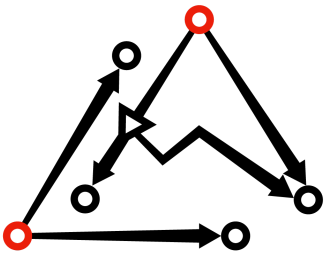

Least Authority

PRIVACY MATTERS

Keychain Module

Security Audit Report

Wallet V

Updated Final Audit Report: 2 April 2025

Table of Contents

- Overview
  - Background
  - Project Dates
  - Review Team
- Coverage
  - Target Code and Revision
  - Supporting Documentation
  - Areas of Concern
- Findings
  - General Comments
    - Code Quality
    - Documentation and Code Comments
    - Scope
  - Specific Issues & Suggestions
    - Issue A: Setting up Wallets Can Result in Race Condition
    - Issue B: Private Keys Are Vulnerable to Supply Chain Attack
    - Issue C: Private Keys Can Migrate To Other iOS Devices Without User Knowledge
    - Issue D: Runtime Error Occurs When Importing A Wallet
    - Issue E: deleteWallet Function Deletes Private Keys for All Wallets
  - Suggestions
    - Suggestion 1: Implement Error Handling
    - Suggestion 2: Use Actively Maintained Dependencies
    - Suggestion 3: Add More Tests
- About Least Authority
- Our Methodology

Overview

Background

Wallet V Labs has requested that Least Authority perform a security audit of the Wallet V. Wallet V is a Web3 wallet that facilitates crypto trading in a simple and cost-effective manner for Global Clients.

Project Dates

- **February 10, 2025 - February 12, 2025:** Initial Code Review *(Completed)*
- **February 13, 2025:** Delivery of Initial Audit Report *(Completed)*
- **March 11, 2025:** Delivery of Updated Initial Audit Report *(Completed)*
- **February 28, 2025:** Verification Review *(Completed)*
- **February 28, 2025:** Delivery of Final Audit Report *(Completed)*
- **March 17, 2025:** Delivery of Updated Final Audit Report *(Completed)*
- **April 2, 2025:** Delivery of Updated Final Audit Report *(Completed)*

Review Team

- Will Sklenars, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Wallet V followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- wallet_audit.js:
  https://github.com/walletv-web3/Audit/blob/main/wallet_audit.js

Specifically, we examined the Git revision for our initial review:

- `511090a1dc5d100e99832b0a858f01cff0790079`

For the verification, we examined the Git revision:

- `cdfea0c11765614139d54992303e94aff3d04444`

For the review, this repository was cloned for use during the audit and for reference in this report:

- https://github.com/LeastAuthority/virgocx-wallet-audit

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Website:
  https://wallet.io

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the wallet;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Malicious attacks and security exploits that would impact the wallet;
- Vulnerabilities in the wallet code, and whether the interaction between the related network components is secure;
- Exposure of any critical or sensitive information during user interactions with the wallet and use of external libraries and dependencies;
- Proper management of encryption and storage of private keys;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team performed a security audit of a core module from the V `react-native` wallet application. The core module is responsible for creating wallets and storing private keys. It is also responsible for signing blockchain transactions. The module interacts with a backend service, which was out of the scope of this audit.

The module supports several wallet creation mechanisms, such as restoring from a private key or mnemonic, or, alternatively, generating a new account. While users also have the ability to delete a wallet, we found that this functionality has been incorrectly implemented, such that deleting a wallet results in all wallets being deleted (Issue E).

When a wallet is created, it is assigned a `walletId`, which is an auto-incrementing integer. To facilitate the increment functionality, the latest `walletId` is stored using `react-native-async-storage`. When a new wallet is created, this ID is incremented and saved back to `react-native-async-storage`. We identified a potential race condition where creating two wallets concurrently could result in a `walletId` collision (Issue A).

