Mastering Mobile Security, Reverse Engineering & Malware Analysis

Alper Basaran

Alper Basaran

Penetration Tester / Cybersecurity Consultant

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

Key Concepts

- Mobile application reverse engineering
- Security architecture and runtime behavior
- Malware analysis and penetration testing
- Secure development practices
- Future trends in mobile security

MAJOR TOPICS

- Reverse engineering tools and techniques
- Advanced security bypass strategies
- Obfuscation and de-obfuscation tactics
- Security in hybrid and WebView-based apps
- Malware analysis on mobile and embedded systems

LEARNING OUTCOMES

+ Understand mobile application architecture and runtime behavior
+ Analyze and reverse-engineer mobile apps and malware
+ Perform penetration testing on diverse mobile platforms
+ Implement secure mobile development and DevSecOps strategies

- Basic understanding of programming and cybersecurity
- Familiarity with mobile operating systems

LET'S GO!

Introduction to Mobile Application Reverse Engineering

Why Penetration Testers Need Reverse Engineering in Mobile App Testing?

- Limitations of black-box testing for mobile apps
- When reverse engineering becomes necessary:
  - Analyzing custom cryptographic functions
  - Bypassing security controls like SSL pinning or root detection
  - Extracting hardcoded secrets, API endpoints, or certificates
  - Understanding logic in obfuscated code

What is Reverse Engineering?

- Reverse engineering ≠ just decompiling code
- Reconstructing the app's internal logic and data handling
- Static vs dynamic reverse engineering

Core Principles of Reverse Engineering

- Abstraction levels:
  - source code → bytecode → assembly → machine code
- Code transformation: converting binary artifacts to readable formats
- Control flow & data flow analysis
- Static Analysis: structure, functions, strings, classes
- Dynamic Analysis: runtime hooks, logging, behavior monitoring

Reverse Engineering Methodology

- Step 1: **Recon** – Collect the APK/IPA, metadata, app behavior
- Step 2: **Unpacking/Unzipping** – Explore assets, manifests, certificates
- Step 3: **Static Analysis** – Decompile, analyze code, search strings/classes
- Step 4: **Dynamic Analysis** – Attach debugger/Frida, observe runtime
- Step 5: **Patching & Modification** – Change app logic or bypass protections
- Step 6: **Rebuild & Re-sign** – Run the modified version

Reverse Engineering Methodology: Step 1: Recon

- **Version Information:** Understanding application versioning and updates
- **Developer Information:** Identifying development team and company details
- **Platform Requirements:** Understanding target platforms and dependencies
- **Permission Analysis:** Examining requested system permissions

Reverse Engineering Methodology: Step 2: Unpacking/Unzipping

+ “Unzip or something”

Reverse Engineering Methodology: Step 3: Static Analysis

- **Package Organization:** Understanding code organization and architecture
- **Class Hierarchy Analysis:** Mapping inheritance and composition relationships
- **Method Signature Analysis:** Understanding API usage and integration points
- **String and Resource Analysis:** Examining embedded data and configuration

Reverse Engineering Methodology: Step 4: Dynamic Analysis

- **Testing Device Configuration:** Setting up analysis-ready mobile devices
- **Monitoring Tool Installation:** Deploying runtime analysis tools
- **Network Monitoring Setup:** Configuring traffic interception and analysis
- **Logging and Documentation:** Establishing comprehensive activity recording

Prerequisites for Effective Reverse Engineering

+ Understanding of mobile app architecture (Android/iOS)
+ Familiarity with Java/Kotlin, Swift/Obj-C, and native code (C/C++)
+ Comfort with static & dynamic analysis tools

Your Reverse Engineering Arsenal

The Reverse Engineering Methodology and Tool Selection

- The Four Phases of Mobile App Reverse Engineering:
  - **Reconnaissance Phase:** Initial information gathering
  - **Static Analysis Phase:** Analyzing code without execution
  - **Dynamic Analysis Phase:** Analyzing runtime behavior
  - **Exploitation/Validation Phase:** Testing findings and vulnerabilities

Factors Influencing Tool Selection

+ Analysis Objectives
+ Target Application Characteristics
+ Available Resources

The Tool Selection Decision Tree

- Analysis Objective: Security Assessment
  - Quick Scan → Automated Tools (MobSF, QARK)
  - Deep Analysis → Manual Tools (Jadx, Ghidra) + Dynamic Tools (Frida)
  - Compliance Check → Specialized Scanners (Checkmarx, Veracode)

The Tool Selection Decision Tree

- Analysis Objective: Malware Analysis
  - Behavior Analysis → Sandbox Tools (Cuckoo, Joe Sandbox)
  - Code Analysis → Disassemblers (Jadx, Ghidra) + Hex Editors
  - Network Analysis → Wireshark + mitmproxy

Building a Systematic Workflow

- Phase 1: Initial Reconnaissance
- **Goal: Gathering basic information**
  - file target_app.apk
  - aapt dump badging target_app.apk

Building a Systematic Workflow

- Phase 2: Static Analysis
- **Goal: Decompile**
  - jadx -d output_jadx target_app.apk
  - apktool d target_app.apk -o output_apktool

Building a Systematic Workflow

- Phase 3: Dynamic Analysis
- **Goal: Monitor the application**
  - adb install target_app.apk
  - frida -U -f com.target.app -l analysis_script.js

