

Least Authority

PRIVACY MATTERS

Safely Wallet

Security Audit Report

Tread Safely Inc.

Updated Final Audit Report: 29 September 2026

Table of Contents

- Overview
  - Background
  - Project Dates
  - Review Team
- Coverage
  - Target Code and Revision
  - Supporting Documentation
  - Areas of Concern
- Findings
  - General Comments
    - System Design
    - Dependencies
    - Code Quality
    - Documentation and Code Comments
    - Scope
  - Specific Issues & Suggestions
    - Issue A: Custom Scheme Deep Link Removes Safely's Lock Screen
    - Issue B: Android Back Removes Safely's Lock Screen and Reveals Protected Content
    - Issue C: Mnemonic May Be Recovered From Keyboard's Text Prediction
    - Issue D: Device Onboarding Is Vulnerable to Man-in-the-Middle (MITM) Attack by Server
    - Issue E: Temporary Cryptographic Secrets Are Not Explicitly Cleared After Use
    - Issue F: Watch-Only Import Does Not Communicate Its BIP84 Requirement
  - Suggestions
    - Suggestion 1: Use Locale-Insensitive String Comparisons in the CRDT
    - Suggestion 2: Specify How the CRDT Should Handle Extreme Timestamps
    - Suggestion 3: Improve Test Coverage
    - Suggestion 4: Avoid Logging Sensitive Information
    - Suggestion 5: Fail Closed If Ledger Application Version Cannot Be Determined
    - Suggestion 6: Add an Automated Dependency Advisory Gate
- About Least Authority
- Our Methodology

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

Overview

Background

Tread Safely Inc. has requested that Least Authority perform a security audit of Safely Wallet, a cross-platform self-custody wallet focused on cryptographic key management for cryptocurrency wallets and other use cases. The initial release focuses only on Bitcoin for iOS and Android and includes a limited feature set.

Project Dates

- **July 29, 2026 - August 25, 2026:** Initial Code Review *(Completed)*
- **August 27, 2026:** Delivery of Initial Audit Report *(Completed)*
- **September 21, 2026:** Verification Review *(Completed)*
- **September 22, 2026:** Delivery of Final Audit Report *(Completed)*
- **September 29, 2026:** Delivery of Updated Final Audit Report *(Completed)*

Review Team

- Frankie Pangilinan, 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 the Safely Wallet followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

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

- `Safely`: https://github.com/andreevco/Safely
    - Features and scope
        - Client software source only.
        - iOS and Android targets.
        - Bitcoin seed phrase generation and import (BIP39).
        - Balance and transaction history display.
        - Watch-only accounts.
        - Bitcoin sending and receiving.
        - End-to-end encrypted cross-device account synchronization (seed phrases, address book entries, wallet metadata).
        - Multiple wallets and accounts.
        - Support for Ledger hardware wallet (optional, subject to the available timeframe).
        - Ledger integration.
        - On-ramps.
        - On-device secret storage.
        - Dependencies and supply chain.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

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

- `790cf26814444ea1d8061f34dacf57ae99e3d822`

For the verification, we examined the following branch and the Git revision:

- https://github.com/andreevco/Safely/tree/chore/audit-fixes
- `c592a4af762350e75101ee52a11cb44ee3d9a628`

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

- `Safely`:
  https://github.com/LeastAuthority/Safely

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:

- In-repository documentation for the CRDT and synchronization protocol, including the threat model:
  - https://github.com/LeastAuthority/Safely/tree/master/packages/sync/doc
  - https://github.com/LeastAuthority/Safely/tree/master/packages/slottree/docs

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

Safely is a cross-platform, self-custodial Bitcoin wallet for iOS and Android. It supports `BIP39`-based software wallets, `watch-only` accounts, Ledger hardware-wallet accounts, multiple wallets and

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

accounts, balance and transaction history display, Bitcoin sending and receiving, and integrations with external onramp providers.

The wallet supports synchronization of wallet data between multiple mobile devices. The synchronization protocol is encrypted end-to-end, with the Safely server acting as an untrusted relay. The server also provides an API for querying the Bitcoin blockchain, fetching fee rates and currency exchange rates, and publishing transactions.

System Design

Our team reviewed the overall system design and the areas outlined below and found that they demonstrate deliberate attention to security, including in areas where issues were identified.

Screen Locking and Capture Prevention

The application centralizes lock state. When the application launches from a terminated state, it correctly selects `LockScreen` when the passcode and application lock are enabled. Recovery phrase display is wrapped in capture prevention UI, which is useful against screenshots and previews shown in the operating system's application switcher even though it cannot prevent direct foreground viewing after a navigation bypass.

