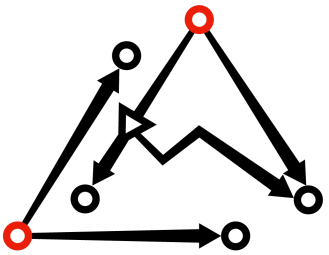

Least Authority

PRIVACY MATTERS

Key Management

Security Audit Report

Joey Wallet

Final Audit Report: 6 June 2025

Table of Contents

Overview

Background

Joey Wallet has requested Least Authority perform a security audit of the key management module of their self-custodial XRP wallet.

Project Dates

- **May 19, 2025 - May 22, 2025:** Initial Code Review *(Completed)*
- **May 23, 2025:** Delivery of Initial Audit Report *(Completed)*
- **June 6, 2025:** Verification Review *(Completed)*
- **June 6, 2025:** Delivery of Final Audit Report *(Completed)*

Review Team

- Paul Lorenc, Security Researcher and Engineer
- Michael Rogers, Security Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of Joey Wallet's key management followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repository is considered in scope for the review:

- Joey Wallet:
  https://github.com/first-ledger/first-ledger-mobile

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

- `586f81fbb65be36b92a3bf8b48df9324b8b9c2eb`

For the verification, we examined the Git revision:

- `01fe1ba589336d2668e46f0fd07bbee218fde1ee`

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

- Joey Wallet:
  https://github.com/LeastAuthority/first-ledger-wallet/tree/audit/xrpl/keys

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:

- N/A.

In addition, this audit report references the following documents:

- `CryptoES` library
  https://www.npmjs.com/package/crypto-es
- `React Native Keychain` library
  https://github.com/oblador/react-native-keychain
- Expo documentation for the `Crypto.randomUUID` function
  https://docs.expo.dev/versions/latest/sdk/crypto/#cryptorandomuuid
- RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace, Section `4.4`
  https://datatracker.ietf.org/doc/html/rfc4122#section-4.4
- `React Native Quick Crypto` library
  https://github.com/margelo/react-native-quick-crypto
- `libsodium.js` library
  https://github.com/jedisct1/libsodium.js
- SEC 1: Elliptic Curve Cryptography, Subsection C.4
  https://www.secg.org/sec1-v2.pdf#subsection.C.4
- CryptoES Cipher Algorithms
  https://github.com/entronad/crypto-es?tab=readme-ov-file#ciphers

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the wallet;
- Attacks that impacts 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 the Joey Wallet's key management, focusing on the proper implementation of key derivation, management, and storage. The lower-level key management primitives include `Deriver.ts`, a class that derives XRPL accounts from various seed sources; `Encryptor.ts`, which handles encryption and decryption using the `CryptoES` library; and `SecureStore.ts`, an interface which uses `Encryptor` and `react-native-keychain` to build an interface which holds keys associated with XRPL accounts in the device's native keychain.

These primitive interfaces are then composed into `KeyBox.ts`, which exports the primary `keyBox` singleton object used by other components in the application to manage XRPL accounts, and interface with the stored keys to perform actions such as signing transactions and creating or deleting additional keys.

Our team examined the Joey Wallet's key management module and found the system to be well-structured, with security clearly prioritized. The team implemented TypeScript interfaces for the core components, dividing the codebase into clear modular units that significantly improve readability. Throughout the codebase, there is constant use of TypeScript modifiers such as `readonly` and `const`, which introduce additional compile-time checks and reinforce expected code behavior. During our audit, we did not identify any issues at the system design level. We recommend that the Joey Wallet team continue with their current approach to composing their key management system, as it reflects a well-considered design and attention to security best practices.

As part of the review process, we also reviewed the implementation for vulnerabilities, coding errors, and adherence to best practice recommendations. We found two low-severity issues relating to the secure store implementation (Issue A, Issue B). We also identified three suggestions related to dependency management (Suggestion 1), code clarity (Suggestion 2), and key derivation (Suggestion 3).

Dependencies

We examined the implemented dependencies in the codebase and identified two developer dependencies with known issues. We recommend improving dependency management (Suggestion 1).

Code Quality

The code is well-organized and clear, with effective use of type declarations to aid readability and detect programming errors. Interfaces have been used for encapsulation and to enable unit testing.

However, our team found that some functions handle error conditions by either returning a null value or throwing an `Error`, requiring their callers to handle both possibilities, which leads to a small amount of unnecessary code.

Tests

The code includes unit tests for key derivation, covering the expected cases as well as error cases. The tests for the encryption code are minimal, testing only that a plaintext can be encrypted and decrypted to produce an identical plaintext. There are no tests for the secure storage code, nor for the utility functions that are used for validation and for deriving an XRPL keypair from a `Web3Auth` private key.

Documentation and Code Comments

There was no documentation provided for the key management code. A lack of documentation hinders the ability to understand the intention of the code, which is critical for assessing the security and the correctness of the implementation. We recommend creating comprehensive project documentation.