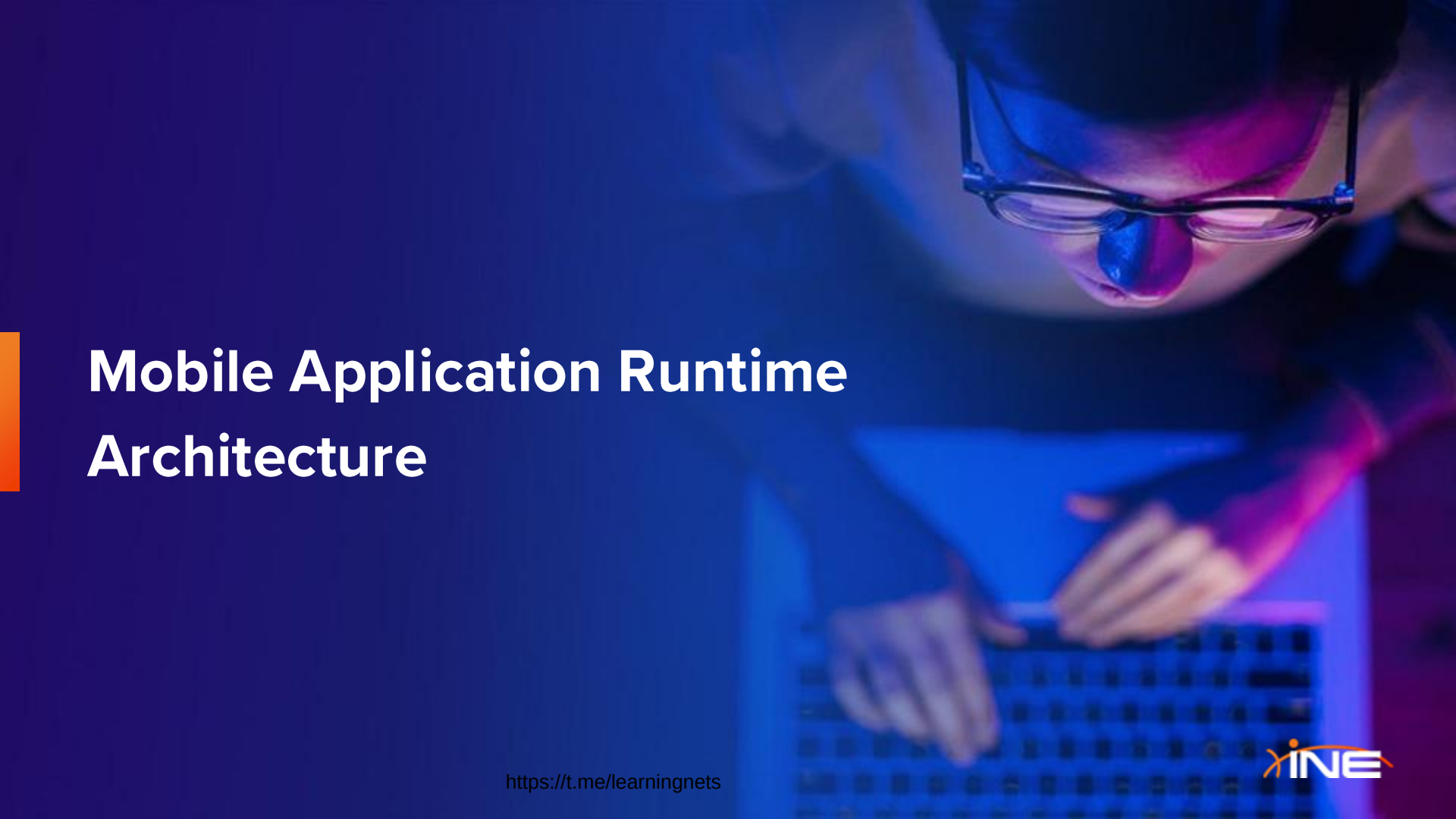
Building a Systematic Workflow

- Phase 4: Deep Analysis
- **Goal: See more**
  - Androguard
  - Ghidra?

Key Takeaways

- Select tools according to the engagement
- Use tools “dynamically” (e.g. Zipalign)
- Know what output to expect && why
- Casually try new tool or use AI to write your own

Mobile Application Runtime Architecture



Android Runtime

- DEX Structure
  - **header** contains magic numbers, checksums, and file size
  - **string pool** holds all string literals used in the application
  - **Type definitions** map out all the classes and primitive types
  - **Method definitions** contain the actual bytecode instructions (app logic)

DEX File Format

- Starting with Android 5.0, Google introduced ART
- When you install an app on an ART device, the DEX bytecode is compiled to native code immediately.
  - Static analysis becomes more effective
  - Runtime attacks become harder (there's no JIT compiler to exploit)

Smali

- Assembly language for the Dalvik virtual machine
- For security analysis, pay attention to:
  - **invoke-** instructions for method calls, especially crypto functions
  - **const-string** for hardcoded values
  - **if-** instructions for validation logic
  - **return-** instructions for control flow manipulation

JIT vs AOT Security Implications

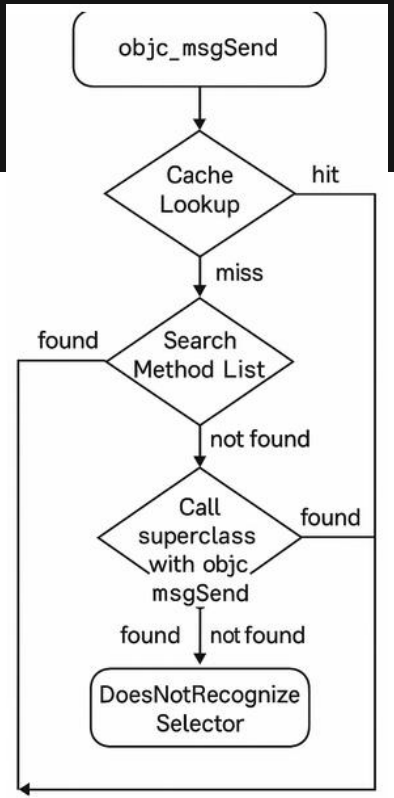
- JIT Compilation attack vectors
  - **Code cache corruption** - attackers could corrupt the compiled code cache
  - **JIT spray attacks** - predictable code generation could be exploited for ROP gadgets
  - **Dynamic code modification** - the runtime nature allowed for code patching attacks

JIT vs AOT Security Implications

- AOT Compilation
  - **Reduced runtime complexity** means fewer moving parts to exploit
  - **Predictable memory layouts** can help with exploit development
  - **Static analysis effectiveness** means defenders can find vulnerabilities easier too

iOS Runtime Architecture

- Everything in Objective-C happens through message passing:
  - objc_msgSend function



Security Implications

- **Method swizzling** lets you replace method implementations
- **ISA pointer corruption** can redirect method calls
- **Category injection** allows code injection at runtime
- **KVO/KVC bypasses** can access private properties

Swift Runtime Components

- Application Binary Interface (ABI)
- Makes some attacks harder:
  - **Type safety** prevents many memory corruption issues
  - **Automatic Reference Counting** reduces use-after-free vulnerabilities
  - **Value semantics** limit shared mutable state

Swift Runtime Components

- Application Binary Interface (ABI)
- Creates new opportunities:
  - Protocol witness table manipulation
  - Generic type confusion attacks
  - Metadata corruption for type confusion

Mach-O File Format

- Mach-O is macOS and iOS's executable format
- A Mach-O file consists of:
  - **Header** with architecture and file type information
  - **Load commands** that tell the loader what to do
  - **Segments** that contain the actual data
  - **Sections** within segments for specific data types

Cross-Platform Considerations
Xamarin / Flutter / Dart

- Common Hybrid Patterns
  - **WebView-based applications** like Cordova or PhoneGap
  - **Native wrappers** around web content
  - **Progressive Web Apps** with native shells
  - **Micro-frontend architectures** with multiple technologies

Cross-Platform Considerations
Xamarin / Flutter / Dart

- For hybrid applications, use a layered approach:
  - **Web Layer Assessment** - Analyze JavaScript, HTML, and CSS
  - **Bridge Security Review** - Examine native-web communication
  - **Native Component Audit** - Traditional native code analysis
  - **Data Flow Analysis** - Track data across technology boundaries
  - **Attack Surface Mapping** - Consider all integration points

Cross-Platform Considerations
Xamarin / Flutter / Dart

- Obfuscation Effectiveness Comparison:
  - **Native Android/iOS:** Code obfuscation requires specialized tools, relatively effective
  - **React Native:** JavaScript minification only, easily reversible
  - **Flutter:** Dart compilation provides moderate protection
  - **Hybrid Web:** Minimal protection, standard web security applies

Core Methodology Framework:
Phase 1: Environmental Reconnaissance

- Runtime environment mapping (OS version, security patches, hardware capabilities)
- Application sandboxing analysis and permission boundaries
- Network connectivity patterns and communication channels
- Dependency mapping (shared libraries, frameworks, third-party SDKs)

Core Methodology Framework: Phase 2: Behavioral Baseline Establishment

+ Normal application flow documentation
+ Resource utilization patterns (CPU, memory, storage, network)
+ Inter-process communication identification
+ Background service behavior analysis

Core Methodology Framework: Phase 3: Systematic Perturbation Testing

+ Input validation boundary testing across all entry points
+ State manipulation through controlled runtime modifications
+ Resource exhaustion and race condition testing
+ Authentication and session management stress testing

Simulating Advanced Attacker Scenarios

- Threat Actor Modeling
  - Opportunistic Attackers
  - Sophisticated Threat Actors
  - Insider Threats

Scenario-Based Testing Methodology

- Structure your approach around realistic attack narratives
  - Initial Access Simulation
  - Persistence Mechanisms
  - Lateral Movement
  - Data Exfiltration
  - Evidence Elimination

Memory Dump Analysis and Runtime Behavior

- Memory Analysis Methodology
  - **Heap Analysis**: Understanding object lifecycles and memory management
  - **Stack Analysis**: Function call patterns and local variable inspection
  - **Static Memory Regions**: Global variables and constant data examination
  - **Dynamic Memory Allocation**: Real-time memory usage pattern analysis