We reviewed application state transitions, Android system Back handling, the lock route, capture prevention, and mounted routes containing recovery phrases. Android Back can remove the lock route without authentication and reveal a protected screen retained below it (Issue B). A related deep link path can return to protected navigation while the application still considers itself locked (Issue A).

Secret Lifetime

The sync design distinguishes sensitive keys from operational keys and documents their storage and authorization expectations. Several cryptographic services already attempt to overwrite temporary buffers, demonstrating awareness of secret lifetime. `MasterKeyService` provides a notable example because it clears the fetched key after both successful and failed operations.

Key Derivation and On-Device Storage

For generated software portfolios, `PortfolioMnemonicFactory` derives 128 bits of domain-separated entropy from account master key material and a portfolio index, then encodes it as `BIP39` words. Imported and generated mnemonics are validated, encrypted through `MnemonicVault` with `VaultKey`, and stored in synchronized portfolio state as ciphertext. Bitcoin account keys are derived locally with `BIP32` under the native SegWit account policy.

The mobile storage layer separates standard `MMKV` state from two SecureStore-backed tiers configured with different device lock accessibility levels. This allows account metadata and synchronous application state to remain separate from encrypted and security-sensitive values held through the operating system Keychain or Keystore abstraction.

We traced BIP39 wallet creation and import, account key derivation, Vault encryption, Keychain and Keystore storage, software signing, and cleanup of temporary key material. We did not identify any defects in the supported key generation or derivation algorithms. However, some temporary onboarding, operational, and signing secrets are not cleared on every terminal path (Issue E).

Input Handling

The application handles external input carefully, with Zod parsers used effectively. Strings representing currency amounts are parsed with attention to internationalization issues. We suggest adding some tests for this code, which is clear and well-structured but necessarily complex (Suggestion 3).

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

Our assessment covered recovery phrase entry, Bitcoin URI handling, QR input, and currency amount parsing and normalization. Amount handling is carefully structured, but recovery phrase entry does not disable keyboard prediction and may leave mnemonic words in keyboard state (Issue C).

Account and Device Management

When used on a single device, the application can operate without creating an account on the Safely server. An account is created automatically and anonymously when needed. Additional devices are added to the account via an onboarding protocol, in which a new device displays a QR code to be scanned by any of the user's previously enrolled devices.

We traced QR onboarding, X25519 agreement, onboarding key derivation, encrypted `MasterKey` transfer, device authorization, revocation, and reconnect behavior. We found that the onboarding protocol does not authenticate the inviter to the receiving device if a server attacker also learns the QR code (Issue D).

Synchronization Protocol

A CRDT is used to synchronize the state of the wallet between the user's devices. The devices use a shared logical clock, with conflicting writes being resolved automatically using last-writer-wins semantics. The logical clock is persisted on each device as part of the CRDT state, so it advances monotonically even if the device's system clock does not.

The CRDT supports multiple schema versions, with each device automatically projecting wallet state updates to all known schema versions.

Snapshots of the CRDT state are relayed between devices via the Safely server, which maintains a chain of snapshot hashes to ensure that each device downloads and merges all previously published snapshots before publishing its own snapshot.

Snapshots are encrypted end-to-end, keeping the wallet state confidential from the server and any on-path adversaries. All synchronization API requests are signed so that the server can authenticate them.

The CRDT and synchronization protocol are well-designed and well-documented. There are unit and end-to-end tests covering the happy path and some edge cases. We suggest adding end-to-end tests for scenarios involving divergent clocks (Suggestion 3). We also suggest using locale-insensitive string comparisons in the CRDT so that devices with different locales converge on the same state (Suggestion 1).

The synchronization protocol is not designed to handle compromised or malicious devices. Any device can make irrevocable changes to the wallet state. According to the threat model, the expected recovery process in case a device is compromised is to create a new account (using backed-up seed phrases) and enroll the remaining uncompromised devices. Devices that no longer need to participate in synchronization can be revoked.

Bitcoin Transaction Processing

The client obtains unspent transaction outputs and fee estimates through the Bitcoin API, then constructs the recipient output, change output, and PSBT locally. Software portfolios derive the required input keys and sign in the client. Ledger portfolios add the expected account derivation metadata and delegate signing to the hardware device. The broadcast request contains only the finalized serialized transaction. The reviewed transaction flow does not send software wallet private keys or Ledger private key material to the backend. Ledger integration uses the Ledger SDK and is generally well-implemented, with one area identified for stricter error handling (Suggestion 5).

We also examined recipient and change output construction, UTXO handling, fee estimation, software and Ledger signing, transaction confirmation, and broadcast. Transaction outputs are constructed locally from

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

the confirmed request, and we did not identify a separate path that sends wallet private keys to a backend or signs an attacker-selected output. The watch-only extended key form accepts an `xpub` without enforcing or clearly communicating its required `BIP84` account policy (Issue F).

