verichains

SECURITY AUDIT OF

Talisman Wallet

Public Report

Mar 12, 2025

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

EXECUTIVE SUMMARY

The Security Audit Report prepared by Verichains Lab on Mar 12, 2025. We would like to thank Talisman for trusting Verichains Lab, delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code of Talisman Wallet. The scope of the audit is limited to the Pull Request provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified **no issues in the source code,** but have some recommendations. Talisman team has acknowledged and have actions to enhance security.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Talisman

Talisman is an Ethereum and Polkadot wallet that makes web3 simple for beginners and unlocks superpowers for pros

1.2. Audit scope

In this Security Enhancement Project, we will consider the following tasks to look for potential vulnerabilities and weaknesses:

- Security review of Source Code and Mechanism Design
- Provide optimization recommendations (if applicable)

Verichains will focus on the Talisman Wallet library, and cryptographic primitives. The audit will cover the following areas:

Cryptography Safety:

- Assess the strength of cryptographic primitives
- Examine the system for any potential information leaks that could lead to guessing, brute forcing, or cryptanalysis of secret materials
- Evaluate the robustness of the cryptographic parameters
- Analyze the quality of the randomness generation mechanism

Code Safety:

- Conduct a thorough examination of the codebase for any vulnerabilities
- Assess input validation mechanisms and identify potential weaknesses
- Evaluate memory management practices to ensure proper handling of sensitive data

By performing this security audit, we aim to identify and mitigate any existing vulnerabilities in Talisman Wallet, enhancing its overall security and ensuring the protection of critical assets. In this project, Verichains will conduct a Cryptography Security Audit on the following repositories, with the understanding that components not included in the scope are considered safe

Component	Repo	Estimated LOC
Update & New Implementation within the following packages: - packages/crypto - packages/keyring - packages/extension-core/domains/keyring	https://github.com/TalismanSociety/talisman/pull/1811 https://github.com/paulmillr/micro-sr25519/blob/main/index.ts	~4000

1.3. Audit Methodology

The security audit process includes three steps:

- Mechanism Design are reviewed to look for any potential problems.
- Source codes are scanned/tested for commonly known and more specific vulnerabilities using public and our in-house security analysis tool.
- Manual audit of the codes for security issues. The source code is manually analyzed to look for any potential problems.

1.3.1. Audit process

Below are overall processes for the Audit service:

Step	Assignee	Description
Step 1: Handle the resource	Talisman	Talisman provides the source code and related documents to Verichains for the audit process.
Step 2: Test & Audit	Verichains	Verichains performs the test and review process, handing to Talisman the detailed reports about the found bugs, vulnerabilities result come with suggestions how to resolve the problem.
Step 3: Bug fixes	Talisman	Talisman has time to check the report, release updates for all reported issues.

Step	Assignee	Description
Step 4: Verification	Verichains	Verichains will double-check all the fixes, patches which related to the reported issues are fixed or not.
Step 5: Publish reports	Verichains	Verichains will create final public reports for the project.

Table 1. Audit process

1.3.2. Vulnerability category

For vulnerabilities, we categorize the findings into categories, depending on their criticality:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the functioning; creates a critical risk; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the code with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the code 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

1.4. Disclaimer

Talisman acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. Talisman understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Talisman agrees that Verichains shall not be

held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the Talisman will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the Talisman, the final report will be considered fully accepted by the Talisman without the signature.

2. AUDIT RESULT

2.1. Overview

The security audit **did not uncover any vulnerabilities in the current implementation.** The codebase demonstrates a strong adherence to best practices in both general security and cryptographic design.

2.1.1. Keyring and Derivation Safety

The key derivation and keyring management mechanisms are generally well-designed and secure in principle.

