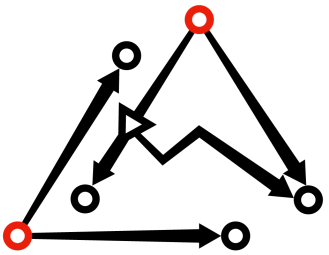

Least Authority

PRIVACY MATTERS

Private Key Storage

Security Audit Report

DropFi

Final Audit Report: 25 August 2025

Table of Contents

Overview

Background

DropFi has requested that Least Authority perform a security audit of the Private Key Storage component of their XRP wallet.

Project Dates

- **August 4, 2025 - August 7, 2025:** Initial Code Review *(Completed)*
- **August 8, 2025:** Delivery of Initial Audit Report *(Completed)*
- **August 25, 2025:** Verification Review
- **August 25, 2025:** Delivery of Final Audit Report

Review Team

- Paul Lorenc, Security Researcher and Engineer
- Dominic Tarr, 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 the DropFi wallet's Private Key Storage component 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:

- https://github.com/dropfi-xrpl/dropfi-wallet-turbo-repo

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

- `9491767e502f86fcab4d4cc3d91d38311710b173`

For the verification, we examined the Git revision:

- `d01dd7108f2529b716d95b35fd391aa3bc216c0e`

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

- https://github.com/LeastAuthority/dropfi-wallet

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:

- Wallet App README:
  https://github.com/LeastAuthority/dropfi-wallet/blob/main/apps/wallet-app/README.md
- Wallet Extension README:
  https://github.com/LeastAuthority/dropfi-wallet/blob/main/apps/wallet-extension/README.md
- TSDoc:
  https://tsdoc.org
- GitHub Advisory Database:
  https://github.com/advisories/GHSA-fjxv-7rqg-78g4

Areas of Concern

Our investigation focused on the following areas:

- Common and case-specific implementation errors;
- Secure key storage and proper management of encryption and signing keys;
- Exposure of any critical information during user interactions;
- Resistance to Distributed Denial of Service (DDos) and similar attacks;
- Vulnerabilities in the code leading to adversarial actions and other attacks;
- Protection against malicious attacks and other methods of exploitation;
- Performance problems or other potential impacts on performance;
- Data privacy, data leaking, and information integrity;
- Inappropriate permissions, privilege escalation, and excess authority; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

DropFi is a wallet for the XRP Ledger (XRPL), implemented in TypeScript and distributed as both a React Native mobile application and a browser extension. DropFi utilizes the WebCrypto API to derive encryption keys from user-supplied passwords and subsequently encrypts the wallet's account data for storage.

On the mobile application, the encrypted account data (ciphertext) is stored using the platform's secure storage API, accessed via the `react-native-encrypted-storage` library. While certain usability trade-offs are present, such as a relatively low number of PBKDF2 iterations, these choices are consistent with common practices adopted by other wallets aimed at broad consumer adoption.

Our audit focused on the key storage and encryption mechanisms implemented in the application. Although we could not identify any issues, we found several areas of improvement that would increase overall security and quality of the implementation.

System Design

Our team found that the codebases for the wallet application and the browser extension duplicate core functionality using different, but very similar, implementations. While we did not identify any functional differences that could introduce errors, we encourage the DropFi team to refactor this logic into a shared module that can be imported by both the application and the extension. This would facilitate maintaining

consistency between the two implementations (Suggestion 1).

DropFi supports the management of multiple private keys (accounts), which are encrypted using a key derived from a user-provided password. Key derivation is performed using PBKDF2 with 100,000 iterations. While this number is relatively low by cryptographic standards, the choice reflects a necessary compromise to ensure acceptable performance on lower-powered mobile devices.

Once derived, the key is used to encrypt account data, and the resulting ciphertext is stored using `react-native-encrypted-storage`, which wraps `EncryptedSharedPreferences` on Android and `Keychain` on iOS.

Even though the account keys are encrypted and the key is strengthened with PBKDF2, an attacker who gains access to the ciphertext could likely brute-force the password, particularly since many users often choose low-entropy passwords for memorability.

The problem of deriving an encryption key from a password is fundamentally different from logging in to a website. If a user database is compromised, an attacker could attempt to reverse hashed passwords. In such cases, the use of a key derivation function increases the computational cost for the attacker and reduces the number of password guesses they can make. While this does not prevent the reversal of particularly weak passwords, the expected value per account is relatively low.

On the other hand, if an attacker gains access to an encrypted wallet account, the potential return per account is much higher due to the value of the cryptocurrency stored in that account, which justifies a significantly higher number of password attempts. Furthermore, when securing a user database, the key derivation function runs on the server, allowing developers to select a high security level and bear the performance cost. However, when the key derivation function is run on the user's device, the parameters must be weak enough to perform reasonably well even on less powerful hardware.

As a result, it should be assumed that users may choose weak passwords and that ciphertext could be decrypted if a moderately sophisticated attacker gains access to it. We have proposed an improvement to mitigate the need for low PBKDF2 settings (Suggestion 3), but emphasize that this is not a substitute for encouraging the use of strong passwords (Suggestion 2).

Dependencies

During our review of the project's dependencies, our team identified a vulnerability in one upstream package. While this package does not appear to be directly used within the in-scope functionality, we recommend it be upgraded or replaced (Suggestion 5).

Code Quality

We found the overall code quality to be satisfactory. The code demonstrates appropriate use of strong typing and is generally well-structured. However, we noted significant duplication between the core logic of the wallet application and the browser extension, which could impact maintainability and increase the risk of inconsistencies.