Third-Party Onramp Integration

Onramps and offramps for exchanging fiat currency for Bitcoin are provided by third parties. These are integrated into the mobile application by fetching a list of providers from the Safely server. A provider's website can be configured to open in an in-app browser, which uses an Expo native module based on SFSafariViewController on iOS or Custom Tabs on Android.

Purchases and sales are reported to the Safely server and can subsequently be fetched by the mobile application for display in the account's activity history. As with the synchronization protocol, the API requests used for onramp integration are signed so that the server can authenticate them.

Dependencies

Automated scanning of the complete `pnpm` dependency graph reported four critical, 48 high, 19 moderate, and two low advisory records. Tracing the affected dependency paths revealed no critical or high advisories that could be triggered by the released wallet at runtime. The affected dependencies were limited to development and build tooling, React Native development code, or runtime packages for which Safely does not meet the required trigger conditions.

The analysis included the complete `pnpm` graph, native dependency manifests represented in the repository, package lifecycle policy, CI configuration, and advisory reachability.

Code Quality

We performed a manual review of the repositories in scope and found the code to be of high quality and well organized. The codebase is divided into modules with clearly defined scopes, with type safety and encapsulation used effectively throughout. In addition, naming is clear, and libraries are used appropriately to avoid reimplementing standard functionality, including state machines and parsing of structured input.

Tests

The repositories in scope include unit tests for many of the modules, as well as end-to-end tests using a mocked server for the synchronization protocol. The pinned workspace compiled successfully, lint completed without errors, and 687 JavaScript and TypeScript tests passed during the initial review. Seven native iOS PBKDF2 and `BIP39` vectors also passed.

Documentation and Code Comments

The project documentation clearly describes the CRDT and the synchronization protocol, including generated documentation for the synchronization API. However, the rest of the codebase and API are less thoroughly documented.

Additionally, although code comments are limited, the clarity of the code generally makes additional comments unnecessary.

Scope

The scope of this review was sufficient to assess the security of the mobile client. We recommend performing an additional review of the server code, specifically the components responsible for account creation, device management, snapshot synchronization, and API request authentication.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

Third-party onramp integrations provided by Mercuryo and Revolut were not in scope for this review, with the exception of the client code that interfaces with them.

Specific Issues & Suggestions

We list the issues and suggestions identified during the review in descending order of severity. 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	SEVERITY	STATUS
Issue A: Custom Scheme Deep Link Removes Safely's Lock Screen	Medium	Resolved
Issue B: Android Back Removes Safely's Lock Screen and Reveals Protected Content	Medium	Resolved
Issue C: Mnemonic May Be Recovered From Keyboard's Text Prediction	Medium	Resolved
Issue D: Device Onboarding Is Vulnerable to Man-in-the-Middle (MITM) Attack by Server	Medium	Resolved
Issue E: Temporary Cryptographic Secrets Are Not Explicitly Cleared After Use	Medium	Resolved
Issue F: Watch-Only Import Does Not Communicate Its BIP84 Requirement	Medium	Resolved
Suggestion 1: Use Locale-Insensitive String Comparisons in the CRDT	Informational	Resolved
Suggestion 2: Specify How the CRDT Should Handle Extreme Timestamps	Informational	Resolved
Suggestion 3: Improve Test Coverage	Informational	Resolved
Suggestion 4: Avoid Logging Sensitive Information	Informational	Partially Resolved
Suggestion 5: Fail Closed If Ledger Application Version Cannot Be Determined	Informational	Resolved
Suggestion 6: Add an Automated Dependency Advisory Gate	Informational	Resolved

Issue A: Custom Scheme Deep Link Removes Safely's Lock Screen

Location

`apps/mobile/app.config.js#L3-L12`

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

apps/mobile/src/app/AppNavigation.tsx#L42-L58

apps/mobile/src/app/navigation/index.ts#L49-L90

apps/mobile/src/app/navigation/tabs/index.tsx#L30-L43

apps/mobile/src/entities/security/LockScreenProvider.tsx#L30-L82

Synopsis

When `safely:///tab` is delivered to an already running application after it has completed its background lock transition and presented `LockScreen`, the pinned React Navigation logic removes `LockScreen` and returns to the existing protected `TabsNavigator` without passcode or biometric authentication. Safely continues to record `isLocked` as `true`, but the protected tab interface becomes current. An attacker controlling and viewing the device while it is already unlocked at the operating system level can then view wallet information that the application lock was intended to protect.

