Steps:

1. MobSF launches a rooted Android emulator.
2. Installs and launches your APK.
3. **Frida** hooks into the app for runtime instrumentation.
4. Monitors API calls, file I/O, network traffic, crypto operations.
5. Generates a comprehensive report.

What it detects:

- **Network traffic:** All HTTP/HTTPS requests and responses.
- **File system access:** Where the app reads/writes files.
- **Clipboard usage:** Does the app read the clipboard?
- **Crypto operations:** What algorithms are used?
- **Intent sniffing:** What intents does the app send/receive?

MobSF — Running an Analysis

```
# Step 1: Start MobSF
docker run -it -p 8000:8000 opensecurity/mobile-security-framework-mobsf

# Step 2: Open browser → http://localhost:8000

# Step 3: Upload your APK/IPA file

# Step 4: Review the Static Analysis report

# Step 5: For Dynamic Analysis:
#   - Start an Android emulator
#   - Configure MobSF to connect to the emulator
#   - Click "Start Dynamic Analysis"
#   - Interact with the app (or use automated exploration)
#   - Click "Generate Report"

# Step 6: Export report as PDF/JSON
```

CI/CD Integration: MobSF has a REST API. You can automate scans in GitHub Actions, GitLab CI, or Jenkins.

MobSF — REST API for Automation

```
# Upload APK for analysis
curl -F 'file=@app-release.apk' \
     http://localhost:8000/api/v1/upload \
     -H 'Authorization: <your-api-key>'

# Response: {"hash": "abc123", "scan_type": "apk", ...}

# Start static analysis
curl -X POST \
     http://localhost:8000/api/v1/scan \
     -H 'Authorization: <your-api-key>' \
     -d 'hash=abc123&scan_type=apk'

# Get report as JSON
curl http://localhost:8000/api/v1/report_json \
     -H 'Authorization: <your-api-key>' \
     -d 'hash=abc123'

# Get report as PDF
curl http://localhost:8000/api/v1/download_pdf \
     -H 'Authorization: <your-api-key>' \
     -d 'hash=abc123' -o report.pdf
```

Secure SDLC — Building Security In

Security is not a phase — it's a **continuous practice**.

Secure Development Lifecycle:

1. **Design:** Threat model before coding.
2. **Develop:** Follow secure coding guidelines.
3. **Build:** Obfuscate, strip debug symbols.
4. **Test:** Run MobSF + manual penetration testing.
5. **Release:** Sign builds, verify integrity.
6. **Monitor:** Crash reports, anomaly detection.

Flutter Security Checklist:

- [ ] Obfuscate release builds (`--obfuscate`)
- [ ] Store tokens in `flutter_secure_storage`
- [ ] Use HTTPS + certificate pinning
- [ ] Implement root/jailbreak detection

Mini Lab

Security Audit with MobSF

Objective: Perform a complete security audit of a Flutter app using MobSF.

Steps:

1. Build a Flutter app with intentional security flaws:
   - Hardcoded API key
   - Token stored in SharedPreferences
   - HTTP (not HTTPS) endpoint
   - No certificate pinning
2. Build the APK: `flutter build apk`
3. Start MobSF via Docker
4. Upload the APK for static analysis
5. Review findings and categorize by severity
6. Fix all issues and rebuild

Summary: Key Takeaways

Security Fundamentals

- OWASP Mobile Top 10 is your baseline.
- Threat model before you code.
- Protect data at rest AND in transit.
- Authentication must be bulletproof.

MobSF

- Free, open-source security scanner.
- Static + Dynamic analysis.
- One-click APK/IPA scanning.
- REST API for CI/CD integration.

Flutter Security

- Obfuscate release builds.
- Use `flutter_secure_storage` for tokens.
- Certificate pinning for network calls.
- Root/jailbreak detection.

Secure SDLC

- Security is continuous, not a phase.
- Automate scanning in CI/CD.
- Audit third-party dependencies.
- Keep a security checklist.

Thank You! 🙌

Questions?

"A security researcher walks into a bar.
The bartender says, 'Password?'
The researcher replies: 'admin123.'
The bartender: 'Welcome back, root.'"