- **Key Derivation:** The @talismn/crypto package implements robust key derivation for various curves (SR25519, Ed25519, ECDSA, Ethereum, Solana) using industry-standard algorithms and libraries. Substrate-style derivations utilize blake2b256, Ethereum derivation follows BIP32 via @scure/bip32, and Solana derivation adheres to SLIP-0010 using HMAC-SHA512. Hardened derivation is enforced for Substrate and Solana, which is a strong security practice.
- **Keyring Management:** The @talismn/keyring package provides a structured Keyring class to manage mnemonics and accounts. It encrypts sensitive data at rest, utilizes strong key derivation for encryption keys, and employs well-vetted cryptographic libraries. The design promotes secure storage and retrieval of cryptographic secrets.
- **extension-core Integration:** The @talismn/extension-core/domains/keyring module builds upon the @talismn/keyring by adding secure local storage persistence using chrome.storage.local. It introduces mechanisms for password protection, atomic operations, and controlled access to secret keys via withSecretKey and withPjsKeyringPair utilities.

2.1.2. Encryption and Decryption Safety

The encryption and decryption implementation using AES-256-GCM and PBKDF2-HMAC-SHA256 is secure and properly implemented using the browser's crypto.subtle API.

- **Algorithm Choice:** AES-256-GCM is a modern and secure symmetric encryption algorithm. It provides both confidentiality and authenticity (integrity checks). PBKDF2 with SHA-256 is a robust key derivation function suitable for password-based encryption.
- **Parameter Selection:**
  - **PBKDF2 Iterations:** 100,000 iterations is a good starting point and provides reasonable brute-force resistance. Increasing this number further would enhance security but also increase computational cost.
  - **AES Key Length:** 256-bit AES key is used, which is considered very secure.
  - **Salt Size:** 16-byte (128-bit) salt is sufficient for PBKDF2.
  - **IV Size:** 12-byte (96-bit) IV is the recommended size for AES-GCM.
- **Library Usage (crypto.subtle):** The code correctly uses crypto.subtle.importKey, crypto.subtle.deriveKey, crypto.subtle.encrypt, and crypto.subtle.decrypt. It specifies the algorithm names and parameters correctly.

2.1.3. Brute Force Security

The keyring's brute-force security is considered strong due to the use of PBKDF2 with a high iteration count and salt.

- **PBKDF2 Iterations:** 100,000 iterations for PBKDF2 significantly increases the computational cost for attackers attempting to brute-force passwords.
- **Salt Usage:** The use of unique, random salts prevents pre-computation attacks and forces attackers to perform key derivation for each password attempt.
- **Password Complexity (User Responsibility):** Brute-force resistance is ultimately tied to user password strength. The codebase itself does not enforce password complexity, making user education and UI guidance crucial for promoting strong password choices.

2.1.4. Dependency Vulnerabilities

- The codebase relies on well-regarded and actively maintained cryptographic libraries (@noble/curves, @scure/bip32, micro-sr25519, @noble/hashes, @scure/bip39, @polkadot/util-crypto).
- A code review of micro-sr25519 **has been completed, and no vulnerabilities were found**.

2.1.5. Key Encryption Integrity

Keys (mnemonics and secret keys) are properly encrypted before being stored and are only decrypted when necessary with password authentication.

- **Encryption Before Storage:** Sensitive data is consistently encrypted using encryptData *before* being persisted in storage.
- **Decryption on Demand:** Decryption using decryptData is only performed when explicitly required and password-protected, such as when accessing secret keys for signing or exporting mnemonics.
- **Controlled Access Utilities:** The withSecretKey and withPjsKeyringPair utilities provide controlled, temporary access to decrypted secret keys, minimizing exposure and ensuring secure cleanup after use.

The key management practices ensure that sensitive keys are encrypted at rest and are not openly vulnerable. The controlled access mechanisms further reduce the risk of unintended exposure of decrypted secrets.

2.1.6. checkPassword Security

The checkPassword mechanism is securely implemented and does not lower the overall keyring security or allow for reverse engineering of the encryption.

- The Keyring is protected by a password, with the checkPassword method handling verification and initialization by hashing the provided password using oneWayHash (Blake3), encrypting a PASSWORD_CHECK_PHRASE with the hash to store as passwordCheck, and

on subsequent calls (when reset is false), decrypting passwordCheck to verify it matches PASSWORD_CHECK_PHRASE.