In addition to the missing documentation, the code is not heavily commented. However, because the code is well-structured and variable names are clear, the limited number of comments is generally sufficient. In particular, some comments explain the rationale behind the code where needed, such as when platform-specific behaviors must be handled.

Scope

The scope of this review was limited to the key management module. The Joey team was also helpful in assisting our team with building and running the entire codebase to facilitate an understanding of how the

reviewed code is used in context.

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: Encryption Key for Secure Store Has Low Entropy	Resolved
Issue B: Secure Store Uses Unauthenticated Encryption	Resolved
Suggestion 1: Update Vulnerable Dependencies	Implemented
Suggestion 2: Use GetOptions Type When Retrieving Data From React Native Keychain	Implemented
Suggestion 3: Hash Web3Auth Private Key Before Deriving XRPL Keypair	Implemented

Issue A: Encryption Key for Secure Store Has Low Entropy

Location

`xrpl/keys/impl/Encryptor.ts#L7`

Synopsis

The encryption key for the secure store is derived from a random UUID (Universally Unique Identifier), which contains 122 bits of entropy. However, keys for symmetric encryption should contain at least 128 bits of entropy and ideally 256 bits.

Impact

High.

An attacker able to decrypt the information stored in the secure store would gain access to the user's XRPL private keys, giving the attacker control over the user's XRPL accounts.

Feasibility

Low.

To carry out an attack, the attacker would first need to gain access to the encrypted information stored in the secure store, and would then need to exploit the low entropy of the encryption key in combination with a hypothetical cryptanalytic attack that would be infeasible if the key's entropy were higher.

At 122 bits, the entropy of the encryption key is high enough to resist a brute-force attack or any known cryptanalytic attack. Reducing the entropy from the recommended 128 bits to 122 bits merely reduces the safety margin against potential future attacks. Maintaining this safety margin is important for the long-term security of encrypted information, but the reduced margin does not pose any immediate risk.

Severity

Low.

Preconditions

The attacker would need to gain access to the encrypted information stored in the secure store, which is held in the keychain of the user's device. The attacker would then need to carry out a currently unknown cryptanalytic attack against the secure store's AES encryption, aided by the low entropy of the key. As noted above, it is not currently feasible to meet these preconditions, so the issue does not pose an immediate risk.

Technical Details

The encryption key for the secure store is generated by first creating a random `UUID` and then hashing the `UUID` with the `SHA-256` hash function to produce a 256-bit key. The `UUID` is generated using the `randomUUID` function from the `expo-crypto` module, which, according to its documentation, uses a cryptographically strong source of randomness. A `UUID` is a 128-bit value, but RFC 4122 specifies that all UUIDs, including those generated from random values, must have six of their bits set to known values to indicate the format and version of the `UUID`. This leaves 122 random bits, and the 256-bit key contains only 122 bits of entropy rather than the expected 256.

Mitigation

It is not necessary to mitigate this issue in the short term.

Remediation

We recommend that the encryption key contain 256 bits of entropy, which can be obtained from the `getRandomBytes` function of the `expo-crypto` module or any other cryptographically secure source of randomness.

Status

The Joey Wallet team has implemented the remediation as recommended.

Verification

Resolved.

Issue B: Secure Store Uses Unauthenticated Encryption

Location

`xrpl/keys/impl/Encryptor.ts#L20`

Synopsis

The secure store uses the default block cipher mode provided by the `CryptoES` library, which is `AES-256-CBC`. This block cipher mode lacks authentication and is partially malleable, allowing modifications to the ciphertext to pass undetected.

Impact

Low.

The secure store is currently used for storing two kinds of encrypted data: account private keys and mnemonics. Both types of data are represented as strings when in their plaintext form, with private keys represented as hexadecimal strings and mnemonics represented as words separated by spaces.

If an attacker were to modify the ciphertext of an account private key, the modified ciphertext would be unlikely to decrypt to a valid hexadecimal string. An invalid hexadecimal string would not be immediately detected by the key management code, but would cause any subsequent cryptographic operations using the key to fail. There is a small probability of the ciphertext decrypting to a valid hexadecimal string with a value unknown to the attacker. In such a case, it is highly likely the string would correspond to a valid private key, also unknown to the attacker. Any XRPL transactions signed with the private key would be rejected by the network, as the signature would not match the public key included in the transaction, having been generated with the modified private key instead of the original private key.

Thus, regardless of whether the modified ciphertext were to decrypt to a valid hexadecimal string, the only impact of modifying the ciphertext would be that the account becomes unusable. An attacker with access to the secure store could achieve the same impact more simply by deleting or overwriting the entire ciphertext, so the use of unauthenticated and malleable encryption would give the attacker little or no advantage.

Similarly, if an attacker were to modify the ciphertext of a mnemonic, the modified ciphertext would be very unlikely to decrypt to a valid mnemonic, and so the only impact of modifying the ciphertext would be to render the mnemonic unusable, which the attacker could achieve more simply by deleting or overwriting the entire ciphertext.