Memory Dump Analysis and Runtime Behavior

- Runtime Behavior Correlation
  - Linking memory state changes to user interactions
  - Identifying sensitive data storage patterns
  - Detecting encryption/decryption operations through memory analysis

Penetrating Encrypted Apps and Obfuscated Code

- Encryption Analysis Framework
  - **Encryption Identification**: Recognizing encryption algorithms and implementations
  - **Key Management Analysis**: Understanding key storage and derivation methods
  - **Runtime Key Extraction**: Techniques for obtaining encryption keys during execution
  - **Plaintext Recovery**: Methods for accessing decrypted data in memory

Smali Language

Smali Language

+ Human-readable representation of Android's Dalvik bytecode
+ Bridge between Java source code and DEX bytecode
+ Essential for Android reverse engineering and malware analysis

Smali Syntax and Instruction Set Architecture

- Register types and naming conventions
  - v0, v1, v2: Local registers
  - p0, p1, p2: Parameter registers
  - this : Reference to current object

Smali Syntax and Instruction Set Architecture

- Basic Instruction Categories
  - move v0, v1: Move register v1 to v0
  - move-object v0, v1 : Move object reference
  - move-result v0 : Move method return value
  - add-int v0, v1, v2 : v0 = v1 + v2
  - mul-int/lit8 v0, v1, 0x5 : v0 = v1 * 5

Smali Syntax and Instruction Set Architecture

- Method invocation
  - invoke-virtual {v0, v1}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z

Smali Syntax and Instruction Set Architecture

```
.method private validatePassword(Ljava/lang/String;)Z
    .locals 3

    # Get the correct password
    const-string v0, "admin123"

    # Compare with user input
    invoke-virtual {v0, p1}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z
    move-result v1

    # Return the result
    return v1
.end method
```

Control Flow Analysis in Smali Code

- Conditional Branches:
  - if-eq v0, v1, :cond_equal => (v0 == v1) goto :cond_equal
  - if-ne v0, v1, :cond_not_equal => if (v0 != v1) goto :cond_not_equal
  - if-lt v0, v1, :cond_less => if (v0 < v1) goto :cond_less
  - if-eqz v0, :cond_zero => if (v0 == 0) goto :cond_zero

Control Flow Analysis in Smali Code

+ Loop Structures :

```
const/4 v0, 0x0          # i = 0
goto :loop_condition

:loop_start
    # Loop body
    add-int/lit8 v0, v0, 0x1    # i++

:loop_condition
    const/16 v1, 0xa            # limit = 10
    if-lt v0, v1, :loop_start   # if (i < 10) continue loop
```

Bypassing Advanced Security Mechanisms

Understanding Detection Layers

- Common Android Root Detection Methods
  - **File-based checks**: /system/app/Superuser.apk, /system/bin/su
  - **Property checks**: ro.build.tags, ro.debuggable
  - **Package manager**: Detecting root management apps
- iOS Jailbreak Detection Techniques
  - **File system checks**: Cydia, unauthorized system modifications
  - **Sandbox violations**: Writing to restricted directories
  - **URL scheme checks**: cydia:// availability

Systematic Bypass Approach

- Common Android Root Detection Methods
  - **File-based checks**: /system/app/Superuser.apk, /system/bin/su
  - **Property checks**: ro.build.tags, ro.debuggable
  - **Package manager**: Detecting root management apps
- iOS Jailbreak Detection Techniques
  - **File system checks**: Cydia, unauthorized system modifications
  - **Sandbox violations**: Writing to restricted directories
  - **URL scheme checks**: cydia:// availability

Understanding Detection Layer Architecture

- Layer 1: Environmental Indicators
- Layer 2: Process and Service Detection
- Layer 3: System Property Analysis

Understanding Detection Layer Architecture
Layer 1: Environmental Indicators

- Examining the runtime environment for indicators of privilege escalation:
  - /system/app/Superuser.apk
  - /system/bin/su
  - /system/xbin/su
  - /data/local/xbin/su
  - /data/local/bin/su
  - /system/etc/init.d/99SuperSUDaemon
  - /dev/com.koushikdutta.superuser.daemon/

Understanding Detection Layer Architecture
Layer 2: Process and Service Detection

+ Monitors running processes and system services for root management tools:

```
public boolean checkRunningProcesses() {
    String[] rootProcesses = {
        "su", "supersu", "daemonsu", "magisk", "magiskhide"
    };
```

Understanding Detection Layer Architecture
Layer 3: System Property Analysis

- Examines build properties and system characteristics:
  - ro.debuggable=1
  - ro.secure=0
  - service.adb.root=1
  - ro.build.tags=test-keys

Systematic Bypass Framework

- Phase 1: Detection Discovery
- Phase 2: Selective Bypass Implementation

```
Java.perform(function() {
    var File = Java.use("java.io.File");
    File.exists.implementation = function() {
        var path = this.getAbsolutePath();
        var rootPaths = ["/system/bin/su", "/system/xbin/su", "/data/local/xbin/su"];

        if (rootPaths.some(rootPath => path.includes(rootPath))) {
            console.log("[+] Hiding root file: " + path);
            return false;
        }
        return this.exists();
```

Emulator/Simulator Detection Bypass

+ Applications detect emulated environments to prevent analysis in controlled settings.

```
public boolean isEmulator() {
    return Build.FINGERPRINT.startsWith("generic")
        || Build.FINGERPRINT.startsWith("unknown")
        || Build.MODEL.contains("google_sdk")
        || Build.MODEL.contains("Emulator")
        || Build.BOARD.equals("QQQ1.200405.005")
        || Build.MANUFACTURER.contains("Genymotion");
}
```

Emulator/Simulator Detection Bypass

- Applications detect emulated environments to prevent analysis in controlled settings.

```
TelephonyManager tm = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
String networkOperator = tm.getNetworkOperatorName();
if (networkOperator.equals("Android") || networkOperator.equals("")) {
}
```

Emulator/Simulator Detection Bypass

- Applications detect emulated environments to prevent analysis in controlled settings.

```
SensorManager sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
List<Sensor> sensors = sensorManager.getSensorList(Sensor.TYPE_ALL);
if (sensors.size() < 7) {
```

Universal Android Bypass Framework

- You may try:

```
Java.perform(function() {
    var Build = Java.use("android.os.Build");
    Build.FINGERPRINT.value = "google/redfin/redfin:11/RQ3A.210805.001.A1/7474174:user/release-
keys";
    Build.MODEL.value = "Pixel 5";
    Build.MANUFACTURER.value = "Google";
```

Universal Android Bypass Framework

+ Or:

```
var TelephonyManager = Java.use("android.telephony.TelephonyManager");
 TelephonyManager.getNetworkOperatorName.implementation = function() {
     console.log("[+] Spoofing network operator name");
     return "Verizon"; // or other realistic carrier
 };
```

Universal Android Bypass Framework

- Or even:

```
setprop ro.product.manufacturer "samsung"
setprop ro.product.model "SM-G973F"
setprop ro.build.fingerprint "samsung/beyond1ltexx/beyond1:10/QP1A.190711.020/
G973FXXU3ASL2:user/release-keys"
```

Bypassing Biometrics, CAPTCHA

+ Android Biometric Architecture

```
BiometricPrompt biometricPrompt = new BiometricPrompt(this, executor,
    new BiometricPrompt.AuthenticationCallback() {
        @Override
        public void onAuthenticationSucceeded(AuthenticationResult result) {
        }

        @Override
        public void onAuthenticationFailed() {
        }
    });
```


Bypassing Biometrics, CAPTCHA

+ Android Biometric Architecture

```
var BiometricManager = Java.use("androidx.biometric.BiometricManager");
BiometricManager.canAuthenticate.implementation = function() {
    console.log("[+] Spoofing biometric availability");
    return 0;
    };
```

Code Obfuscation and De-obfuscation

Understanding the Obfuscation Landscape

- Developers obfuscate code to:
  - intellectual property protection
  - Anti-piracy measures
  - Malware authors use it to evade detection

