From Defense to Offense: A Comprehensive Security Analysis of Smart-device Apps

Chenye Ni
Faculty of Science
University of Auckland
Auckland, New Zealand
cni586@aucklanduni.ac.nz

Yuxin Lin
Faculty of Science
University of Auckland
Auckland, New Zealand
yiln330@aucklanduni.ac.nz

Xiaoqing Miao
Faculty of Science
University of Auckland
Auckland, New Zealand
xmia665@aucklanduni.ac.nz

Hongkai Li
Faculty of Science
University of Auckland
Auckland, New Zealand
hli615@aucklanduni.ac.nz

I. Fortify

In building our AI chatbot, we implemented a multi-layered set of security defenses to ensure the system is resilient against various potential attacks. This section details the security hardening strategies we adopted, the specific implementation steps, and the challenges encountered during the process along with how we addressed them.

A. Fake Domain Deception

We implemented a fake domain deception strategy by constructing a carefully designed honeypot system to mislead potential attackers. The core idea behind this strategy is to create a fake domain that closely resembles the real API domain, tricking attackers during code inspection or man-in-the-middle (MITM) interception attempts. This method is particularly effective because attackers searching for domain names in the Java layer are likely to encounter numerous seemingly legitimate API calls. During MITM attacks, the fake backend returns a 200 OK status with plausible responses. Unless the attacker notices the missing letter "t" in the domain name, they may fall into this carefully crafted trap and waste significant time analyzing misleading data.

1) Implementation Steps:

a. **Constructing the fake infrastructure**
- Deployed a static website under elliotwen.info to mimic the real app interface
- Set up a fake AI backend at ai.elliotwen.info to serve plausible responses

b. **Embedding obfuscation logic**
- Configured the Kotlin frontend to send 10 requests to the fake backend upon clicking the send button
- The fake backend returns 200 OK for selected API keys

c. **Native-layer URL manipulation**
- Implemented a concealed string processing function in the native layer
- Dynamically removed one "t" from the stored URL ai.elliottwen.info before request dispatch
- Kept the double-"t" URL string in the Java layer to facilitate discovery by attackers

d. **Enhanced deception effects**
- Implemented background jobs that periodically send requests to the /auth endpoint
- These calls serve as camouflage for later retrieval of real key fragments through a custom /auth route

2) Challenges: Creating a highly convincing fake backend posed a number of challenges. The deceptive service had to be realistic enough across multiple layers to fool attackers. We rapidly developed a fully functional fake backend system (https://github.com/MarlinDiary/FakeBackend) and deployed it on the Railway platform to ensure stability. This effort yielded results far beyond our expectations—according to peer feedback, one attacker tried over 100 API keys on the honeypot, all of which failed. Another student was misled during memory inspection, further validating the effectiveness of the deception strategy.

B. Key Fragmentation

We implemented a multi-layered key fragmentation strategy that splits the API key into five separate segments, each encrypted and stored in a different location using distinct methods. The primary goal of this design is to ensure that attackers cannot retrieve the full key through static analysis or reverse engineering alone. By distributing the fragments across the Java layer, native layer, network responses, and image metadata—and combining multiple encryption techniques such as AES and XOR—we constructed a complex key reconstruction chain.

1) Implementation Steps:

a. **Environment Setup**
- Integrated the OpenSSL library for AES decryption
- Integrated the CURL library for network request functionality

b. **Fragment 1: Java-Native Hybrid Storage**
- Encrypted the first key fragment using AES
- Stored the decryption key in the Java layer
- Hardcoded the ciphertext in the native layer
- Used OpenSSL at runtime to perform decryption

c. **Fragment 2: Native-Layer Isolated Storage**
- Encrypted the second key fragment with AES
- Stored both the key and ciphertext in different native-layer locations