- Knowing the plaintext PASSWORD_CHECK_PHRASE and the ciphertext passwordCheck *does not* directly allow reverse engineering of the encryption key used for other secrets (mnemonics, secret keys). This is because passwordHash is derived from the user's password using Blake3 (a one-way hash), and this passwordHash is used as the *password* input to the encryptData and decryptData functions, which in turn use PBKDF2 for deriving the actual encryption key (as in encryption.ts). Therefore, knowing PASSWORD_CHECK_PHRASE and passwordCheck only verifies the correctness of the provided password, not the underlying encryption key derivation process for sensitive data.

2.1.7. Review of Callers of Sensitive Methods in extension-core/domains/keyring

- Here's a breakdown of the files and their roles:
  - **getKeypairTypeFromAccount.ts**: Utility to determine the @polkadot/util-crypto KeypairType based on the Talisman Account type. Used when interacting with Polkadot.js APIs, likely for DApp injection.
  - **getSecretKeyFromPjsJson.ts**: A utility function to extract a secret key from a JSON object encrypted using the @polkadot/util-crypto keyring format (Pkcs8). This is likely used for migration from older versions of the Talisman wallet or Polkadot.js-based keyrings.
  - **migration-utils.ts**: Contains utility functions to map between Talisman KeypairCurve and @polkadot/util-crypto KeypairType enums. Used for migration and interoperability with Polkadot.js ecosystem.
  - **store.ts**: This is the core of the keyring module. It implements the KeyringStore class, which manages the keyring data in local storage.
    - It uses chrome.storage.local to persist the keyring data.
    - It utilizes RxJS Observables (ReplaySubject, Observable) to provide reactive streams of accounts and mnemonics, allowing components to subscribe to keyring changes.
    - It wraps keyring operations in withLock and updateWithPassword/updateWithoutPassword methods to ensure atomic updates and password protection for sensitive operations.
    - It exposes methods to add, get, update, remove mnemonics and accounts, mirroring the API of the underlying @talismn/keyring but adding storage persistence and password handling.
    - It includes methods for password changes, backup, and restore, leveraging the @talismn/keyring's export/import functionalities.
  - **utils.ts**: Provides utility functions related to keyring operations, specifically getNextDerivationPathForMnemonicId. This function helps in automatically deriving the next available account path for a given mnemonic and curve, preventing address collisions.
  - **withPjsKeyringPair.ts**: A higher-order function withPjsKeyringPair that temporarily creates a @polkadot/keyring KeyringPair object for a given Talisman account address. It fetches the secret key from the keyringStore (decrypting it using the current password), constructs a KeyringPair, executes a callback function (cb) with this pair, and then securely cleans up the KeyringPair (locking and removing it). This

is designed to allow interaction with Polkadot.js APIs that expect a KeyringPair object for signing or other operations.
    - **withSecretKey.ts**: Similar to withPjsKeyringPair.ts, but instead of creating a KeyringPair, withSecretKey provides direct access to the decrypted secret key (as Uint8Array) and the curve type to a callback function (cb). It also handles password retrieval, decryption, and secure cleanup (zeroing out the secret key in memory after use).
    - **migrations/migrateFromPjsKeyring.ts**: Implements a migration process to import keyring data from the older @polkadot/ui-keyring format into the new @talismn/keyring-based storage. This migration handles mnemonics, various account types (Talisman keypairs, Polkadot Vault, Ledger, Signet, watch-only, contacts), and attempts to maintain account associations with mnemonics where possible. It includes error handling and progress tracking for the migration process.
- The extension-core/domains/keyring module appears to be well-designed, building upon the secure foundation of the keyring package.
    - **Secure Storage:** Keyring data is stored in chrome.storage.local, which provides browser-level sandboxing and encryption of local storage data. While local storage is not as secure as dedicated hardware security modules, it's a reasonable choice for browser extensions.
    - **Password Protection:** Sensitive keyring operations (adding mnemonics, deriving accounts, accessing secret keys, changing password, backup/restore) are protected by password authentication. The updateWithPassword wrapper ensures that these actions require a valid password retrieved from the passwordStore.
    - **Atomic Operations and Locking:** The #lock mechanism and withLock function aim to prevent race conditions and ensure atomic updates to the keyring data, which is important for data integrity, especially when dealing with asynchronous operations and reactive updates.
    - **Secret Key Handling in withPjsKeyringPair and withSecretKey:** These utility functions are designed to provide *temporary*, controlled access to decrypted secret keys.
        - **Principle of Least Privilege:** They encourage the principle of least privilege by providing the secret key only within the scope of the callback function (cb).

