Mobile Programming

Device Integrity & Security Detection

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



Root Detection | Emulator Detection | Tamper Detection | SMS OTP | Debugger

Table of Contents

1. Why Device Integrity Detection?
2. Root & Jailbreak Detection
3. Emulator Detection
4. Tamper Detection & App Integrity
5. Debugger Detection
6. SMS OTP Auto-Fill
7. Combining Detection Strategies
8. Mini Lab: Building a Secure Banking App
9. Summary & Key Takeaways

Why Device Integrity Detection?

Common Attack Scenarios:

- **Rooted/Jailbroken devices** — Full filesystem access, privilege escalation.
- **Emulators** — Automated testing, bots, fraud attacks.
- **Debuggers** — Step-through code execution, memory inspection.
- **Repackaged apps** — Malware injection into legitimate apps.
- **Screen capture** — Sensitive data exposed via screen recording.

Why it matters:

- Banking apps prevent operations on rooted devices.
- Gaming apps block emulators to prevent bots.
- Enterprise apps protect corporate data.
- Payment apps verify app integrity before transactions.

> **Key Principle:** If your app handles sensitive data or money, you MUST verify the environment it's running in.

Root & Jailbreak Detection

Root (Android) and **Jailbreak (iOS)** give users admin access to the OS.

What attackers can do with root/jailbreak:

- Access any file, including secure storage.
- Bypass permission checks.
- Install surveillance malware.
- Modify app behavior at runtime.
- Extract hardcoded secrets.

Root & Jailbreak Detection

Flutter packages for detection:

```
dependencies:
  flutter_jailbreak_detection: ^1.10.0
  root_check: ^1.0.0
```

Basic implementation:

```dart
import 'package:flutter_jailbreak_detection/flutter_jailbreak_detection.dart';

Future<bool> checkDeviceIntegrity() async {
  // Check if device is jailbroken/rooted
  final isJailbroken = await FlutterJailbreakDetection.jailbroken;

  // Check if developer mode is enabled
  final isDeveloperMode = await FlutterJailbreakDetection.developerMode;

  return isJailbroken || isDeveloperMode;
}
```

Root & Jailbreak Detection

Manual Android checks (platform channels):

```dart
// Check for su binary
final suBinary = await rootCheck.checkRoot();
// Check for custom ROM
final customRom = await rootCheck.checkRootCustom();
// Check for root management apps
final rootApps = await rootCheck.existsRootMagisk();
```

iOS jailbreak detection:

- Check for Cydia/Sileo app existence.
- Check for suspicious file paths (`/Applications/Cydia.app`).
- Check if `fork()` succeeds (should fail on non-jailbroken).
- Check for symbolic link anomalies.

> **Note:** No single check is foolproof. Use **multiple detection methods** for defense in depth.

Emulator Detection

Emulators are used for automated testing, bot attacks, and fraud.

Why attackers use emulators:

- **Account creation fraud** — Create thousands of fake accounts.
- **Bonus abuse** — Exploit new account bonuses automatically.
- **Gaming bots** — Automated gameplay for resource farming.
- **Reverse engineering** — Run app in controlled environment.

Emulator Detection

Flutter package: `flutter_emulator_detection`

```
dependencies:
  flutter_emulator_detection: ^1.0.0
```

Implementation:

```dart
import 'package:flutter_emulator_detection/flutter_emulator_detection.dart';

Future<bool> isRunningOnEmulator() async {
  final isEmulator = await FlutterEmulatorDetection.isEmulator;
  return isEmulator;
}

// Combined check
Future<Map<String, bool>> securityCheck() async {
  return {
    'isEmulator': await FlutterEmulatorDetection.isEmulator,
    'isJailbroken': await FlutterJailbreakDetection.jailbroken,
    'isDeveloperMode': await FlutterJailbreakDetection.developerMode,
  };
}
```

Emulator Detection

Manual checks for Android:

```dart
// Platform channel to native code
static const platform = MethodChannel('com.example/app/security');

// Check for emulator-specific indicators
Future<bool> checkEmulator() async {
  final result = await platform.invokeMethod('checkEmulator');
  return result['isEmulator'] ?? false;
}
```

Android native detection:

- Check `Build.PRODUCT` contains "emulator" or "generic".
- Check `Build.HARDWARE` contains "goldfish" or "ranchu".
- Check `ro.bootloader` contains "unknown".
- Check for Google Mapsroid (emulator-specific).
- Check for `telephony.isFromExternalCarriers()`.

