Advances in Computer and Communications, 2023, 4(5), 309-317
https://www.hillpublisher.com/journals/acc/
ISSN Online: 2767-2875

Android Reverse Engineering and Defense Analysis Taking a Remote Rehabilitation Training APP as an Example

Bin Liu

Panzhihua College, Panzhihua, Sichuan, China.

How to cite this paper: Bin Liu. (2023) Android Reverse Engineering and Defense Analysis Taking a Remote Rehabilitation Training APP as an Example. *Advances in Computer and Communication*, 4(5), 309-317.
DOI: 10.26855/acc.2023.10.009

Received: September 28, 2023
Accepted: October 26, 2023
Published: November 22, 2023

***Corresponding author:** Bin Liu, Panzhihua College, Panzhihua, Sichuan, China.

Abstract

With the widespread use of Android smartphones and the rapid development of mobile application technologies, the number of malicious attacks and security threats is increasing. Reverse engineering techniques provide effective protection for discovering and analyzing malicious applications, exploiting vulnerabilities, and protecting software mechanisms. This paper combines an analysis of the current state of Android software security with an understanding of Android reverse engineering, including the structure of APK files and decompilation techniques. Taking a remote rehabilitation training app as an example, this paper explores the analysis of malicious applications, vulnerability exploitation, and software protection mechanisms using reverse engineering tools such as Kali Linux, APKTool, and Jadx-GUI. It also provides strategies for secure coding practices, application hardening, and runtime protection mechanisms. Through this research, we aim to deepen our comprehensive understanding of Android reverse engineering and defense. This will help developers enhance the security of their Android applications and protect them from malicious attacks.

Keywords

Android, reverse engineering, vulnerability analysis, software protection mechanisms, secure coding

1. Introduction

Android, as one of the most popular mobile operating systems, boasts a vast application ecosystem, making Android software security a critical area of concern [1]. However, due to its open nature and extensive user base, Android mobile software faces a myriad of security issues. Currently, the Android platform is confronted with various security threats, including malicious software and viruses, information leakage, privacy concerns, unauthorized access and misuse of permissions, network attacks, and vulnerabilities, among others. Understanding the current state of Android software security is paramount for developers and users alike. It aids in raising awareness of security risks and promotes the development and usage of more secure applications [2].

Android reverse engineering involves the analysis and research of Android applications to comprehend their internal structure, functionalities, and operational mechanisms. It encompasses techniques such as reverse analysis, decompilation, and dynamic debugging, all aimed at dissecting and understanding how applications function. Android reverse engineering holds significant importance and relevance. Through reverse engineering techniques, security experts can conduct in-depth analysis of malicious applications, identify potential vulnerabilities and threats, and provide corresponding defense measures. Reverse engineering aids in understanding the behavior patterns and attack methods of malicious code, improving security performance, enhancing the stability and security of applications, and

safeguarding intellectual property and business interests. Android reverse engineering plays a pivotal role in areas such as security research, vulnerability remediation, feature optimization, anti-piracy efforts, and intellectual property protection. It provides developers and security experts with effective tools and methods to analyze and safeguard Android applications [3].

2. Analysis of Android System Security Issues

Android system security issues primarily encompass three aspects: application security, system vulnerabilities, and malicious software.

(1) Application Security

Android applications often contain sensitive information, such as user personal data and account details. Reverse engineering techniques can be employed to analyze applications, unveiling their internal logic, which may empower hackers to attack applications, steal sensitive data, or forge applications.

(2) System Vulnerabilities

The Android operating system itself harbors security vulnerabilities, which hackers can exploit to gain system privileges, control devices, or execute denial-of-service attacks.

(3) Malicious Software

Malware developers employ reverse engineering techniques to analyze Android applications and subsequently inject malicious code into these applications, with the intent of acquiring user data, disseminating spam, or engaging in other illicit activities.

3. Fundamentals of Android Reverse Engineering

(1) APK File Structure and Decompilation

An APK file is the installation package for Android applications, comprising a compressed file containing all the resources and code of the application. The primary components of the APK file structure include AndroidManifest.xml, classes.dex, the res folder, and the lib folder.