The module we reviewed leverages the `react-native-keychain` module for the secure storage of private keys. Our team found the choice of `react-native-keychain` to be an acceptable one, as it is used by several well-audited wallets, such as MetaMask. However, we identified several shortcomings in the configuration and use of `react-native-keychain`, which subjects user private keys to unnecessary risk. We found that user private keys are rendered accessible whenever the device is unlocked (Issue B), and also identified that user accounts can be vulnerable to an attacker who is able to compromise a user's Apple credentials. Furthermore, due to the current `react-native-keychain` configuration, private keys will be backed up to iCloud and downloaded by any other iOS devices the user has, further increasing the attack surface.

Dependencies

The code provided for the audit only shows the dependencies used, and not their versions. Due to this, our team was unable to check the specific dependency versions for vulnerabilities. However, we identified one dependency that is deprecated, and another that has been archived and moved (Suggestion 2).

Code Quality

We performed a manual review of the repositories in scope and found the codebases to be generally organized and well-written. The functions are split up into logical units, and the code is systematically structured, enhancing readability. However, we found that the module appears to represent a work in progress rather than a finished module, as there are several bugs and inconsistencies throughout. For example, lines 245-248 are not contextually relevant within a module, and are presumed to serve as a form of documentation, illustrating how to use key functions within the module. Additionally, we found that importing a wallet will result in a runtime error (Issue D), and that no error handling has been implemented yet (Suggestion 1).

Tests

During our review, we noted that the codebase provided did not include tests and reported this as Suggestion 3, recommending the addition of both manual and unit testing, as Issue D could have been detected through these testing approaches. However, the Wallet V team later clarified that manual testing had been performed prior to submission, with test records maintained separately. Following the initial review, these tests were incorporated into the codebase and subsequently reviewed by our team, thereby resolving the suggestion.

Documentation and Code Comments

There was no documentation provided for the wallet. However, some of the functions have minimal code comments and the module was sufficiently self-documenting; hence, the lack of documentation was not an issue.

Scope

The scope of this review included one module within the wallet system. As a result, the scope was sufficient to assess the implementation of the module but insufficient to evaluate the security of the wallet system as a whole.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Setting up Wallets Can Result in Race Condition	Resolved
Issue B: Private Keys Are Vulnerable to Supply Chain Attack	Resolved
Issue C: Private Keys Can Migrate To Other iOS Devices Without User Knowledge	Resolved
Issue D: Runtime Error Occurs When Importing A Wallet	Resolved

Issue E: deleteWallet Function Deletes Private Keys for All Wallets	Resolved
Suggestion 1: Implement Error Handling	Resolved
Suggestion 2: Use Actively Maintained Dependencies	Resolved
Suggestion 3: Add More Tests	Resolved

Issue A: Setting up Wallets Can Result in Race Condition

Location

`vcx-devs/Audit/wallet_audit.js#L143`

Synopsis

If multiple wallet creation operations run simultaneously, there is a risk of naming collisions due to non-atomic writes.

Impact

Two wallets could be given the same `walletId`. This could cause unexpected behavior when rendering wallets in the application, or when sending blockchain transactions.

Preconditions

Two or more wallets would need to be created concurrently, either by creating a new wallet, or by restoring a wallet using a private key or mnemonic.

Feasibility

As this module is designed to service a mobile application, it is unlikely a user will create wallets in quick succession. Hence, this issue is unlikely to occur.

Technical Details

The `getNextWalletId` function increments the `walletId` value stored in `AsyncStorage`. The increment involves a read, followed by a write with the incremented value. If `getNextWalletId` is called concurrently, two calls may read the same `walletId` value and save the same incremented value `walletId + 1`. This would result in `getNextWalletId` being called twice, although `walletId` would only be incremented by 1 (rather than 2). The flow on effect is that two wallets will be created with the same `walletId`.

Remediation

`AsyncStorage` does not provide support for atomic operations, or a compare-and-set functionality. To remediate this issue, we recommend using a data storage service that provides atomic operations, such as `sqlite`, for which there is a `react-native` implementation. Alternatively, we recommend implementing a locking mechanism in the JavaScript code that rejects any concurrent calls to `getNextWalletId`.

Status

The Wallet V team has implemented a client-side locking mechanism, which rejects concurrent calls to `getNextWalletId`.

Verification

Resolved.