- Used OpenSSL within the native layer to decrypt at runtime

d. **Fragment 3: Dynamically Retrieved via Network**
- Implemented a custom `/auth` endpoint that returns a fixed signature when queried with a specific trigger key
- Encrypted both the trigger key and the real key fragment using AES
- At runtime, sent a request to `/auth` and extracted a 16-character decryption key from position 10 of the signature
- Decrypted the actual key fragment using the extracted key

e. **Fragment 4: Hidden in Image Metadata**
- Encrypted the fourth key fragment using AES
- Embedded the ciphertext into the UserComment field of PNG image metadata
- Exposed the image through the `/generate_image` endpoint

f. **Fragment 5: Custom XOR-Based Encryption**
- Performed three rounds of XOR encryption, each using a different key
- Stored all three XOR keys and the encrypted content in the native layer
- Implemented a custom decryption algorithm in the native code

2) Challenges: Integrating C libraries such as OpenSSL and CURL was one of the most technically challenging aspects of this implementation. We ultimately chose to use precompiled open-source C libraries to simplify integration—a reasonable tradeoff given the project timeline, although this would pose security concerns in a production environment. In practice, we recommend compiling these libraries from source to ensure integrity and security.

While implementing image-based metadata storage, we discovered that Android provides inconsistent support for metadata across different image formats. Specifically, native Android APIs offer very limited support for reading custom metadata fields from JPEG files. Through testing, we found that PNG is the most reliable format for this purpose, consistently supporting read/write access to fields such as UserComment, making it ideal for this use case.

C. Native-Layer Networking

We implemented a native-layer networking strategy, shifting all sensitive API requests entirely to the native code. The key advantage of this approach is that unless the attacker reroutes all HTTPS traffic at the device level, common proxy tools such as Charles or mitmproxy are unable to intercept the requests, as they do not pass through Android's network stack. Similarly, general-purpose Frida scripts struggle to hook into native-layer CURL requests. Most importantly, this architecture ensures that the full API key and the key assembly process never appear in the Java layera critical security consideration, as exposing sensitive data in Java significantly increases the attack surface and associated risks.

1) Implementation Steps:

a. **Runtime Key Assembly**
- Concatenate all key fragments within the native layer
- The fully assembled key exists only briefly in memory and is cleared immediately after use

b. **Native Network Communication**
- Use the CURL library to perform HTTPS requests
- Return only the minimal processed result to the Java layer via JNI

2) Challenges: During the hardening phase, our initial understanding of native-layer proxy behavior was incomplete. We incorrectly assumed that native-layer networking was entirely isolated from the Android system and thus immune to interception by proxy tools. This led to a gap in our man-in-the-middle (MITM) attack defenses. In reality, although CURL-based requests do not obey Android's proxy settings, attackers can still redirect traffic at the system level (e.g., using iptables rules) and potentially intercept communications. This gap in understanding highlighted the importance of developing a deeper and more holistic grasp of the underlying networking mechanisms when designing security measures.

D. MitM Protection

We implemented a multi-layered defense mechanism against man-in-the-middle (MitM) attacks, as a successful MitM compromise would expose all HTTP headers to the attacker. To mitigate this threat, we deployed certificate pinning at the application layer, added additional verification logic in the native layer, and actively monitored for suspicious proxy configurations. This layered strategy ensures that even if one protection layer is bypassed, others remain in place to minimize the risk of sensitive data leakage.

1) Implementation Steps:

a. **Application-Layer Certificate Pinning**
- Used CertificatePinner to pin the R10 certificate for the fake domain
- Blocked the key assembly process upon certificate validation failure

b. **Native-Layer Certificate Validation**
- Used OpenSSL to independently verify the R10 certificate within the native layer
- Implemented a screen-freeze mechanism in case of certificate validation failure
- This layer acts as a last-resort safeguard and is rarely triggered

c. **Proxy Detection Mechanism**
- Performed proxy checks both at app launch and before sending requests
- Inspected system properties such as `http.proxyHost` and `https.proxyHost`
- Scanned environment variables such as `HTTP_PROXY` and `HTTPS_PROXY`

- Triggered immediate screen freeze if suspicious proxy usage was detected

E. Frida Detection

We implemented a Frida detection mechanism at the native layer to proactively check for the presence of Frida before performing any key assembly. As one of the most popular dynamic analysis tools, Frida is frequently used to hook or modify application behavior, posing a serious threat to key confidentiality. By performing detection before executing sensitive operations, we can effectively block attackers from using Frida to analyze or extract key materials.