Our team also considered a second attack in which the attacker would not modify the encrypted values stored in the secure store but would instead swap the encrypted values without otherwise modifying them. For example, the private keys of two accounts could be swapped, so that when the application attempted to retrieve the private key for one of the accounts, the secure store would instead return the other account's private key. Again, the only impact of this attack would be the creation of invalid transactions that would be rejected by the network due to a mismatch between the public and private keys.

Despite the issue's lack of impact on the kinds of data currently stored in the secure store, our team has included it in this report, as it could affect the use of the secure store for storing other kinds of data in the future. In particular, any plaintext up to 16 bytes in length could be modified in a predictable manner by an attacker, due to the partial malleability of the CBC block cipher mode.

Feasibility

Low.

Severity

Low.

Preconditions

The attacker would need to be able to modify values stored in the keychain of the user's device.

Technical Details

By default, the `CryptoES` library uses the `AES-256-CBC` block cipher mode for encryption. This mode lacks authentication, resulting in modifications to the ciphertext going undetected during decryption.

The `AES-256-CBC` mode is also partially malleable: modifications to the first 16 bytes of the ciphertext have predictable effects on the first 16 bytes of the decrypted plaintext. Any such modification also affects every subsequent block of the plaintext. As a result, the overall effect on the plaintext is not fully under the attacker's control when the plaintext exceeds 16 bytes.

Remediation

We recommend replacing the `AES-256-CBC` mode with an authenticated encryption mode, such as `AES-256-GCM`, which is provided by the `react-native-quick-crypto` library, or `XSalsa20/Poly1305`, which is provided by the `libsodium.js` library.

To remove the risk of an attacker swapping encrypted values without modifying them, we recommend including the key[1] under which the value will be stored in the plaintext of the encrypted value, so that it is covered by the authenticated encryption. When retrieving the value, this key should be checked to confirm that it matches the expected key.

Status

The Joey Wallet team has implemented the remediation as recommended.

Verification

Resolved.

Suggestions

Suggestion 1: Update Vulnerable Dependencies

Location

`first-ledger-wallet/package.json`

Synopsis

Analyzing `package.json` for dependency versions revealed two developer dependencies with known issues. The static code analyzer `eslint` is currently locked to version `8.57.1`, which is deprecated. Additionally, the supporting library `eslint-config-standard-with-typescript` has been deprecated in favor of a new package: `eslint-config-love`.

Mitigation

For this specific case, we recommend updating `eslint` to a maintained version, and migrating from `eslint-config-standard-with-typescript` to `eslint-config-love`.

We also recommend adopting a process that emphasizes secure dependency usage to avoid introducing vulnerabilities to the application and to mitigate supply-chain attacks. This process should include:

- Manually reviewing and assessing currently used dependencies;
- Upgrading dependencies with known vulnerabilities to patched versions with fixes;
- Replacing unmaintained dependencies with secure and battle-tested alternatives, if possible;
- Pinning dependencies to specific versions, including pinning build-level dependencies in the `package.json` file to a specific version;
- Only upgrading dependencies upon careful internal review for potential backward compatibility issues and vulnerabilities; and
- Including Automated Dependency auditing reports in the project's CI/CD workflow.

Status

The Joey Wallet team has implemented this suggestion by pinning dependency versions. However, our team notes that three dependencies currently have recently published vulnerabilities:

[1] "Key" is used here in the context of a "key-value store," rather than referring to an "encryption key."

`http-proxy-middleware`, `image-size`, and `vite`. We encourage the Joey Wallet team to regularly review and upgrade dependencies.

Verification

Implemented.

Suggestion 2: Use GetOptions Type When Retrieving Data From React Native Keychain

Location

xrpl/keys/ReactNativeKeychain.ts#L43

Synopsis

The `ReactNativeKeychain` class incorrectly uses the `SetOptions` type for both storing and retrieving data.

Mitigation

We recommend using the `GetOptions` type for retrieving data.

Status

The Joey Wallet team has implemented this suggestion.

Verification

Implemented.

Suggestion 3: Hash Web3Auth Private Key Before Deriving XRPL Keypair

Location

xrpl/keys/utils.ts#L55

Synopsis

A private key from `Web3Auth` is used to derive an XRPL keypair by passing it to the `generateSeed` function of the `ripple-keypairs` library. This function ignores all but the first 16 bytes of the data passed to it. With the current version of `Web3Auth`, this behavior does not pose a risk, as the private key is a raw 256-bit scalar, and the first 16 bytes of the key should contain 128 bits of entropy. However, this format is not documented and may change in the future. If a future version of `Web3Auth` were to use a different format, such as the SEC1 format that is standard for `secp256k1` private keys, the first seven bytes of that format would be predictable, and the seed returned by the `generateSeed` function would only have 72 bits of entropy.

Mitigation

We recommend hashing the `Web3Auth` private key before passing it to the `generateSeed` function, to allow the first 16 bytes of the data passed to `generateSeed` to contain 128 bits of entropy regardless of the format of the private key.

Status

The Joey Wallet team has implemented this suggestion.

Verification

Implemented.

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.