Understanding the Obfuscation Landscape

- String Obfuscation Techniques
  - XOR Encryption
  - Base64 Encoding
  - String Splitting and Concatenation

String Encryption Obfuscated

- eXclusive OR
- What's going on here?
- Plaintext (A): 01000001
- Key: 01001010
- Encrypted 00001011

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Understanding the Obfuscation Landscape

- Control Flow Obfuscation
  - Opaque Predicates
  - Dead Code Insertion
  - Exception-Based Control Flow
  - Reflection-Based Obfuscation

Understanding the Obfuscation Landscape

- Control Flow Obfuscation
  - Opaque Predicates

```
if (x * x >= 0) {
    Do This
} else {
    Do That
}
```

Understanding the Obfuscation Landscape

- Control Flow Obfuscation
  - Reflection-Based Obfuscation

```
myObject.sensitiveMethod();
```

```
Class<?> clazz = Class.forName("com.example.MyClass");
Method method = clazz.getDeclaredMethod("sensitiveMethod");
method.invoke(myObject);
```

Understanding the Obfuscation Landscape

- Control Flow Obfuscation
  - Reflection-Based Obfuscation

```
httpClient.sendUserData(userData);
```

Understanding the Obfuscation Landscape

- Control Flow Obfuscation
  - Reflection-Based Obfuscation

```
String className = decrypt("aHR0cENsaWVudA==");
String methodName = decrypt("c2VuZFVzZXJEYXRh");
Class<?> clazz = httpClient.getClass();
Method method = clazz.getDeclaredMethod(methodName, String.class);
method.invoke(httpClient, userData);
```

Understanding the Obfuscation Landscape

- Control Flow Obfuscation
  - Reflection-Based Obfuscation
  - Look for:
    - `Class.forName()`
    - `getDeclaredMethod()`
    - `method.invoke()`

Understanding the Obfuscation Landscape

```
Java.perform(function() {
    var Class = Java.use("java.lang.Class");
    Class.forName.overload('java.lang.String').implementation = function(className) {
        console.log("[+] Class.forName called with: " + className);
        return this.forName(className);
    };
```

- Look for:
  - `Class.forName()`
  - `getDeclaredMethod()`
  - `method.invoke()`

Understanding the Obfuscation Landscape

```
var Method = Java.use("java.lang.reflect.Method");
    Method.invoke.overload('java.lang.Object', '[Ljava.lang.Object;').implementation =
function(obj, args) {
        console.log("[+] Method.invoke called on: " + this.getName());
        console.log("[+] Arguments: " + args);
        var result = this.invoke(obj, args);
        console.log("[+] Result: " + result);
        return result;
```

- Look for:
  - `Class.forName()`
  - `getDeclaredMethod()`
  - `method.invoke()`

Understanding the Obfuscation Landscape

- Advanced Techniques
  - Packing and Compression
  - Symbol Stripping (`validatePassword` => `func_234`)

De-obfuscation Strategic Approaches
Static Analysis

- Pattern Recognition:
- Cross-referencing
- Entropy Analysis
- Data Flow Analysis

De-obfuscation Strategic Approaches
Dynamic Analysis

+ Runtime montoring
+ Memory dumping

Obfuscation Classification Framework By Implementation Methodology

- **Static Transformation**: Applied during compilation/packaging
- **Dynamic Protection**: Applied during runtime execution
- **Hybrid Approaches**: Combination of static and dynamic techniques

String Encryption
Plaintext

```
String apiKey = "sk_live_4eoSlaDyP7B9TGWtjLdZBz";
String serverUrl = "https://api.payment-processor.com/v1/";
```

String Encryption Obfuscated

```
public class StringDecryptor {
    private static final byte[] KEY = {0x4A, 0x7E, 0x2C, 0x8F, 0x1B, 0x9D, 0x3A, 0x6C};

    public static String decrypt(String encrypted) {
        byte[] data = Base64.decode(encrypted, Base64.DEFAULT);
        for (int i = 0; i < data.length; i++) {
            data[i] ^= KEY[i % KEY.length];
        }
        return new String(data);
    }
```

String Encryption
Common Vulnerabilities

+ Key reuse with known plaintext

+ Key reuse with k

Source: imdb

String Encryption
Common Vulnerabilities

- Key reuse with known plaintext

Source:https://www.networkpages.nl/enigma-a-complexity-titan/

String Encryption
Common Vulnerabilities

- Key reuse with known plaintext
  - Known plaintext = "https://"
  - Encrypted data = "AQU6AuLW"

String Encryption
Common Vulnerabilities

- Statistical analysis for repeating key:
  - XOR with repeating key preserves some statistical properties
  - Identical plaintext bytes at positions i and i+key length will produce identical ciphertext bytes

String Encryption
Common Vulnerabilities

- Detecting key length
  - Try different key lengths and look for patterns
  - Higher coincidence rate suggests correct key length

String Encryption Implementation Vulnerabilities

- Hardcoded key in source code
- No key derivation
- No integrity verification (Decryption always "succeeds" even with wrong key)

String Encryption
Automated Key Recovery

+ Statistical frequency analysis

English Letter Frequency (based on a sample of 40,000 words)

Letter	Count		Letter	Frequency
E	21912		E	12.02
T	16587		T	9.10
A	14810		A	8.12
O	14003		O	7.68
I	13318		I	7.31
N	12666		N	6.95
S	11450		S	6.28

Source: https://pi.math.cornell.edu/~mec/2003-2004/cryptography/subs/frequencies.html

String Encryption
Automated Key Recovery

- Dictionary attack with common words
  - http, https, api, key, secret, token'
  - Test if this key works for the entire string

String Encryption
Automated Key Recovery

+ Brute force for short keys

String Encryption
Dynamic Analysis Approach (Frida)

```
Java.perform(function() {
    var StringDecryptor = Java.use("com.example.StringDecryptor");

    StringDecryptor.decrypt.implementation = function(encrypted) {
        console.log("[+] String decryption called");
        console.log("    Input: " + encrypted);

        var result = this.decrypt(encrypted);

        console.log("    Output: " + result);
        console.log("    Length: " + result.length);

        if (result.includes("http") || result.includes("key") || result.includes("secret")) {
            console.log("[!] SENSITIVE DATA DECRYPTED: " + result);
        }

        return result;
    };
```

Obfuscation Challenges

- **Multi-Stage Decryption**: Multiple encryption layers requiring sequential decryption
- **Context-Dependent Keys**: Decryption keys derived from runtime environment
- **Split String Assembly**: Strings split across multiple variables and assembled at runtime

Manual De-obfuscation Methodologies

+ Phase 1: Reconnaissance and Pattern Analysis
+ Phase 2: String De-obfuscation Strategies
+ Phase 3: Control Flow Reconstruction
+ Phase 4: Reflection De-obfuscation

Manual De-obfuscation Methodologies

- Phase 1: Reconnaissance and Pattern Analysis
  - Base64: Regex: [A-Za-z0-9+/]{20,}={0,2}
  - Hex encoded: Regex [0-9a-fA-F]{16,}
  - XOR patterns (XOR, xor, etc.), crypto imports(javax, crypto, security, etc.), Decode (decode, decrypt, deobfuscate, etc.)