Decompilation is the process of restoring an APK file to more human-readable source code. Decompilation tools can parse the APK file, extract Java class files, and convert them back into Java source code. This allows developers and security researchers to analyze the application's logic, algorithms, and implementation details. Decompilation aids in understanding the internal workings of the application, discovering potential vulnerabilities and security issues, and conducting reverse engineering research and analysis. However, it's important to note that the results of decompilation may not be identical to the original code, as compiler optimizations and code obfuscation techniques can result in partial information loss or obfuscation. [4]

(2) Dynamic Analysis and Static Analysis Techniques

In reverse engineering, dynamic analysis and static analysis are two commonly used technical approaches. Dynamic analysis involves running the application and monitoring its behavior to gather information. It entails using debuggers, emulators, or specialized tools to dynamically execute the application and collect data about code execution paths, input-output data, network communication, system calls, and more. Dynamic analysis helps researchers understand the application's runtime behavior, detect malicious activities and vulnerabilities, and acquire runtime information for debugging and analysis.

Static analysis, on the other hand, involves obtaining information about the application without actually running it. It entails analyzing the application's binary files, source code, or decompiled results to understand its structure, algorithms, logic, and potential vulnerabilities. Static analysis assists researchers in identifying security issues, comprehending the application's operational principles and business logic, and conducting code reviews and vulnerability analysis.

(3) Automation Tools

In the field of Android reverse engineering, numerous automation tools are used for analyzing and researching applications, system vulnerabilities, and security issues, such as APKTool, Jadx.

4. Android Reverse Engineering Methods and Techniques

(1) Vulnerability Exploitation and Reverse Vulnerability Analysis

In the field of Android reverse engineering, vulnerability exploitation and reverse vulnerability analysis stand as

pivotal research areas. The primary objective here is to identify and comprehend these vulnerabilities while developing corresponding attack techniques or defense measures.

Vulnerability exploitation primarily involves two aspects: vulnerability discovery and exploitation technique development. Vulnerability discovery entails finding vulnerabilities within applications or the operating system through reverse engineering and code analysis. This may encompass the discovery and exploitation of common vulnerability types such as buffer overflows, integer overflows, and code injection. After identifying vulnerabilities, researchers develop exploit code to achieve their attack objectives. This could involve crafting specific shellcode, Return-Oriented Programming (ROP) chains, or leveraging memory layouts, among other techniques.

Reverse vulnerability analysis primarily comprises disassembly and reverse engineering, dynamic debugging, and vulnerability testing. Disassembly and reverse engineering involve dissecting applications or operating systems through disassembly and reverse engineering processes to analyze their code and data structures. This helps in understanding the root causes of vulnerabilities and potential attack vectors. Dynamic debugging and vulnerability testing involve conducting dynamic debugging and vulnerability testing within controlled environments to validate the existence of vulnerabilities, assess their potential impact, and analyze possible attack vectors.

(2) Software Protection Mechanism Analysis

In Android reverse engineering, analyzing software protection mechanisms is a crucial task. It involves identifying and evaluating the protective measures employed within applications while uncovering potential vulnerabilities or weaknesses. Software protection mechanism analysis encompasses four main aspects:

1) Anti-debugging and Anti-tampering Techniques: Analyzing the anti-debugging and anti-tampering techniques used within applications, such as debugger detection, code obfuscation, and encryption. The effectiveness and security of these techniques are assessed, and attempts are made to bypass or attack them.

2) Security Verification and Authorization: Analyzing security verification and authorization mechanisms within the application, such as login systems, license validation, and digital signatures. The security and reliability of these mechanisms are evaluated, and efforts are made to bypass or simulate them.

3) Decompilation Techniques: Analyzing the decompilation techniques used within the application, including code obfuscation, decompilation protection, and anti-debugging protection. The level of protection these techniques offer to the application code is assessed, and attempts are made to reverse or analyze the decompiled results.

4) Encryption and Data Protection: Analyzing the encryption algorithms and data protection mechanisms used within the application, such as encrypting sensitive data and managing cryptographic keys. The strength and reliability of these mechanisms are evaluated, and attempts are made to crack or analyze the encrypted data.