Issue B: Private Keys Are Vulnerable to Supply Chain Attack

Location

vcx-devs/Audit/wallet_audit.js#L11

Synopsis

The private keys are stored in `react-native-keychain`, which provides data encrypted at rest. However, `react-native-keychain` has been insufficiently configured, resulting in keys that are vulnerable when the device is unlocked.

Impact

Since the private keys are always accessible when the phone is unlocked, the application is able to sign transactions without requiring explicit confirmation from the user. Since no explicit confirmation is required, a bug in the wallet code could sign a transaction without the user's awareness. Additionally, any malicious code that infiltrates the application and is imported into the core module–for example, through a supply chain attack–will be able to access the keys at any time. Either scenario could result in the loss of user funds.

Preconditions

The device must be unlocked, and an attacker must be able to inject malicious code into the codebase, either through a supply chain attack, or by an insider who is able to deploy an update.

Feasibility

If the preconditions are met, the extraction of private keys or signing of arbitrary transactions would be trivial.

Technical Details

The module applies the `react-native-keychain` setting `storage`: `Keychain.STORAGE_TYPE.AES_GCM_NO_AUTH`, which the `react-native-keychain` documentation describes as a medium security setting that is suitable for application-level secrets and cached data. With this setting, authorization is not required to access the data. This is insufficient protection for private keys. Furthermore, the keychain is misconfigured with the setting `accessControl`: `Keychain.ACCESS_CONTROL.ALWAYS`. This is incorrect and is equivalent to setting `accessControl`: `undefined`, as the property `ALWAYS` does not exist on the ACCESS_CONTROL enum.

Remediation

We recommend minimizing private key exposure to application code as much as possible without compromising user experience, and requiring user confirmation whenever the private keys are accessed. For the `accessControl` setting, we recommend configuring the keychain to either require a passcode, or biometric input, depending on user preference.
For the `storage` setting, the `react-native-keychain` default is `'Best available storage'`, according to the documentation. We therefore recommend removing the configuration property to use the default setting.

Status

The Wallet V team has updated the keychain configuration to require a biometric or device passcode to access stored data.

Verification

Resolved.

Issue C: Private Keys Can Migrate To Other iOS Devices Without User Knowledge

Location

[vcx-devs/Audit/wallet_audit.js#L19]

Synopsis

Due to the way `react-native-keychain` is configured, user private keys can migrate to another device, which is logged in with the same Apple credentials.

Impact

If a user's Apple credentials are compromised, an attacker may be able to steal the user's funds. Additionally, a user's wallets may automatically migrate to their other devices, which might not align with the user's intentions and expose their funds to unnecessary risk.

Preconditions

The user must have a wallet set up on an Apple device.

Feasibility

As this is the default behavior of the system, both the attack scenario through compromised Apple credentials, as well as the unintended wallet migration scenario, are feasible.

Technical Details

`react-native-keychain` is configured with `accessible: Keychain.ACCESSIBLE.WHEN_UNLOCKED`, rather than the more secure setting `Keychain.ACCESSIBLE.WHEN_UNLOCKED_THIS_DEVICE_ONLY`. By default on iOS, `Keychain` items that are marked `ThisDeviceOnly` do not get backed up or synced to new devices, whereas items that are not marked `ThisDeviceOnly` can migrate via an encrypted backup restore or via the iCloud `Keychain`. Since Android does not follow this pattern, the private keys will not be backed up and migrated by default on Android devices.

Mitigation

We suggest warning current users of the system (iOS only) that wallet migration may have occurred so they can take steps to secure their wallets and Apple accounts.

Remediation

We recommend using the setting `Keychain.ACCESSIBLE.WHEN_UNLOCKED_THIS_DEVICE_ONLY` to prevent private keys from being backed up.

Status

The Wallet V team has updated the configuration to use `ACCESSIBLE.WHEN_UNLOCKED_THIS_DEVICE_ONLY`; therefore, the sensitive data is no longer backed up to cloud services.

Verification

Resolved.

Issue D: Runtime Error Occurs When Importing A Wallet

Location

vcx-devs/Audit/wallet_audit.js#L132

Synopsis

The `createWalletByPrivateKey` function references a property on an object that is not defined in the appropriate scope. This will result in an error at runtime.