Manual De-obfuscation Methodologies

- Phase 2: String De-obfuscation Strategies
  - XOR Decryption Implementation
  - Multi-Stage Decryption Chain

Manual De-obfuscation Methodologies

- Phase 3: Control Flow Reconstruction
- Biggest challenge: Opaque Predicate Elimination
  - Remove always-true/always-false conditions

```
public class NestedOpaquePredicates {

    public void complexNestedLogic(String input) {

        boolean level1_true = input.length() >= 0;
        boolean level1_false = input.hashCode() == (input.hashCode() + 1);

        if (level1_true) {
            int hash = input.hashCode();
            boolean level2_true = (hash * hash + hash) % 2 == 0;
            boolean level2_false = (hash * (hash + 1)) % 2 == 1;

            if (level2_true && !level2_false) {
                long timestamp = System.nanoTime();
                boolean level3_true = timestamp > 0;

                if (level3_true) {
                    processInput(input);
                }

                if (level2_false) {
                    maliciousFunction();
                }
            }

            if (level1_false) {
                anotherMaliciousFunction();
            }
        }
    }
}
```

```
public class NestedOpaquePredicates {

    public void complexNestedLogic(String input) {
        // Multiple layers of opaque predicates

        // Level 1 opaque predicates
        boolean level1_true = input.length() >= 0;  // Always true
        boolean level1_false = input.hashCode() == (input.hashCode() + 1);  // Always false

        if (level1_true) {
            // Level 2 opaque predicates
            int hash = input.hashCode();
            boolean level2_true = (hash * hash + hash) % 2 == 0;  // Always true
            boolean level2_false = (hash * (hash + 1)) % 2 == 1;  // Always false

            if (level2_true && !level2_false) {  // Always true condition
                // Level 3 opaque predicates
                long timestamp = System.nanoTime();
                boolean level3_true = timestamp > 0;  // Always true

                if (level3_true) {
                    // Actual business logic buried deep
                    processInput(input);
                }

                if (level2_false) {
                    // Dead code
                    maliciousFunction();
                }
            }

            if (level1_false) {
                // More dead code
                anotherMaliciousFunction();
            }
        }
    }
}
```

Manual De-obfuscation Methodologies

- Phase 4: Reflection De-obfuscation
- Biggest challenge: Not finding method and class names during static analysis

Manual De-obfuscation Methodologies

- Phase 4: Reflection De-obfuscation
- Instead of: SecurityManager.checkPermission()

```
public void hiddenMethodCalls() throws Exception {
    Class<?> securityClass = Class.forName("java.lang.SecurityManager");
    Method checkMethod = securityClass.getMethod("checkPermission",
        Class.forName("java.security.Permission"));
```

Manual De-obfuscation Methodologies

“checkpermission” is now a string

- Phase 4: Reflection De-obfuscation
- Instead of: SecurityManager.checkPermission()

```
public void hiddenMethodCalls() throws Exception {
    Class<?> securityClass = Class.forName("java.lang.SecurityManager");
    Method checkMethod = securityClass.getMethod("checkPermission",
        Class.forName("java.security.Permission"));
```

Manual De-obfuscation Methodologies

- Phase 4: Reflection De-obfuscation
- NO CLASS NAMES!

```
public void dynamicClassLoading() throws Exception {
        String className = computeClassName();
        Class<?> dynamicClass = Class.forName(className);
        Object instance = dynamicClass.newInstance();
```

De-obfuscation Tools

- Java decompilers: jadx, cfr, procyon, fernflower
- Dex analyzers: baksmali, dex2jar, enjarify
- String extractors: strings, binwalk, custom extractors
- Control flow analyzers: ghidra, ida pro, radare2
- Crypto analyzers: cryptography, pycrypto, custom crypto

Hybrid, Cross-Platform & WebView-Based App Security

Hybrid vs Native vs Cross-Platform Apps

- **Native Apps:** Platform-specific development (Java/Kotlin for Android, Swift/Objective-C for iOS)
- **Hybrid Apps:** Web technologies wrapped in native containers
- **Cross-Platform:** Single codebase targeting multiple platforms

Hybrid vs Native vs Cross-Platform Apps

- Applications that use web technologies - HTML, CSS, JavaScript - but package them inside a native container
- E.g. Cordova
  - ”Web app wrapped it in a WebView component inside a native shell”
  - JavaScript code runs in this WebView
  - when it needs to access native features (camera or GPS) it communicates through a 'bridge'

Hybrid vs Native vs Cross-Platform Apps

- React Native
  - No Webview
  - JavaScript code gets translated into native UI components
  - “Bridge” for communication between JavaScript and native code

Hybrid vs Native vs Cross-Platform Apps

- Flutter
  - Flutter compiles Dart code to native machine code
- Xamarin
  - runs .NET code through a runtime layer

Security-wise?

- Attack surface
- Traditional native apps: the native code
- Web apps: web technologies.
- Hybrid apps: BOTH + the communication bridge between them

Recon

- Cordova: www directory
- React Native: bundle files ('index.android.bundle' or 'main.jsbundle')

Testing for Java Objects Exposed Through WebViews (MASTG-TEST-0033)

```
public class MSTG_ENV_008_JS_Interface {

        Context mContext;

        /** Instantiate the interface and set the context */
        MSTG_ENV_005_JS_Interface(Context c) {
            mContext = c;
        }

        @JavascriptInterface
        public String returnString () {
            return "Secret String";
        }

        /** Show a toast from the web page */
        @JavascriptInterface
        public void showToast(String toast) {
            Toast.makeText(mContext, toast, Toast.LENGTH_SHORT).show();
        }
}
```

Source: https://mas.owasp.org/MASTG/tests/android/MASVS-PLATFORM/MASTG-TEST-0033/#static-analysis

Local Storage Abuse in Hybrid Apps

- Traditional Web Apps:
  - Local storage (localStorage, sessionStorage) exists only in the browser
  - Accessible only via JavaScript in the web context
  - Protected by same-origin policy

Local Storage Abuse in Hybrid Apps

- Traditional Native Apps:
  - Use native storage mechanisms (SharedPreferences on Android, NSUserDefaults on iOS)
  - Only accessible to the native app code
  - Protected by OS-level app sandboxing

Local Storage Abuse in Hybrid Apps

- Hybrid Apps - The Problem:
  - Have BOTH web storage AND native storage
  - The bridge between web and native can access both layers
  - Creates multiple pathways to the same data

Security Issues

- Dual Access Pathways: Sensitive info reachable via WebView and Local Storage
- Attack:
  - Attacker injects JavaScript ()
  - Uses bridge to access native storage:via XSS or compromised WebView NativeBridge.getSecureData('encryption_key')
  - Escalates from web-layer compromise to native data access

Dynamic Instrumentation in Hybrid Apps

Project: Root detection bypass for cordova plugin devicecompile

Try this code out now by running `$ frida --codeshare damaidec/root-detection-bypass-for-cordova-plugin-devicecompile -f YOUR_BINARY`

```
Java.perform(function(){
    try {
        var Root = Java.use("cordova.plugin.devicecompile.d

        if (Root) {
            console.log("cordova.plugin.devicecompile detec
            Root.IsDrived.overload().implementation = funct
                return false;
            };
        } else {
            console.log("cordova.plugin.devicecompile Not d
        }
    } catch (error) {
        console.error("An error occurred:", error);
    }
});
```

Project: Hermes engine hook react native function calls

Try this code out now by running `$ frida --codeshare Thwarakan/hermes-engine-hook-react-native-function-calls -f YOUR_BINARY`

```

let libhermesBaseAddress = Module.findBaseAddress("libhermes.so");
let hermesRuntimeImplCallAddress = libhermesBaseAddress.add(0x1f3931 - 0x00100000);
let runtimePtr = Module.findExportByName("libhermes.so", "_ZN8facebook6hermes17makeHermesRuntimeERKN6hermes2vm13RuntimeConfigE");
let valueToStringAddr = Module.findExportByName("libjsi.so", "_ZNK8facebook3jsi5Value8toStringERNS0_7RuntimeE");


Interceptor.attach(hermesRuntimeImplCallAddress, {
    onEnter: function(args) {
        console.log("HermesRuntimeImpl::call intercepted");

        // Extracting arguments
        let func = args[1]; // jsi::Function
        let jsThis = args[2]; // jsi::Value
        let jsArgs = args[3]; // jsi::Value
        let count = args[4]; // size_t count

        // Logging arguments
```

