verichains

SECURITY AUDIT OF

RICE IOS WALLET

Rice Wallet

Public Report

Mar 17, 2022

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.

EXECUTIVE SUMMARY

This Security Audit Report prepared by Verichains Lab on Mar 17, 2022. We would like to thank Rice Wallet for trusting Verichains Lab in auditing the Rice iOS Wallet. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Rice iOS Wallet. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the application, along with some recommendations.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Rice Wallet

Rice Wallet is a non-custodial wallet that helps you store, invest and manage cryptocurrencies from one place with the best user experience.

Rice Wallet provides users with:

- An easy way to search and evaluate every single DeFi asset on the market.
- Buy & sell DeFi assets at the best prices from selected liquidity pools.
- Keep your portfolio in your pocket. Everything you need to manage your assets is available in a single app.

1.2. About Rice iOS Wallet

Rice iOS Wallet is the iOS version of Rice Wallet which was written in Swift Programming Language.

1.3. Audit scope

In this particular project, a timebox approach was used to define the consulting effort. This means that **Verichains Lab** allotted a prearranged amount of time to identify and document vulnerabilities. Because of this, there is no guarantee that the project has discovered all possible vulnerabilities and risks.

Furthermore, the security check is only an immediate evaluation of the situation at the time the check was performed. An evaluation of future security levels or possible future risks or vulnerabilities may not be derived from it.

This audit was conducted on commit 54b4a823438f2c34e453a886fe10475d0b522941 from git repository *https://github.com/rice-wallet/rice-ios*.

1.4. Audit methodology

Verichains Lab's audit team mainly used the **Open Web Application Security Project (OWASP) Mobile Security Testing Guide (MTSG)**. The **MSTG** is a comprehensive manual for mobile app security development, testing and reverse engineering. It describes technical processes for verifying the controls listed in the **OWASP Mobile Application Verification Standard (MASVS)**. During the audit process, the audit team also used several tools for viewing, finding and verifying security issues of the app, such as following:

#	Name	Version
1	Mobile Security Framework (MobSF)	v3.5.0 beta
2	Frida tools	14.2.13
3	Xcode	13

Table 1. Tools used for audit

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the application functioning; creates a critical risk to the application; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the application with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the application with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 2. Severity levels

1.5. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure application. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The Rice iOS Wallet was written in Swift Programming Language.

The main features of the Rice iOS Wallet are:

- Creating wallets for Ethereum-compatible chains like ETH, BSC and Polygon. Manage wallets, send and receive tokens.
- Swap tokens using 1inch Network.
- Search and evaluate every single DeFi asset on the market.
- Manage portfolio.

2.2. Findings

During the audit process, the audit team found some vulnerabilities in the given version of Rice iOS Wallet.

#	Issue	Severity
1	Unsafe time measurement which can lead to unrestricted passcode bruteforce	CRITICAL
2	Unsafe direct call to keychain	HIGH

Audit team also suggested some possible enhancements.

#	Issue	Severity
1	Unsecure sensitive data store in memory without cleanup	INFORMATIVE

2.3. Issues

2.3.1. PasscodeUtil.swift - Unsafe time measurement which can lead to unrestricted passcode bruteforce CRITICAL

Rice iOS Wallet implements a lock mechanism which locks passcode attempts when someone entered wrong passcode too many times (password guessing/bruteforce attack), and the locked time (60 seconds) is defined as in the snippet bellow:

```
73  // Time in second to allow user tries again
74  func timeToAllowNewAttempt() -> Int {
75      guard let date = self.currentMaxAttemptTime() else { return 0 }
```

```
76      let timePassed = floor(Date().timeIntervalSince(date))
77      return max(0, 60 - Int(timePassed))
78  }
```

Normal date time should not be trusted for time measurement in secure contexts, attackers can modify system datetime to a future timestamp (after release time) to bypass the restriction and continue to try another passcode. This approach does not require jailbreaking and can be automated. After successfully bruteforcing the passcode, the attackers can use it to steal mnemonic seed and private key.