Tamper Detection & App Integrity

Tampering = Modifying app code, resources, or signature to bypass security.

Types of tampering:

- **Repackaging** — Decompile, modify, republish with different signature.
- **Code injection** — Insert malicious code via hooks (Frida, Xposed).
- **Resource modification** — Change images, strings, layouts.
- **Signature bypass** — Modify signature verification logic.

Tamper Detection & App Integrity

App signature verification:

```dart
import 'package:app_signature/app_signature.dart';

// Get app signing certificate signature
Future<String?> getAppSignature() async {
  final signatures = await AppSignature.getAppSignatures();
  return signatures.isNotEmpty ? signatures.first : null;
}

// Verify against expected signature
Future<bool> verifyAppIntegrity() async {
  final expectedHash = 'YOUR_EXPECTED_SIGNATURE_HASH';
  final currentHash = await getAppSignature();
  return currentHash == expectedHash;
}
```

> **Important:** Store the expected hash server-side, not in the app!

Tamper Detection — Flutter Packages

Available packages:

```yaml
dependencies:
  app_signature: ^4.0.0          # App signature verification
  flutter_secure_storage: ^9.0.0 # Secure storage for integrity checks
  screenshot_callback: ^1.0.0    # Detect screenshots
```

Screenshot detection (Android):

```dart
import 'package:screenshot_callback/screenshot_callback.dart';

final screenshotCallback = ScreenshotCallback();
screenshotCallback.addListener(() {
  print('Screenshot detected!');
  // Alert security team or log incident
});
screenshotCallback.register();
```

Debugger Detection

Debuggers allow attackers to step through code and inspect memory.

Common debuggers:

- **Frida** — Runtime instrumentation, function hooking.
- **Xposed** (Android) — Module-based code modification.
- **LLDB** (iOS) — Native debugging.
- **gdb** — Command-line debugger.

```dart
import 'dart:io';

Future<bool> isDebuggerAttached() async {
  if (Platform.isAndroid) {
    // Check for tracer PID in /proc/self/status
    final file = File('/proc/self/status');
    final content = await file.readAsString();
    return content.contains('TracerPid:');
  }
  return false;
}
```

Debugger Detection

Frida detection packages:

```
dependencies:
  flutter_frida_gadget: ^1.0.0  # Detect Frida injection
```

Basic Frida detection:

```
import 'package:flutter_frida_gadget/flutter_frida_gadget.dart';

Future<bool> detectFrida() async {
  final fridaDetected = await FlutterFridaGadget.isFridaRunning();
  return fridaDetected;
}
```

Manual Frida checks:

- Check for open ports commonly used by Frida (27042, 27043).
- Check for `frida-server` in process list.
- Check for shared library files (`frida-agent.so`).

SMS OTP Auto-Fill

SMS OTP auto-fill improves UX while maintaining security.

How it works:

1. SMS arrives with a one-time password (OTP).
2. App receives SMS via `sms_autofill` package.
3. OS parses the OTP from the message.
4. App prefills the verification input field.

Important: SMS autofill is an OS feature, not a security vulnerability — the OS handles consent.

SMS OTP Auto-Fill

Flutter package: `sms_autofill`

```
dependencies:
  sms_autofill: ^2.4.0
```

SMS OTP Auto-Fill

```dart
import 'package:sms_autofill/sms_autofill.dart';
class OtpScreen extends StatefulWidget {
  @override
  _OtpScreenState createState() => _OtpScreenState();
}
class _OtpScreenState extends State<OtpScreen> {
  final TextEditingController _codeController = TextEditingController();
  @override
  void initState() {
    super.initState();
    _listenForSms();
  }
  Future<void> _listenForSms() async {
    await SmsAutoFill().listenForCode();
  }
  @override
  void dispose() {
    SmsAutoFill().unregisterListener();
    _codeController.dispose();
    super.dispose();
  }
```

SMS OTP Auto-Fill (Cont.)

```
  @override
  Widget build(BuildContext context) {
    return TextField(
      controller: _codeController,
      keyboardType: TextInputType.number,
      maxLength: 6,
      decoration: InputDecoration(
        labelText: 'Enter OTP',
        hintText: 'Auto-filled from SMS',
        counterText: '',
      ),
      autovalidateMode: AutovalidateMode.onUserInteraction,
      validator: (value) {
        if (value?.length != 6) {
          return 'OTP must be 6 digits';
        }
        return null;
      },
    );
  }
}
```