Hybrid App Checklist

- *Identify the hybrid framework used*
- *Extract and analyze web assets*
- *Test WebView security controls*
- *Discover and exploit JavaScript bridges*
- *Analyze local storage security*
- *Document findings with proof-of-concept*

Architecture Components

+ WebView containers and JavaScript engines
+ Native bridge layers and API access
+ Plugin ecosystems and third-party integrations
+ Asset packaging and code organization

Framework-Specific Testing Approaches

- Cordova/PhoneGap Security Testing
  - config.xml configuration analysis
  - Plugin permissions and capabilities
  - Content Security Policy (CSP) implementation
  - File system access controls

Framework-Specific Testing Approaches

- Cordova/PhoneGap Security Testing
- Common Vulnerabilities:
  - Unrestricted network access
  - Excessive plugin permissions
  - Weak CSP policies
  - Insecure local file access

Framework-Specific Testing Approaches

- Ionic Framework Assessment
- Angular/React/Vue component security
- Capacitor vs Cordova plugin usage
- HTTP interceptors and API calls
- State management vulnerabilities

Framework-Specific Testing Approaches

- Ionic Framework Assessment
- Testing Approach
  - Source code analysis of TypeScript/JavaScript
  - Network traffic interception
  - Local storage inspection
  - Plugin vulnerability assessment

Framework-Specific Testing Approaches

- React Native Security Analysis
  - **JavaScript Thread (JS Thread):** Runs your React Native JavaScript code, handles business logic, state management, and API calls
  - **UI Thread (Main Thread):** Handles native UI rendering, user interactions, and system events
  - **Bridge:** Asynchronous communication layer between the two threads

Framework-Specific Testing Approaches

- React Native Security Analysis
- Security Implications
  - Data Serialization Vulnerabilities
  - Timing Attacks
  - Thread Isolation Bypass
  - Memory Corruption
- https://codeshare.frida.re/@Thwarakan/hermes-engine-hook-react-native-function-calls/

Framework-Specific Testing Approaches

+ Flutter Application Testing
+ Unique Characteristics

```
Dart Source Code → Dart Kernel → AOT Compiled Native Code → Binary Executable
    ↓
main.dart → main.dill → libapp.so (Android) / App.framework (iOS)
```

Framework-Specific Testing Approaches

- Flutter Application Testing

```
strings libapp.so | grep -E "(http|api|key)"
```

- Might reveal:
  - https://api.secret-server.com/v1/
  - sk-1234567890abcdef
  - production-database-key

WebView Security

- WebView Protection Mechanisms
  - Same-Origin Policy enforcement
  - JavaScript interface restrictions
  - File access limitations
  - SSL certificate pinning

Local Storage Abuse in Hybrid Apps

- Storage Mechanisms in Hybrid Apps
  - HTML5 Local Storage and Session Storage
  - WebSQL and IndexedDB,
  - Cordova/PhoneGap file storage
  - Framework-specific storage solutions

Local Storage Abuse in Hybrid Apps

- Common Storage Vulnerabilities
  - Sensitive Data Exposure:

```
localStorage.setItem('user_token', 'eyJhbGciOiJIUzI1NiIs...');
localStorage.setItem('api_key', 'sk-1234567890abcdef');
```

Local Storage Abuse in Hybrid Apps

- Common Storage Vulnerabilities
  - Session Management Issues:
    - Persistent login tokens
    - Unencrypted user data

Local Storage Abuse in Hybrid Apps

- Common Storage Vulnerabilities
  - Storage Analysis Techniques:

```
adb shell
cd /data/data/com.app.name/app_webview/Local\ Storage/
cat *.localstorage
```

Dynamic Instrumentation in Hybrid Apps

- Frida WebView hooking

```
Java.perform(function() {
    var WebView = Java.use("android.webkit.WebView");
    WebView.loadUrl.implementation = function(url) {
        console.log("[+] WebView loading: " + url);
        this.loadUrl(url);
    };
});
```


Key Takeaways

- Hybrid apps inherit both web and mobile security challenges
- JavaScript bridges represent a critical attack surface
- WebView configurations must be carefully analyzed

Key Takeaways

- Local storage in hybrid apps can expose sensitive data
- Dynamic instrumentation provides powerful capabilities
- Framework-specific knowledge is essential

Mobile Malware Analysis

Mobile Malware Goals and Objectives

- Primary Targets:
  - Personal Data
  - Credentials
  - Device Information
  - Corporate Data

Mobile Malware Goals and Objectives

- Data Exfiltration Methods:
  - HTTP/HTTPS POST requests to C2 servers
  - SMS to premium numbers
  - Email attachments
  - Cloud storage uploads
  - Bluetooth/NFC transmission (rare)

Surveillance Malware (Spyware)

- **Audio Recording:** Ambient sound capture, call recording
- **Video Capture:** Camera access, screen recording
- **Location Tracking:** GPS, cell tower, Wi-Fi positioning
- **Communication Monitoring:** SMS, messaging apps, email
- **Keystroke Logging:** Password and text input capture

Financial Fraud Malware

- **Banking Trojans:** Overlay attacks on banking apps
- **SMS Interception:** OTP and 2FA bypass
- **Cryptocurrency Theft:** Wallet credential stealing
- **Premium Service Fraud:** Unauthorized subscriptions and charges

Financial Fraud Malware

```
public class OverlayService extends Service implements View.OnTouchListener {
    private WindowManager windowManager;
    private View overlayView;
```

Mobile Infection Vectors

- **Malicious Apps:** Fake utilities, games, productivity apps
- **Trojanized Legitimate Apps:** Repackaged popular applications
- **Developer Account Compromise:** Pushing malware through legitimate channels
- **Update Poisoning:** Malicious updates to legitimate apps

Mobile Infection Vectors

- **Social engineering**
- **USB Charging Stations:** Juice jacking attacks
- **Malicious QR Codes:** Direct APK downloads
- **Bluetooth Exploitation:** BlueBorne and similar attacks
- **Wi-Fi Network Attacks:** Captive portal malware distribution

"Pokemon GO" kullanıcılarına siber tuzak

Siber güvenlik uzmanı Alper Başaran yaptığı açıklamada, dünyayı saran "Pokemon GO" çılgınlığının siber suçlular için de önemli bir fırsat haline geldiğine dikkati çekti.

13 Temmuz 2016 Çarşamba
14:52

Bir çizgi filmden uyarlanan "Pokemon GO" adlı artırılmış sanal gerçeklik oyununu, forumlar ya da resmi satış mağazası dışındaki kaynaklardan yükleyen kullanıcıların, "droidjack" olarak adlandırılan bir "arka kapı" yazılımıyla telefonlarındaki bilgilerin çalınabileceği bildirildi.

Siber güvenlik uzmanı Alper Başaran yaptığı açıklamada, dünyayı saran "Pokemon GO" çılgınlığının siber suçlular için de önemli bir fırsat haline geldiğine dikkati çekti.

Olamaz....