5. Practical Applications of Android Reverse Engineering

Now, we will use a remote rehabilitation training app (referred to as the rehab app) as an example to illustrate and analyze the usage of reverse engineering tools such as APKTool, Jadx-GUI, ADB, and penetration testing on the Kali Linux platform. This analysis includes activities like anti-emulator or virtual machine environment analysis, app permission and functionality analysis, algorithm, and vulnerability analysis.

The app is developed on the Android operating system using Android Studio as the development tool. Kotlin is the chosen programming language. In terms of technical frameworks, it utilizes the Jetpack component library to simplify data management, interface updates, and local storage for the rehabilitation program.

Furthermore, it employs the Retrofit library for network requests, facilitating communication with the backend server. Additionally, it integrates the Google Fit API to access and record users' health data, including metrics like step count, heart rate, and sleep patterns.

(1) Analysis of Anti-Emulator or Virtual Machine Environments

The rehabilitation app was run in the LDPlayer emulator, but it displayed an installation failure. After decompiling using Jadx-GUI and APKTool, no emulator detection code was found in the decompiled code. It was confirmed that the app does not employ any software protection mechanisms for detecting debuggers.

(2) Analysis of App Permissions and Functionality

Analysis of permissions and functionality involves reviewing and studying an application's permission declarations, function calls, and API usage. Through this analysis, researchers can understand the permissions and functionality required by the application, assess their reasonableness and security, and identify potential issues such as permission misuse, sensitive data leakage, or functional defects [3].

Based on the source code of the rehabilitation app, it was determined that the app has two types of users: doctors

and regular patients. Doctors are primarily responsible for managing patient rehabilitation training and have access to patient training statistics, doctor-patient communication modules, and doctor information modules. Patients execute the training plans provided by doctors and have access to the rehabilitation training module, rehabilitation training statistics module, doctor-patient communication module, and patient information module. Under the patient interface, core AI recognition algorithms are invoked. Additionally, the app supports login through third-party APIs. The main code framework for implementing the rehabilitation app's functionality is shown in Figure 1.

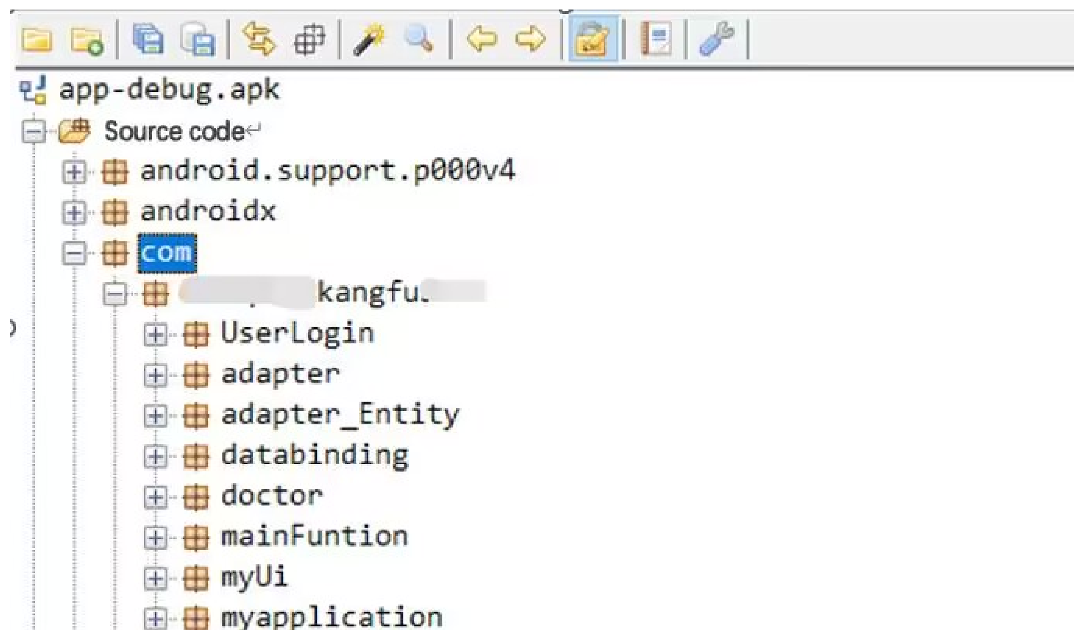

Figure 1. Schematic Diagram of Functional Implementation Code Framework for a Remote Rehabilitation Training App.