1) Implementation Steps:

a. **Memory Mapping Inspection**
- Read the contents of the `/proc/self/maps` file
- Searched for suspicious patterns such as "frida" or "gum-js-loop"

b. **Default Port Scanning**
- Attempted to connect to port 27042, which is the default port used by Frida
- A successful connection indicates that a Frida server is active

c. **Library Load Check**
- Inspected dynamic libraries loaded into memory via process maps
- Searched for known Frida artifacts such as "frida-agent.so" or "libfrida-gadget.so"

F. Root Detection

We implemented a dual-layer root detection mechanism, performing comprehensive checks at both the Java and native levels. Rooted devices pose a serious threat to application security, as attackers can easily extract sensitive data. Through rigorous root detection, we ensure that the application does not run in insecure environments, thereby protecting API keys at the most fundamental level.

1) Implementation Steps:

a. **Java-Layer Root Detection**
- Checked for the presence of known root management app package names
- Scanned common file paths for the existence of the `su` binary
- Attempted to execute the `su` command and analyzed the response
- Inspected the build tags for suspicious keywords
- Queried system properties for root indicators

b. **Native-Layer Root Detection**
- Iterated through common paths to locate the `su` binary
- Parsed `/proc/mounts` to check if system partitions were mounted as writable
- Scanned for directories associated with root management tools
- Detected hidden Magisk directories
- Attempted to run the `su` command from native code
- Tried to write to protected directories such as `/system` to test for elevated privileges

2) Challenges: After enforcing strict root detection, we discovered that the application would only run properly on official Android systems that include Google Play Services. Many custom ROMs or devices lacking Google services were misidentified as insecure environments. Although this reduced device compatibility, we considered it a necessary trade-off given the sensitivity of the API key and the security standards required.

G. Code Obfuscation

We applied a range of code obfuscation techniques to our native code using Obfuscator-LLVM (OLLVM) during the build process. Since all sensitive logicincluding key handling, network requests, and security checkswas implemented in the native layer, this code became the primary target for attackers. By applying advanced techniques such as control flow flattening, bogus control flow insertion, and instruction substitution, we significantly increased the difficulty of static analysis, making it harder for attackers to understand or tamper with critical security mechanisms.

1) Implementation Steps:

a. **Configure Build Environment**
- Added OLLVM obfuscation flags to the `CMakeLists.txt` configuration
- Defined compiler flags for debug builds to enable obfuscation during development

b. **Apply Obfuscation Techniques**
- Enabled **control flow flattening (fla)**: transformed function logic into flat `switch-case` structures
- Enabled **bogus control flow (bcf)**: inserted fake branches with an 80% probability, repeated 3 times for deeper layering
- Applied **instruction substitution (sub)**: replaced simple instructions with semantically equivalent but more complex sequences, repeated 3 times
- Performed **basic block splitting (split)**: split each basic block into 5 smaller fragments to obscure logic

II. Attack

During the security testing phase, we successfully identified and exploited several vulnerabilities in the applications developed by other teams. This section outlines the discovered weaknesses, exploitation techniques, potential impacts, and recommended mitigation strategies. We hope these findings contribute to improving the overall security posture of the class.

A. Open-source Intelligence (OSINT)

We conducted information gathering through open-source intelligence (OSINT) techniques. Specifically, we searched GitHub for the domain `ai.elliottwen.info` and unexpectedly discovered several public repositories containing sensitive information. By analyzing the commit history, branches,

and source code of these repositories, we were able to retrieve hardcoded API keys.

1) Affected Teams:

- **Team 2**: A public GitHub repository was found containing the full source code and hardcoded API keys.
- **Virus Soup**: Also exposed a repository with sensitive code publicly accessible.