RECOMMENDATION

Instead of Date() based time measuring, it is recommended to implement secure date time measuring using combination of time-synchronization from trusted source and local real-time clocks APIs like mach_absolute_time on iOS. These return the elapsed time since the system was booted, including time when the device goes to deep sleep. This clock is guaranteed to be monotonic and continues to tick even when the CPU is in power saving mode, so is the recommended basis for general purpose interval timing.

UPDATES

- *Mar 17, 2022*: This issue has been acknowledged and fixed by the Rice Wallet team in commit 6747ab57414f6b0da038a59efab0f03a03f7951b.

2.3.2. PasscodeUtil.swift - Unsafe direct call to keychain HIGH

According to well-known still-opening issue on KeychainSwift's GitHub *https://github.com/evgenyneu/keychain-swift/issues/15*, the returned value from this wrapper library can be nil under some unknown conditions. Therefore, following method can return 0 under similar condition:

```
func currentNumberAttempts() -> Int {
        guard let attempts = self.keychain.get(kNumberAttempts) else { re…
  turn 0 }
        return Int(attempts) ?? 0
    }
```

The problem also appear in other functions:

```
func currentMaxAttemptTime() -> Date? {
        guard let maxAttemptTime = self.keychain.get(kMaxAttemptTime), le…
  t double = Double(maxAttemptTime) else { return nil }
        return Date(timeIntervalSince1970: double)
    }
```

```
    // Time in second to allow user tries again
    func timeToAllowNewAttempt() -> Int {
        guard let date = self.currentMaxAttemptTime() else { return 0 }
        let timePassed = floor(Date().timeIntervalSince(date))
        return max(0, 60 - Int(timePassed))
    }
```

Using that behavior, attackers can manage to reset the brute process as long as he can manage to satisfy the required conditions.

More critically, the whole passcode screen can be bypassed:

```
112  fileprivate func didFinishEnterPasscode(_ passcode: String) {
113            guard let currentPasscode = PasscodeUtil.shared.currentPass…
     code() else {
114                self.stop {}
115                return
116            }
```

Relating implementation of the stop function:

```
69  func stop(completion: @escaping () -> Void) {
—   ————————————————————————————————————————————————————————————————
84                } else if case .verifyPasscode = self.type {
85                    self.navigationController.dismiss(animated: true, com…
    pletion: completion)
86                }
87        }
```

As in the above snippet, if currentPasscode() returns nil, stop function will be called, the whole passcode screen overlay will be dismissed, and that's over. This attack flow can be triggered by entering 5 characters of passcode, lock the device for a long enough time, and then unlock it and immediately input the last character. Keychain may not be ready at that time, so it will return nil for currentPasscode() in that case.

RECOMMENDATION

Possible solution already proposed within the issue thread (*https://github.com/evgenyneu/keychain-swift/issues/15*), application need to maintain some flags to be used to trust or not the data returned from keychain. Actually, there is only one case to consider that is when api calls returned zero. Application can keep retrying or delegate until the chain is ready.

UPDATES

- *Mar 17, 2022*: This issue has been acknowledged and fixed by the Rice Wallet team in commit 6747ab57414f6b0da038a59efab0f03a03f7951b.

2.4. Possible enhancements

2.4.1. Unsecure sensitive data store in memory without cleanup INFORMATIVE

Password and private key are stored in-memory using swift's String type, which does not clean up when released back to system, later process *may* read the plain data stored later. Consider checking the article from Apple:

https://developer.apple.com/library/archive/documentation/Security/Conceptual/SecureCodingGuide/SecurityDevelopmentChecklists/SecurityDevelopmentChecklists.html#//apple_ref/doc/uid/TP40002415-CH1-SW6

Scrub (zero) user passwords from memory after validation

Passwords must be kept in memory for the minimum amount of time possible and should be written over, not just released, when no longer > needed. It is possible to read data out of memory even if the application no longer has pointers to it.

UPDATES

- *Mar 17, 2022*: This issue has been acknowledged by the Rice Wallet.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Mar 01, 2022*	Private Report	Verichains Lab
1.1	*Mar 17, 2022*	Public Report	Verichains Lab

Table 3. Report versions history