Tests

The repositories in scope include some tests; however, our team found that the overall coverage is limited. No unit tests were identified for the core code within the scope of this audit. Instead, testing appears to be limited to a single file of integration tests targeting the application as a whole. We recommend implementing a comprehensive test suite (Suggestion 4).

Documentation and Code Comments

The documentation provided was sufficient for the scope of this audit. Although some parts of the codebase include comments that describe specific lines or functionality, there is limited documentation regarding the expected inputs and behavior of classes and functions. We recommend that the DropFi team adopt a standardized commenting format, such as TSDoc, to improve clarity and provide developers with better context when working with TypeScript components.

Scope

The scope of this audit was limited to the key encryption and storage functionality. Even though this represents a core component of the wallet, it constitutes only a small subset of the overall functionality.

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
Suggestion 1: Deduplicate Code Between Wallet Application and Extension	Unresolved
Suggestion 2: Encourage User to Select Strong Password	Unresolved
Suggestion 3: Adjust PBKDF2 Iteration Count Based on Device Capability	Resolved
Suggestion 4: Improve Test Coverage	Unresolved
Suggestion 5: Update and Replace Vulnerable Dependencies	Resolved
Suggestion 6: Remove Logging for Production	Resolved

Suggestions

Suggestion 1: Deduplicate Code Between Wallet Application and Extension

Location

`wallet-app/src/core`

`wallet-extension/src/core`

Synopsis

The core functionality of handling the key derivation, encryption, and decryption is reimplemented in the wallet and the extension, with some small differences. This duplication makes it difficult to determine whether the implementations are functionally identical. Any inconsistencies could introduce compatibility issues, such as failure to decrypt a wallet, which could result in lost funds.

Additionally, the wallet extension core includes utility methods (for example, `arrayBufferToBase64`) that are imported from an external library into the wallet application. The wallet extension also implements utility methods for adding or removing accounts and updating the password, which are absent in the wallet application.

Mitigation

We recommend refactoring the shared functionality into a well-tested library that can be imported by both the wallet application and the extension.

Status

The DropFi team has indicated that they intend to address this issue in a future update.

Verification

Unresolved.

Suggestion 2: Encourage User to Select Strong Password

Location

User interface.

Synopsis

As discussed in the System Design section, the key derivation function alone does not prevent an attacker from discovering the password. It only increases the computational effort required. Therefore, the overall security of the encrypted data remains highly dependent on the user selecting a strong, high-entropy password.

Mitigation

We recommend that the application provide clear, user-facing guidance encouraging the selection of strong passwords. This could include visual password strength indicators and educational messaging about the importance of password complexity. We do not recommend actually preventing the user from deliberately choosing a weak password, as forgetting their own password is a bigger risk in practice.

Status

The DropFi team has acknowledged the issue and plans to refactor it in future versions.

Verification

Unresolved.

Suggestion 3: Adjust PBKDF2 Iteration Count Based on Device Capability

Location

`wallet-app/src/core/Encryptor.ts`

Synopsis

The security of the password-derived encryption key depends largely on the number of PBKDF2 iterations used. In the current implementation, a relatively low iteration count is selected to maintain acceptable performance on low-power devices. However, this results in uniformly low security across all devices, including high-performance ones where higher iteration counts would be feasible without degrading the user experience.

Mitigation

We propose setting the iteration count dynamically based on the power of the device, for example, by selecting the maximum number of iterations that can complete within a target duration (`200` milliseconds). This would allow stronger key derivation on high-powered devices while maintaining usability across a wide range of hardware.

Status

The DropFi team has implemented a system that profiles the time a device requires to perform PBKDF2 and applies stronger settings on more powerful devices.

Verification

Resolved.

Suggestion 4: Improve Test Coverage

Location

`main/test-dapp.html`

Synopsis

No unit tests were identified for the in-scope code. The only test present (`test-dapp.html`) provides limited coverage and lacks thorough validation of core functionality. For example, the `signMessage` test fails only if the `sign` method does not throw an error but does not confirm that valid signatures are correctly generated.

Mitigation

We recommend developing a more comprehensive test suite that includes both positive and negative test cases. Tests should confirm correct behavior for valid inputs and verify that invalid inputs produce appropriate errors.

Status

The DropFi team has acknowledged this issue and plans to implement tests in the future

Verification

Unresolved

Suggestion 5: Update and Replace Vulnerable Dependencies

Synopsis

Running `npm audit` on the codebase reveals that several dependencies are outdated, including at least one with a known critical vulnerability. Although the exploitability of these issues within the current audit scope is unclear, keeping dependencies up to date is a widely recommended practice to reduce the risk of introducing vulnerable code into the project.

Specifically, the `form-data` package includes a critical security issue involving the use of a weak source of randomness. This package does not appear to be directly used in the in-scope functionality but is included as a transitive dependency. Further details can be found in the advisory here.

Mitigation

We recommend updating or replacing the reported dependencies.

Status

The `axios` package was updated to fix this critical vulnerability in the latest version of the mobile app and wallet extension. However, we note that the currently unused UI package still pulls in a vulnerable version of the dependency.

Verification

Resolved.

Suggestion 6: Remove Logging for Production

Location

`wallet-extension/src/pages/content/offscreen.ts`

Synopsis

The current implementation emits logs for debugging purposes here and here. Although this can be beneficial during the development process, it also poses a vector for leaking sensitive data in a production release.

Mitigation

We recommend removing or conditionally disabling debug logging in production builds to avoid exposing potentially sensitive information.

Status

The DroFi team has removed logging for production.

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.