2) Impact Analysis: Leaving code repositories public poses a serious security risk. Attackers can analyze the source code to understand internal logic, identify potential vulnerabilities, and extract hardcoded secrets or configuration files. In real-world scenarios, such leaks may result in data breaches, service abuse, or complete system compromise.

3) Mitigation Strategies:

a. Always set repositories containing sensitive code to *private*, with strict access control.
b. Use dummy or test API keys during development to ensure that accidental leaks do not expose production credentials.

B. Algorithm Reproduction via Reverse Engineering

We discovered that several teams, although relocating the encryption logic to the Native layer, failed to apply sufficient obfuscation protections. Using reverse engineering tools such as IDA Pro, we were able to analyze native binaries, reconstruct the encryption logic in Python, and ultimately recover the API keys.

1) Affected Teams:

a. **Anonymous**
   - The API key was cleverly hidden in the pixel data of a `Static.png` image, with the decoding algorithm implemented in the Native layer.
   - However, due to the lack of obfuscation, we successfully extracted the complete image analysis logic using IDA Pro.
   - We reimplemented the algorithm in Python and extracted the key from the image.

b. **Farming No Fighting**
   - A relatively simple XOR encryption scheme was used.
   - Through reverse analysis, we identified the algorithm: 128 bytes of encrypted data and a key were XORed byte-by-byte in the Native layer.
   - The simplicity of the scheme made reproduction trivial.

c. **Team 4**
   - The application had runtime issues that prevented dynamic debugging.
   - Nevertheless, we statically analyzed the binary with IDA Pro and recovered the encryption logic.
   - A Python reimplementation enabled us to recover the key.

d. **Sample App 5**
   - Used Claude's Model Context Protocol (MCP) to automate interactions with IDA Pro.
   - The native-stored API key was successfully extracted via automated decryption.

2) Impact Analysis: Without proper obfuscation, native code offers minimal resistance to static analysis. Once an encryption algorithm is understood and reimplemented, all associated protections become ineffective. Simple algorithms like XOR are particularly vulnerable, as even partial knowledge or basic cryptanalysis may suffice to break them. In practice, such weaknesses may result in API key exposure, unauthorized service usage, and financial loss.

3) Mitigation Strategies:

a. Apply strong obfuscation techniques (e.g., OLLVM) to all Native code to increase the difficulty of reverse engineering.
b. Replace weak algorithms like XOR with industry-standard encryption (e.g., AES), and enforce secure key management.
c. Use dynamic encryption mechanisms to avoid fixed logic that can be fully understood via static analysis.
d. Distribute key fragments and core algorithms across multiple modules to increase analysis complexity.

C. Static Tampering via Smali Code Modification

We identified several teams whose apps were vulnerable to static tampering through Smali code. By decompiling APK files and modifying specific Smali functions to log sensitive data, attackers can repackage the app and extract API keys at runtime. This attack technique is particularly effective because it bypasses most runtime protections.

1) Affected Teams:

a. **Payload Injectors**
   - Modified the `retrieveAuthHeader` function in Smali.
   - Injected log statements to print authentication headers to the console.

b. **REversanity**
   - Attempted to obscure logic by moving it to Native layer.
   - However, key logic remained in Java and was easily tampered with via Smali.
   - Successfully logged critical variable values with inserted print statements.

c. **The Sorcerer**
   - Applied Java obfuscation to increase analysis difficulty.
   - Careful analysis revealed function `a` as the target.
   - Modified Smali to extract return values from the function.

d. **The Z Squad**
   - Systematically analyzed Smali code to locate sensitive functions.
   - Inserted logging at function exits to capture confidential data.

e. **Sample App 4**
   - Found API key logic directly in Smali without obfuscation.

- Added simple logging statements to expose the entire key.

2) Impact Analysis: Smali-based static tampering is especially dangerous because it bypasses many common protections. Even when apps use complex encryption and obfuscation, any plaintext key that appears in the Java layer can be captured via inserted logging logic. This poses a serious threat to any Android app that handles sensitive data in Java.