Secure Cleanup: Both functions explicitly perform cleanup operations after the callback execution: withPjsKeyringPair locks and removes the temporary KeyringPair, and withSecretKey fills the secretKey Uint8Array with zeros to erase it from memory. This is a crucial security practice to minimize the time decrypted secrets are exposed in memory.

2.2. Findings

The security audit **did not identify any vulnerabilities in the current implementation**. However, we highlight the following critical assumptions and potential risks that impact the overall security posture:

2.2.1. Brute-Force Protection Depends on Password Strength

The system's resistance to brute-force attacks relies entirely on the strength of the user's password. If users choose weak or easily guessable passwords, an attacker can potentially decrypt sensitive secrets stored in local storage.

RECOMMENDATION

Enforce strong password policies and consider increasing PBKDF2 iterations further if performance allows to enhance brute-force resistance.

UPDATE

- *Mar 11, 2025:* "We intentionally allow weak or easily guessable passwords to ensure a smooth UX during wallet installation. We do however compensate this with a 2 layers of PBKDF2 hashing: User password is hashed with 900,000 iterations to generate a secure keyring password, and this keyring password is hashed with 100,000 iterations to secure each individual keys. **Action**: No changes planned."
- This approach is **justifiable** when the risk of users forgetting their passwords poses a greater threat, potentially leading to the loss of their funds.

2.2.2. Cryptographic Security Depends on Randomness

The security of the cryptographic stack is fundamentally dependent on the quality of random number generation, specifically the browser-provided **crypto.getRandomValues()**. If the environment uses a weak, fallback, or compromised random source, it could severely undermine the confidentiality and integrity of all cryptographic operations.

RECOMMENDATION

Ensure the application fails securely or alerts users in environments lacking secure randomness, such as older browsers or non-standard runtimes.

UPDATE

- *Mar 11, 2025:* "The keyring indeed relies on the browser's native crypto module. Since the Talisman wallet is restricted to Chromium (v102+) and Firefox (v109+), it operates within a controlled environment where these cryptographic APIs are considered secure. **Action:** Added a note in the @talisman/keyring README to caution against using the library in unsupported environments: https://github.com/TalismanSociety/talisman/pull/1811/commits/1ea01e1cf2acd38e4f516fba82a93a24c97a3d5b"

2.2.3. Third-Party Cryptographic Libraries

The implementation makes use of third-party cryptographic libraries (@noble/curves, @scure/bip32, micro-sr25519, @noble/hashes, @scure/bip39, @polkadot/util-crypto), which serve as critical components of the overall security. The correctness, constant-time guarantees, and maintenance of these libraries are external dependencies. Any vulnerabilities, supply-chain attacks, or insecure updates in these libraries could compromise system integrity.

RECOMMENDATION

Regularly monitor third-party cryptographic dependencies. Pin versions, validate the source of WebAssembly builds (if used).

UPDATE

- *Mar 11, 2025:* "We acknowledge that dependency versions were not explicitly pinned. Though for this new keyring dependencies were carefully selected to exclude WebAssembly-based implementations. **Action**: We updated @talismn/crypto to pin dependencies versions, which required a small code change because of a recent update in micro-sr25519 https://github.com/TalismanSociety/talisman/pull/1811/commits/d72d16a3dd48d3c77e843d4808e9100e49bf3370"

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Mar 11, 2025*	Public Report	Verichains Lab

Table 3. Report versions history