(3) Algorithm and Vulnerability Analysis

Algorithm and vulnerability analysis in Android reverse engineering is the process of studying the internal structure, algorithms, and potential vulnerabilities of Android applications. It involves using techniques such as decompilation, static analysis, and dynamic analysis to analyze an application's code and behavior. Through the analysis of algorithms and code, researchers can discover potential security vulnerabilities, such as insufficient input validation, code injection, and permission issues, and provide corresponding fixes and defense measures. The goal of vulnerability analysis is to identify weaknesses that may exist in the application, to improve its security and provide a more reliable application.

1) Invalid Token Login Validation

Token validation is a common authentication method used to verify a user's identity and permissions. Through reverse analysis, it was found that in the rehabilitation app, login validation is performed through a third-party API, and there exists ineffective token validation in the API.

The following is a code snippet that shows the ineffective token validation in the app:

```
public class Login_API {
    // This shows a valid token
    private static final String SECRET_KEY = "126b2ccc9703a6ec1039cd41e2111c774a761227829c29857b68cc6e72c05558";
    public static String generateToken(String username) {
        Date expiresAt = new Date(System.currentTimeMillis() + 3600000); // Set to expire in 1 hour
        Algorithm algorithm = Algorithm.HMAC256(SECRET_KEY);
        String token = JWT.create()
                .withSubject(username)
                .withExpiresAt(expiresAt)
                .sign(algorithm);
```

```
        return token;
    }

    public static boolean verifyToken(String token) {
        return true; // This method directly returns true without validating the token, allowing bypassing of login.
    }
}
```

The above code in the `verifyToken` method directly returns `true` without performing a validity check on the token, resulting in ineffective token validation, thus bypassing the login.

2) Insecure HTTP Protocol Transmission

In the rehabilitation app, for example in the "Doctor Standard Action Utilization Code Implementation" module, the OkHttpClient does not add support for HTTPS, causing Android to directly call API backend interfaces using the HTTP protocol and using the OKHTTP library for interface calls. When transmitting data, the API does not use a secure HTTPS transmission protocol, making it susceptible to interception. Code vulnerabilities need to be fixed, and data transmission security measures should be implemented.

The following is a key code snippet where the rehabilitation app uses the HTTP protocol to call API backend interfaces:

```
fun PostAccounts(
url: String，userName: String，password: String，callback: okhttp3.Callback) {
val client = OkHttpClient()
val mediaType = MediaType.parse( string: "text/json")
val body=RequestBody.create(
mediaType,
content: "(\r\n \"userName\": \"$userName\",\r\n \"password\": \"$password\",\r\n \"email\":***********.com\"\n)"
val request = Request.Builder().url(url).method( method: "PoST", body)
.addHeader( name: "clientId"，TCAC_User.clientId)
.addHeader( name: "Secret"，TCAC_User.Secret)
.addHeader( name: "Content-Type"，value: "application/json").build()
client.newCall(request).enqueue(callback)
```

3) Hook Injection

Hook injection refers to malicious applications or attackers using Android system APIs and mechanisms to tamper with, modify, or intercept the normal behavior of an application. After analyzing the source code of the rehabilitation app, a segment of code related to patient exercise verification and logging under patient user context was found. This code has the potential to execute hook injection, and the injected code looks like this:

```
public class HookedMainActivity extends MainActivity {
    private static final String TAG = "HookedMainActivity";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Log.d(TAG, "onCreate: Hooked!");
        // Intercept and modify the execution of motion verification
        hookMotionVerification ();
    }
    private void hookMotionVerification () {
        // Hook and modify, output custom log information
        // ...
        Log.d(TAG, "hookSensitiveOperation: Executive Training hooked!");
        // ...
    }
}
```

6. Android Defense Recommendations and Measures

(1) Secure API Development

In Android development, API security is of paramount importance. To ensure the security of APIs, developers can take the following measures: Restrict access permissions to sensitive APIs; Use HTTPS for communication; Implement effective authentication and authorization mechanisms; Perform input validation and filtering; Prevent API abuse and reverse engineering; Securely store API keys and credentials; Conduct security testing and audits; Continuously update and patch APIs[5].