SMS OTP Auto-Fill

App signature for SMS filtering (Android):

```
<!-- AndroidManifest.xml -->
<application>
  <!-- Required for SMS autofill -->
  <uses-permission android:name="android.permission.RECEIVE_SMS"/>

  <!-- For Android 8.0+ -->
  <receiver android:name="com.google.android.gms.auth.api.phone.SmsRetrieverReceiver"
            android:exported="true">
    <intent-filter>
      <action android:name="com.google.android.gms.auth.api.phone.SMS_RETRIEVED_ACTION"/>
    </intent-filter>
  </receiver>
</application>
```

iOS Info.plist:

```
<key>NSPhoneNumberUsageDescription</key>
<string>We need SMS access to auto-fill your verification code</string>
```

Combining Detection Strategies

No single check is 100% reliable. Use defense in depth.

```
class SecurityChecker {
  Future<SecurityReport> runAllChecks() async {
    return SecurityReport(
      isRooted: await _checkRoot(),
      isEmulator: await _checkEmulator(),
      isDebugger: await _checkDebugger(),
      isTampered: await _checkTamper(),
      fridaDetected: await _checkFrida(),
    );
  }

  Future<bool> isDeviceSecure() async {
    final report = await runAllChecks();
    return !report.isCompromised;
  }
}
```

Combining Detection Strategies

Security report model:

```
class SecurityReport {
  final bool isRooted;
  final bool isEmulator;
  final bool isDebugger;
  final bool isTampered;
  final bool fridaDetected;

  bool get isCompromised =>
    isRooted || isEmulator || isDebugger || isTampered || fridaDetected;

  List<String> get issues {
    final List<String> problems = [];
    if (isRooted) problems.add('Device is rooted/jailbroken');
    if (isEmulator) problems.add('Running on emulator');
    if (isDebugger) problems.add('Debugger detected');
    if (isTampered) problems.add('App integrity compromised');
    if (fridaDetected) problems.add('Frida injection detected');
    return problems;
  }
}
```

Combining Detection Strategies

```
// In your app's main or security service
Future<void> checkAndRespond() async {
  final report = await SecurityChecker().runAllChecks();
  if (report.isCompromised) {
    showDialog(
      context: context,
      barrierDismissible: false,
      builder: (ctx) => AlertDialog(
        title: Text('Security Warning'),
        content: Text(
          'This device may be compromised.\n\n'
          'Issues detected:\n${report.issues.map((i) => '• $i').join('\n')}',
        ),
        actions: [
          TextButton(
            onPressed: () => exit(0),
            child: Text('Exit'),
          ),
        ],
      ),
    );
  }
}
```

Mini Lab

Building a Secure Banking App

Objective: Build a Flutter banking app with device integrity checks and SMS OTP.

1. **Device integrity check** on app launch — block rooted/emulator devices.
2. **App signature verification** — ensure app hasn't been tampered with.
3. **SMS OTP auto-fill** — implement login verification with auto-filled OTP.
4. **Combined security report** — display all detected issues.
5. **Biometric authentication** — use `local_auth` for additional security.
6. **Secure storage** — store auth tokens in `flutter_secure_storage`.

Bonus Challenges:

- Add screenshot detection and blocking
- Implement server-side signature verification

Summary: Key Takeaways

Detection Types

- **Root/Jailbreak** `flutter_jailbreak_detec`
- **Emulator** — `flutter_emulator_detection`
- **Tamper** — App signature verification
- **Debugger** — TracerPid and Frida detection
- **SMS OTP** — `sms_autofill` package

Defense in Depth

- No single check is 100% reliable.
- Combine multiple detection methods.
- Store expected hashes server-side.
- Always verify app integrity.

Key Packages

- `flutter_jailbreak_detection` — Root/jailbreak detection
- `flutter_emulator_detection` — Emulator detection
- `app_signature` — Signature verification
- `sms_autofill` — SMS OTP auto-fill
- `flutter_secure_storage` — Secure token storage
- `local_auth` — Biometric authentication

Best Practices

- Run security checks on app launch.
- Show clear warnings to users.

Thank You! 🙌

Questions?

"A hacker walked into a bar,
but the bartender said 'Root access denied.'
So the hacker rooted the bartender,
and the bartender said 'Welcome, admin.'"