3) Mitigation Strategies:

a. Implement application signature verification and integrity checks to detect APK tampering.
b. Offload all sensitive operations to the Native layer to reduce the Java attack surface.
c. Use advanced obfuscation tools to increase the difficulty of analyzing Smali code.

D. Dynamic Instrumentation via Frida

We conducted runtime analysis and behavioral modification of several apps using the dynamic instrumentation framework **Frida**. Frida allows attackers to inject JavaScript code during app execution, hook critical functions, bypass security checks, and extract sensitive data in real time. This method is particularly powerful because it enables behavioral alteration without modifying the APK itself.

1) Affected Teams:

a. **Click Here for Free Wifi**
   - Implemented Frida detection and clever integrity validation (integrity was part of the key computation logic).
   - However, due to lack of obfuscation in the Native layer, we were able to locate the detection function.
   - Used Frida to hook and bypass all protection mechanisms.

b. **coriander-lovers**, **Teacher likes our team**, **Team 3**
   - These teams implemented certificate pinning.
   - Successfully bypassed pinning via Frida hook.
   - Used Charles proxy to intercept API keys from HTTP headers.

c. **Flamingos**, **The Latecomers**, **Sample App 1**
   - Hooked network-related functions using Frida.
   - Intercepted full request headers before transmission.
   - No need for proxy tools to obtain the API key.

d. **Sample App 2**
   - API key split into 15 fragments.
   - Hooked all fragment retrieval functions via Frida.
   - Successfully reassembled the complete key.

e. **Sample App 3**
   - Identified key JNI function `stringFromJNI()`.
   - Hooked the function with Frida to capture the returned key from Native layer.

2) Impact Analysis: Dynamic instrumentation attacks highlight the importance of runtime protection. Even with comprehensive static protections in place, lack of runtime defenses allows tools like Frida to completely bypass security layers. In particular, Native code without obfuscation is vulnerable to rapid analysis and function hooking. In real-world deployments, this could lead to bypassed authentication, decrypted communication, stolen sensitive data, and tampered app logic.

3) Mitigation Strategies:

a. Implement Frida detection in both Java and Native layers with cross-validation.
b. Obfuscate all security-related functions to prevent easy identification.
c. Continuously verify the integrity of critical functions at runtime.
d. Enforce multi-layered certificate validation to resist hooking attempts.
e. Apply strict root detection, since tools like Frida typically require root access.

E. Memory Scanning via Cheat Engine

We performed remote memory scanning and analysis using Cheat Engine in combination with CEServerARM64. This attack method directly inspects the application's runtime memory space to search for specific data patterns or strings. Even if the application applies sophisticated encryption and protection mechanisms, any sensitive data that appears in plaintext in memory at any moment may be discovered and extracted.

1) Affected Teams:

a. **ChaosIsALadder**
   - This team had the most robust protections among all we tested.
   - Key concatenation, security checks, and network requests were all implemented in the Native layer.
   - Traditional static analysis and dynamic hooking were ineffective.
   - By establishing a remote connection via CEServer-ARM64, we scanned memory space and successfully located and dumped the complete API key data.

2) Impact Analysis: Memory scanning attacks expose a fundamental security issue: no matter how sophisticated the protection mechanisms are, sensitive data must exist in readable form in memory during use. This makes memory scanning particularly dangerous, as it can bypass nearly all traditional protection layers. In real-world scenarios, malware or attackers with system-level privileges can extract sensitive information such as passwords, keys, and user data via memory scanning. Even data that exists only briefly in memory can be captured.

3) Mitigation Strategies:

a. Store numerous fake keys in memory to increase the difficulty of identifying the real one.
b. Detect known memory inspection tools and behaviors, and terminate the app immediately if such tools are found.
c. Offload all sensitive API request logic to a proprietary backend server so that the client only communicates with your own server, eliminating exposure of third-party API keys on the client side.