.....Yanılmıyorsam

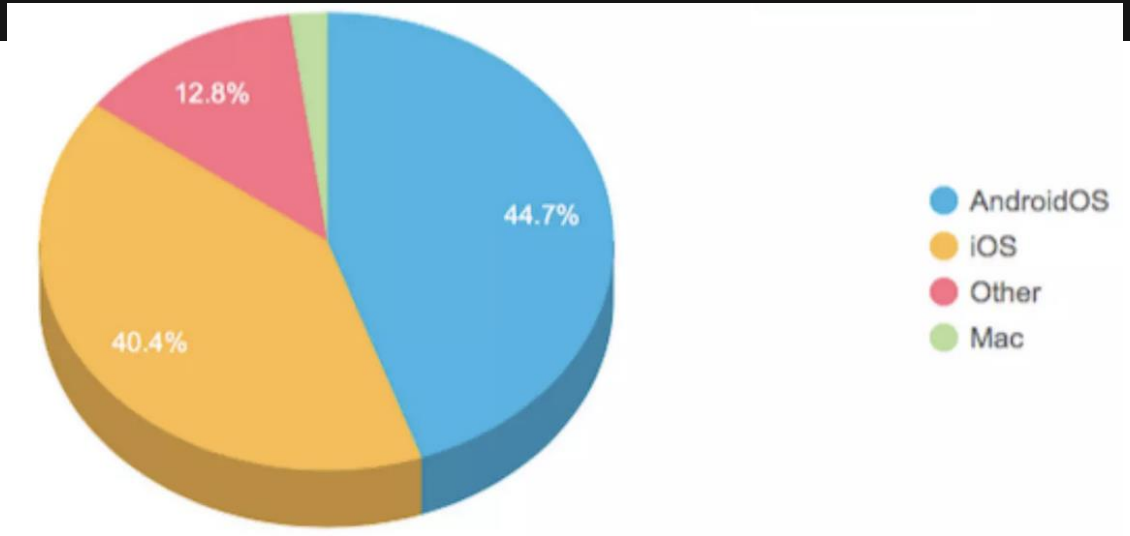
FENA OLTALANDINIZ!

Ama paniklemeyin... Burada kötü niyetli kimse yok.

Az önce telefonunuza okuttuğunuz kod, telefonunuza bu web sayfasına gitmesini söyledi. Eğer bu site kötü niyetli bir site olsaydı, telefonunuzu tamamen ele geçirebilirdi. Üstelik siz bunun farkına bile varmazdınız.

Special thanks to: Berk Goksel

“Victim” Distribution



Mobile Infection Vectors

- Supply Chain Attacks
  - **Third-party Libraries:** Malicious SDKs and frameworks
  - **Development Tools:** Compromised development environments
  - **Distribution Networks:** CDN compromise
  - **Hardware Pre-installation:** Factory-installed malware

Caveat

- Malware analysis is easier if you know what you are looking for
- Check https://androzoo.uni.lu/ for samples and play around

Key Takeaways

- Combine static and dynamic analysis
- You may face anti-analysis techniques
- Continue to learn

Penetration Testing on Smart TVs & Embedded Mobile OS Devices

Android TV Security Model

- **SELinux Policies:** Mandatory access controls for TV-specific operations
- **Android Permissions:** Modified permission model for TV apps
- **Package Manager:** APK installation and verification
- **Content Providers:** Data sharing between TV apps

Smart TV Ecosystem Architecture

- Unique Attack Surface
  - TV-specific input methods (remote control, voice)
  - HDMI-CEC control interfaces
  - Cast/screen mirroring protocols
  - TV tuner and broadcast signal processing

Firmware Acquisition Methods

+ UART/Serial console access
+ JTAG debugging
+ wget http://firmware.samsung.com/[MODEL]/[VERSION]/firmware.zip

Common Firmware Components

- Bootloader (U-Boot)
- Kernel (Linux)
  - System binaries
  - Configuration files
  - Application frameworks
  - Default applications
- Application partition
- User data partition

Key Takeaways

- Multiple attack surfaces exist
- Common vulnerabilities
- Don’t skip firmware analysis
- Don’t skip network attacks
- Don’t skip web interface attacks
- Don’t skip researching the TV!!

Secure Mobile Development & DevSecOps

Secure Coding for Mobile APIs

- Principle 1: Never Trust the Client
  - ALWAYS apply server-side validation
  - The mobile app should only send identifiers (like itemId) never sensitive values (prices, permissions, business logic decisions, etc)

Secure Coding for Mobile APIs

- Principle 2: Input Validation is Your First Line of Defense
- Three Layers of Input Validation:
  - **Format Validation:** Is the data in the expected format?
  - **Range Validation:** Is the data within acceptable limits?
  - **Business Logic Validation:** Does this data make sense in your application context?

Secure Coding for Mobile APIs

- Principle 3: Secure Communication Channels
- Instead of trusting any certificate authority, you embed your server's certificate fingerprint directly in your mobile app.

Mastering Secure Mobile Storage

- The Storage Security Spectrum:
  - **Least Secure:** Plain text files, SharedPreferences, UserDefaults
  - **More Secure:** Application sandbox, private directories
  - **Most Secure:** Encrypted storage, hardware-backed keystores
- Android provides EncryptedSharedPreferences and EncryptedFile classes that handle encryption automatically.
- iOS Keychain

Hardware-Backed Security

- The Security Stack (from least to most secure):
  - **Application Layer:** Your app's security measures
  - **Operating System:** Platform security features
  - **Trusted Execution Environment (TEE):** Secure isolated environment
  - **Hardware Security Module:** Dedicated security chip

Avoid Common Security Anti-Patterns

- Anti-Pattern #1: Hardcoded Secrets
- Anti-Pattern #2: Client-Side Security Validation
- Anti-Pattern #3: Weak Session Management

Making Threat Modeling Practical

- Feature: [Feature Name]
- Assets - [List valuable data and functionality]
- Entry Points - [List all ways data enters the system]
- Trust Levels - [Define different user permission levels]
- STRIDE Analysis
  - Spoofing –
    - Threat: [Description]
    - Mitigation: [How to prevent] –
    - Test: [How to verify protection]
    - [Repeat for T, R, I, D, E]
- Risk Assessment
  - [High/Medium/Low priority threats]
  - [Recommended actions]

DevSecOps for Mobile Applications

- Why Mobile DevSecOps is Different?
  - Apps are distributed through app stores with review processes
  - Updates can't be deployed instantly like web applications
  - Users may not update apps immediately
  - Security issues in released apps affect devices you don't control

Mobile DevSecOps Pipeline

- Phase 1: Code Commit Security Checks
- Phase 2: Build-Time Security Validation
- Phase 3: Pre-Production Security Testing
- Phase 4: Continuous Security Monitoring

Mobile DevSecOps Pipeline
Phase 1: Code Commit Security Checks

- Check for hardcoded secrets
  - grep -r "api_key\|password\|secret" src
- Check for dangerous permissions in manifest
  - grep -i "dangerous" app/src/main/AndroidManifest.xml
- Dependency vulnerability scanning
- Fail build if critical vulnerabilities found

Mobile DevSecOps Pipeline
Phase 2: Build-Time Security Validation

- Run automated security analysis on built APK
- Verify APK is properly signed
- Test API endpoints for common vulnerabilities

Mobile DevSecOps Pipeline
Phase 3: Pre-Production Security Testing

- Test API security
- Test mobile app security
- Test authentication bypass
- Test authorization flaws
- OWASP Testing guide

Mobile DevSecOps Pipeline
Phase 3: Pre-Production Security Testing

- **Create Security Gates:** Security gates are automated checkpoints that prevent insecure code from progressing through your pipeline:

Mobile DevSecOps Pipeline
Phase 4: Continuous Security Monitoring

- Runtime security monitoring in the mobile app
  - Monitor for tampering attempts
  - Monitor for debugging attempts
  - Monitor API usage patterns
  - Check device security posture

Simple Workflow Introduction

```
build:
   - checkout code
   - compile application
   - run unit tests
   - deploy to staging
```

```
build:
  - checkout code
  - dependency scan (OWASP Dependency Check)
  - compile application
  - static security scan (SAST)
  - run unit tests
  - dynamic security scan (DAST)
  - security gate (pass/fail based on results)
  - deploy to staging
```

```
build:
  - checkout code
  - secret scanning (detect hardcoded credentials)
  - dependency scan with vulnerability database
  - license compliance check
  - compile application
  - static analysis (SAST) with custom mobile rules
  - container/artifact scanning
  - unit and integration tests
  - dynamic testing (DAST) on staging environment
  - penetration testing (automated)
  - compliance validation
  - security metrics collection
  - security gate with configurable thresholds
  - deploy with security attestation
  - runtime monitoring setup
```