Impact

Due to this error, it will be impossible for a user to import a wallet for a given private key.

Preconditions

The user would need to be in possession of the private key for an account they intend to import into the application.

Feasibility

This error will occur every time a user tries to create a wallet for a given private key.

Technical Details

In the `return` statement of the `createWalletByPrivateKey` function, there is the expression `wallet.address`. However, `wallet` is not defined within the `return` statement's scope. Due to this, accessing the address property will result in an error similar to `TypeError: Cannot read property 'address' of undefined`. Furthermore, we note that the Solana and EVM blocks declare different variable names and datatypes.

Remediation

We recommend defining the wallet variable in a scope that the `return` statement has access to. To improve consistency, we suggest updating the EVM and Solana code blocks so that they express a common API. We also suggest performing manual testing on the function, and writing unit tests. Furthermore, we recommend migrating to TypeScript, which eliminates the possibility of such runtime errors occurring.

Status

The Wallet V team has updated the `createWalletByPrivateKey` function, eliminating the runtime error.

Verification

Resolved.

Issue E: deleteWallet Function Deletes Private Keys for All Wallets

Location

vcx-devs/Audit/wallet_audit.js#L185

Synopsis

The `deleteWallet` function takes as an argument the `uuid` of the wallet to be deleted. The function should delete the mnemonic and private key for that wallet only, but instead deletes the private keys for all of the user's wallets.

Impact

After a user deletes a wallet, they will not be able to sign transactions for any of their accounts and will have to set them all up again. If the user does not have their account mnemonics backed up, irreversible loss of funds could occur.

Preconditions

A user would need to have several wallets loaded in the application and have the intention to delete one of them.

Feasibility

Given the preconditions, the issue will occur.

Technical Details

The `deleteWallet` function contains the following statement:

```
for (let i = 0; i < wallets.length; i++) {
    await removeKeychainValue('address_' + wallets[i].address + '_type_' +
wallets[i].generateType)
}
```

The `for loop` iterates over all wallets in the `wallets` array and deletes the private key for each. `wallets` is assumed to be an array of all wallets, although this detail cannot be confirmed from the code available to us.

Mitigation

To mitigate the issue, a user who has deleted a wallet can reimport their other wallets. Alternatively, if users simply use the application with a single wallet, this issue will not occur.

Remediation

When iterating through the array of wallets, we recommend only deleting private keys for the wallet that matches the supplied `uuid`.

Status

The Wallet V team has added a check before deleting the private key so that a private key is only deleted if it matches the `uuid` passed into the `deleteWallet` function.

Verification

Resolved.

Suggestions

Suggestion 1: Implement Error Handling

Location

Throughout the module.

Synopsis

None of the functions within the module handle errors gracefully. When an exception occurs, the software will fail silently, and the user may not be aware that an error has occurred.

Mitigation

We recommend implementing error handling with descriptive messages that can be relayed to the user.

Status

The Wallet V team has improved the code so that each `catch` block now throws an `error` with a descriptive message.

Status

Resolved.

Suggestion 2: Use Actively Maintained Dependencies

Location

`vcx-devs/Audit/wallet_audit.js#L8`

`vcx-devs/Audit/wallet_audit.js#L6`

Synopsis

The `micro-ed25519-hdkey` library has been deprecated, and the `@solana/web3.js` library has been archived and moved.

Mitigation

We recommend replacing these dependencies with other libraries that are being actively maintained.

Status

The Wallet V team has removed the deprecated `micro-ed25519-hdkey` library and replaced it with the actively maintained version, `micro-key-producer`. The archived `@solana/web3.js` library has also been replaced with `tweetnacl-js`..

Verification

Resolved.

Suggestion 3: Add More Tests

Location

`wallet_audit.js`

Synopsis

Currently, there are no tests at all within the codebase. Having good test coverage can help with the early identification of bugs and also facilitate refactoring.

Mitigation

We recommend conducting thorough manual testing and implementing unit tests.

Status

The Wallet V team has implemented a suite of unit tests, which provides adequate coverage.

Verification

Resolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and

unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test

code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.