Taking HTTPS communication as an example, within the "Doctor Standard Actions Implemented by Software Code" module of the Rehabilitation App, you can modify the configuration of OkHttpClient to include support for HTTPS. Below is the specific modified code for this section:

```
fun PostAccounts(
url: string， userName: String， password: String，  callback: okhttp3.Callback
) {
val client = OkHttpClient()
val mediaType = MediaType.parse("text/json")
val body = RequestBody.create(
   mediaType,
    "{\r\n \"userlame\": \"$suserMame\",\r\n \"password\": \"$password\",\r\n \"email\": \"xxx@163.com\"\r\n}"
)
val request = Request.Builder().url(url).method( method: "POST"， body)
   .addHeader( "ClientId"， TCAC_User.ClientId)
   .addHeader( "Secret"， TCAC_User.Secret)
   .addHeader( "content-Type",value: "application/json")
   .build()
client.newCall(request). enqueue(callback)}
```

(2) Runtime Environment Detection

1) Root Device Detection

Rooted devices may pose security risks as they have elevated privileges and greater system access. Applications can use a Root detection library to check if a device is rooted. The implementation can involve inspecting system files, device properties, or specific Root characteristic files and returning the corresponding result.

For rooted devices, they typically have permission management software and the 'su' executable program. Detection measures for the 'su' executable program are as follows:

Detecting the presence of the 'su' executable program can quickly determine if the device is rooted. The 'su' program is generally located in directories such as /sbin/, /system/bin/, /system/xbin/, or /system/bin/failsafe/. If 'su' is found in any of these directories, it can be inferred that the device is rooted.

2) Simulator or Virtual Machine Environment Detection

Detection in this section can be performed by examining device properties, hardware features, or specific simulator or virtual machine indicators to determine the current environment. The principle is that simulators and physical devices differ in services, properties, files, and hardware. These differences can be used as characteristics for detecting a simulator environment. For virtual machine environments, you can detect the support for the x86 or AMD64 instruction set.

The pseudocode for its detection is as follows:

```
// Detect device hardware information
String deviceModel = Build.MODEL;
String deviceManufacturer = Build.MANUFACTURER;
String processorType = System.getProperty("os.arch");
// Check if hardware information matches known emulator features
boolean isSimulator = checkSimulatorHardware(deviceModel, deviceManufacturer, processorType);
// Check for emulator-related files and directories
boolean hasSimulatorFiles = checkSimulatorFiles();
// Check for emulator-specific system properties
```

```
boolean hasSimulatorSystemProperty = checkSimulatorSystemProperty();
// Check for virtualization instruction set support
boolean hasVirtualizationSupport = checkVirtualizationSupport(processorType);
// Check for emulator or virtual machine indicators
boolean hasSimulatorIndicators = checkSimulatorIndicators();
// Determine if the device is in an emulator or virtual machine environment
boolean isEmulator = isSimulator || hasSimulatorFiles || hasSimulatorSystemProperty || hasVirtualizationSupport || hasSimulatorIndicators;
// Print the result
System.out.println("Emulator: " + isEmulator);
```

(3) Data Transmission Security

Ensure data transmission security by using HTTPS, encrypting data on the frontend using encryption algorithms, decrypting data on the backend, avoiding storing sensitive data in plaintext in code or configurations, and validating and filtering input and output data.

Below is a portion of the code from the Rehabilitation App's API request. This is the key code modified after data encryption measures:

```
// Create an HTTPS connection
URL url = new URL("https://www.xxx.com/api/dbtelerehabilitationsystem");
HttpsURLConnection connection = (HttpsURLConnection) url.openConnection();
// Set SSL context (optional)
SSLContext sslContext = SSLContext.getInstance("TLS");
sslContext.init(null, null, new SecureRandom());
connection.setSSLSocketFactory(sslContext.getSocketFactory());
// Set connection properties
connection.setRequestMethod("POST");
connection.setDoOutput(true);
connection.setDoInput(true);
// Prepare request data
String requestData = prepareRequestData(); // Prepare data to be sent
byte[] encryptedData = encryptData(requestData); // Encrypt the data
// Send encrypted data
OutputStream outputStream = connection.getOutputStream();
outputStream.write(encryptedData);
outputStream.flush();
outputStream.close();
// Get response data
InputStream inputStream = connection.getInputStream();
byte[] responseBytes = readInputStream(inputStream);
inputStream.close();
// Decrypt response data
String responseData = decryptData(responseBytes); // Decrypt the response data
// Process response data
processResponseData(responseData);
// Close the connection
connection.disconnect();
```