Mobile-Specific SAST Rules

- Android-Specific Rules:
  - Detect exported activities without proper permission checks
  - Flag insecure intent handling (implicit intents for sensitive data)
  - Identify improper SSL certificate validation
  - Check for inadequate file permissions
  - Detect insecure shared preferences usage
  - Flag debug flags enabled in production builds

Mobile-Specific SAST Rules

- iOS-Specific Rules:
  - Identify insecure keychain usage
  - Detect improper URL scheme handling
  - Flag disabled ATS (App Transport Security)
  - Check for insecure backup inclusion
  - Detect logging of sensitive information
  - Flag improper biometric authentication implementation

Mobile-Specific SAST Rules

- Cross-Platform Rules:
  - Hardcoded cryptographic keys and passwords
  - Weak cryptographic algorithms (MD5, SHA1, DES)
  - Insecure random number generation
  - SQL injection vulnerabilities in local databases
  - Insecure network communication patterns

SAST Integration Strategies

- **IDE Integration:** Catches issues before they're even committed.
- **Pre-commit Hooks:** Prevents vulnerable code from entering your codebase.
- **Pull Request Integration:** Provides security review context for code reviewers.
- **Build Pipeline Integration:** Ensures no vulnerabilities slip through.

SAST Pipeline

```
sast-analysis:
   runs-on: ubuntu-latest
   steps:
   - name: Checkout Code
     uses: actions/checkout@v4

   - name: Build Application
     run: |
       # Android build
       ./gradlew assembleDebug
       # iOS build would go here for iOS projects

   - name: MobSF Static Analysis
     run: |
       # Run MobSF for comprehensive mobile security analysis

   - name: SonarQube Analysis
     env:
       SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
     run: |
       # Custom mobile security rules

   - name: Semgrep Security Scan
     run: |
       # Semgrep with mobile security rules

   - name: Process SAST Results
     run: |
       python scripts/process-sast-results.py
```

Vulnerability Threshold Enforcement

- Block: Critical vulnerabilities with confirmed exploits
- Warn: High and medium severity issues
- Track: All findings for baseline establishment

Building a Security Culture

- Monthly Security Trainings
- Month 1: Secure Coding Fundamentals
- Month 2: Mobile Platform Security
- Month 3: API Security
- Month 4: Threat Modeling Workshop

Key Takeaways

+ Security is a Design Decision, Not an Add-On
+ Hardware-Backed Security is Your Best Friend
+ The Mobile Client is Hostile Territory
+ Automation Catches What Humans Miss
+ Threat Modeling Makes Security Concrete
+ Security Culture is as Important as Security Technology

Future Trends in Mobile Security

AI/ML in Mobile Malware Detection and Evasion
AI Arms Race

- Current State Analysis:
  - Traditional signature-based detection limitations
  - Rise of polymorphic and metamorphic mobile malware
  - Machine learning's role in behavioral analysis

AI/ML in Mobile Malware Detection and Evasion
AI Arms Race

- AI-Powered Detection Techniques:
  - Dynamic analysis using behavioral modeling
  - Graph neural networks for app flow analysis
  - Ensemble methods combining static and dynamic features
  - Real-time threat intelligence integration

AI/ML in Mobile Malware Detection and Evasion
Adversarial ML and Evasion Techniques

- Attacker Adaptations:
  - Adversarial examples in mobile malware
  - GAN-generated malicious code
  - Model extraction and inversion attacks
  - Concept drift exploitation

AI/ML in Mobile Malware Detection and Evasion
Adversarial ML and Evasion Techniques

- Defense Mechanisms:
  - Adversarial training techniques
  - Model ensemble diversity
  - Federated learning for privacy-preserving detection

Behavioral Biometrics and Future Authentication

- Emerging Behavioral Patterns:
  - Gait analysis using accelerometer/gyroscope data
  - Touch dynamics and pressure patterns
  - Voice authentication with emotional context
  - Eye movement tracking and attention patterns

Behavioral Biometrics and Future Authentication

- Continuous Authentication Systems:
  - Risk-based authentication scoring
  - Multi-modal biometric fusion
  - Context-aware authentication triggers
  - Privacy-preserving biometric templates

Post-Quantum Cryptography for Mobile Applications

- Current Cryptographic Vulnerabilities:
  - RSA and ECC susceptibility to Shor's algorithm
  - Timeline estimates for cryptographically relevant quantum computers
  - NIST post-quantum cryptography standardization process

Mobile Implementation Challenges

- Key size implications for mobile storage
- Computational overhead on mobile processors
- Battery life impact assessment
- Network bandwidth requirements

Emerging Platform Security Challenges
Mobile Cloud Gaming Security

- Architecture-Specific Threats:
  - Stream interception and manipulation
  - Latency-based timing attacks
  - Cloud-side save game tampering
  - Input lag exploitation for cheating

Emerging Platform Security Challenges
Mobile Cloud Gaming Security

- Security Measures:
  - End-to-end stream encryption
  - Client-side input validation
  - Secure state synchronization
  - Anti-cheat system integration

Emerging Platform Security Challenges
5G Application Security

- New Attack Surfaces:
  - Network slicing vulnerabilities
  - Edge computing security gaps
  - Massive IoT device management
  - Ultra-low latency requirements vs. security

Emerging Platform Security Challenges
5G Application Security

- 5G Security Enhancements:
  - Enhanced authentication protocols
  - Network function virtualization security
  - Secure multi-access edge computing
  - Privacy-preserving location services

Web3 and dApp Mobile Security

- Decentralized Application Risks:
  - Smart contract vulnerabilities in mobile interfaces
  - Private key management on mobile devices
  - Cross-chain bridge security
  - MEV attacks through mobile applications

Web3 and dApp Mobile Security

- Security Frameworks:
  - Hardware security module integration
  - Multi-signature wallet implementations
  - Decentralized identity management
  - Zero-knowledge proof applications

Zero-Trust Principles for Mobile

- Core Components:
  - Never trust, always verify
  - Assume breach mentality
  - Least privilege access
  - Continuous monitoring and validation

Zero-Trust Principles for Mobile

- Mobile-Specific Adaptations:
  - Device health attestation
  - Application behavior baselines
  - Network micro-segmentation
  - Identity-centric security policies

Zero-Trust Principles for Mobile

- Architecture Components:
  - Mobile device trust scores
  - Conditional access policies
  - API gateway security
  - Micro-segmented network access

Zero-Trust Principles for Mobile

- Integration Challenges:
  - User experience balance
  - Performance impact minimization
  - Legacy application compatibility
  - Offline scenario handling

AI Evasion Techniques?

- The Evolution of Detection:
  - **Signature-based:** "This exact code pattern is malicious"
  - **Heuristic-based:** "Code that does X, Y, and Z might be malicious"
  - **AI-based:** "This behavior pattern is statistically similar to malware"

Summary

Key Concepts

- Mobile application reverse engineering
- Security architecture and runtime behavior
- Malware analysis and penetration testing
- Secure development practices
- Future trends in mobile security

LEARNING OUTCOMES

+ Understand mobile application architecture and runtime behavior
+ Analyze and reverse-engineer mobile apps and malware
+ Perform penetration testing on diverse mobile platforms
+ Implement secure mobile development and DevSecOps strategies

Real-World Applications (Optional, but encouraged)

- Identifying malicious behaviors in sideloaded apps
- Testing the resilience of enterprise hybrid apps
- Strengthening DevSecOps pipelines for mobile environments

Next Steps

- Explore mobile threat hunting techniques
- Advance to red teaming for mobile platforms
- Follow INE's advanced cybersecurity or malware reverse engineering paths

THANKS FOR WATCHING!

EXPERTS AT MAKING YOU AN EXPERT