(4) Code Integrity Checking

One of the methods for code integrity checks is software signature verification, which is used to verify the integrity and source of Android applications. Every Android application must be signed with a digital certificate to ensure its identity and integrity. Signing is mandatory for each app, and the verification method involves calculating the values of the signatures at runtime and during publication using hash functions and then comparing these values to detect code integrity. For instance, to implement code integrity checks in the Rehabilitation App using the checkCodeIntegrity method, the pseudocode would be as follows:

```
function checkCodeIntegrity() returns boolean:
    // Get the signature value at the time of publication
    string expectedSignature = getExpectedSignature()
    // Get the signature value at runtime
    string actualSignature = getActualSignature()
    // Calculate the hash value of the signature at the time of publication
    string expectedHash = calculateHash(expectedSignature)
    // Calculate the hash value of the runtime signature
    string actualHash = calculateHash(actualSignature)
    // Compare the hash value of the publication signature with the runtime signature
    if expectedHash equals actualHash then:
        return true
    else:
        return false
```

(5) Obfuscation Techniques Enhance Code Security

Android obfuscation techniques increase the complexity and obfuscation of application code using methods such as code compression, symbol renaming, string encryption, and control flow obfuscation. This enhances the difficulty of reverse engineering and improves the security of the application. Obfuscation techniques make application code obscure and challenging to analyze and reverse engineer.

Below is an example of how code compression, symbol renaming, string encryption, and control flow obfuscation can be applied to obfuscate the code discussed in Section 3.3 of this article regarding "Invalid Token Verification" to provide a reencoded version:

```
fun a(
    b: String, c: String, d: String, e: okhttp3.Callback
) {
    val f = OkHttpClient()
    val g = MediaType.parse("a")
    val h = RequestBody.create(gi("j", c, d, "k"))
    val l = Request.Builder().m(b).m("m", h).m("n", "o")
        .m("p", "q").m("r", "s").build()
    f.n(1).o(e)
}
fun i(j: String, c: String, d: String, k: String): String {
    val u = StringBuilder()
    u.append("{\r\n \"v\ ": \"$c\", \r\n \ "w\ ": \"$d\ ", r\n \"x\".:\ "$k\"\r\n}")
    return u.toString()
}
```

7. Conclusion

Reverse engineering technology plays a crucial role in the realm of Android security. Through reverse engineering, we gain insights into an application's inner workings, potential vulnerabilities, and security threats. This provides vital information for security researchers, developers, and app stores to bolster the security of applications. However, reverse engineering also brings forth certain risks and challenges. The continual emergence of malicious applications, the advancement of vulnerability exploitation, and reverse vulnerability analysis place higher demands on application security [5].

Hence, it is imperative to strengthen research in software protection mechanisms, encryption algorithms, and permission controls to enhance an application's resilience against reverse engineering and malicious attacks. The future of Android reverse engineering technology will witness the integration of deep learning, hardware-level reverse engineering, security assessment, and defense. It will also involve the enhancement of decompilation techniques combined with reverse engineering to improve the accuracy and efficiency of malicious application detection and vulnerability analysis.

These developments will further drive the evolution of Android reverse engineering technology, augmenting the

security and resilience of applications.

References

[1] Fengsheng Qiang. Android Software Security Guide [M]. Electronic Industry Press, 2019.

[2] Zhou Shengtao. Android Security Technology Disclosure and Prevention [M]. People's Post and Telecommunications Press, 2015.

[3] Liu Qing, Li Yanlong, Wang Zhongzhong. Research on the detection and analysis technology of Andriod App's illegal collection of personal information behavior [J]. Changjiang Information Communication, 2022 (007): 035.

[4] Wang Guangyu. Research on Android Malware Detection Method Based on Multiple Features [D]. Xi'an University of Electronic Science and Technology, 2019.

[5] Miao Haojian. Research on Data Protection for Android Applications Based on Reverse Engineering [D]. Xi'an University of Electronic Science and Technology, 2018.