Preconditions

For this issue to occur, the following must hold true:

- A passcode must be configured, and the optional application lock must be enabled.
- Safely must already be running, have transitioned to the background, and have completed presenting `LockScreen` above an existing `TabsNavigator`.
- The operating system must deliver the known `safely:///tab` URI to the warm process after another local application opens it or the user follows it.
- Unauthorized observation must require local control or visibility of the device while it is already unlocked at the operating system level. A remote sender does not receive wallet data merely by supplying the link.

Likelihood

Low.

Impact

High.

Severity

Medium.

Technical Details

`LockScreenProvider` records the intended lock state in `isLocked` and presents `LockScreen` when that value becomes `true`. The presenting effect depends on `isLocked` and `isEnabled`, but not on navigation state, so removing the route does not cause it to run again.

The Expo configuration registers the `safely` custom scheme, and the navigation configuration maps `tab` to `TabsNavigator`. For both `safely:///tab` and `safely:///tab/beta`, the pinned React Navigation router reuses the existing `TabsNavigator` and removes `LockScreen` above it. This does not invoke the authenticated `unlock` callback or change `isLocked`, leaving protected navigation current while the provider still records the application as locked.

This path shares the failed application lock invariant described in Issue B but uses an independently triggerable custom scheme entry point, has broader potential platform coverage, and has lower demonstrated impact.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

Mitigation

Until an update is available, we suggest that users rely on the operating system device lock and fully terminate Safely before allowing another person to control or view an unlocked device. Avoiding untrusted `safely:` links reduces user-initiated exposure but does not prevent another local application from invoking the registered scheme. These measures mitigate the risk but do not make the application lock reliable.

Remediation

We recommend keeping protected routes unmounted or otherwise unreachable while `isLocked` is `true` and exposing them only after authenticated unlock. Protected deep links should be rejected or deferred until authentication succeeds. iOS and Android device tests should also be added to cover warm and terminated application states and repeated background and foreground transitions.

Status

The Safely team has followed the recommended deferral of deep links until unlock.

Verification

Resolved.

Issue B: Android Back Removes Safely's Lock Screen and Reveals Protected Content

Location

`apps/mobile/src/entities/security/LockScreenProvider.tsx#L30-82`

`apps/mobile/src/app/AppNavigation.tsx#L42-62`

`apps/mobile/src/app/navigation/index.ts#L74-91`

`apps/mobile/src/screens/LockScreen/LockScreen.tsx#L1-95`

`apps/mobile/src/screens/SecurityScreen/screens/RecoveryPhraseSheet/RecoveryPhraseSheet.tsx#L13-L76`

Synopsis

When the running application is sent to the background, Safely places `LockScreen` above whichever screen is current but leaves that screen in the navigation stack. After the application returns to the foreground, Android's system Back button removes `LockScreen` without requiring a passcode or biometric authentication and reveals the previous screen while Safely still considers itself locked. This applies generally to screens retained below the lock, not only to recovery screens. The highest-impact case is the recovery phrase reveal screen, which is the application's phrase export path: the route retains the plaintext mnemonic, so an attacker with control of and visibility into the unlocked device can record the phrase, restore the wallet elsewhere, and spend the funds it controls. The wallet import screen can likewise return with a phrase that was entered before backgrounding.

Preconditions

For this issue to occur, the following must hold true:

- A passcode must be configured, and the optional application lock must be enabled.
- The process must already be running with another screen current and must then transition to the background, causing `LockScreen` to be presented above that route.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

- The attacker must be able to control and view the unlocked Android device and issue the standard system Back action. Physical interaction was confirmed by the Safely team. Alternate remote-control or Bluetooth input channels were not tested.
- To expose information, the underlying screen must already display or retain that information.
- For recovery phrase compromise, the underlying route must be the recovery-phrase reveal screen, or a wallet import form must still contain a phrase entered by the user. Switching to another application while revealing or importing recovery material can create this state.

Likelihood

Low.

Impact

High.

Severity

Medium.

Technical Details

`LockScreenProvider` sets `isLocked` to `true` when the application backgrounds and presents `LockScreen` above the current route. Its presenting effect depends on `isLocked` and `isEnabled`, but not on navigation state, so removing the route does not cause Safely to present it again.

`LockScreen` is registered in the same root stack as the application's protected screens. Although the route sets `gestureEnabled: false`, the component does not register a React Navigation removal guard or an Android `BackHandler`. React Navigation handles Android's system Back action in JavaScript and calls `navigation.goBack()` when the stack can go back. This removes `LockScreen` without invoking the authenticated `unlock` callback. The retained route then becomes current while `isLocked` remains `true`.

`RecoveryPhraseModal` retains an authenticated plaintext mnemonic in its route parameters, and the wallet import form retains entered text in component state. This issue cannot initiate a new phrase reveal, but it can expose recovery material already retained beneath `LockScreen`.

Mitigation

Until an update is available, we suggest that users rely on the operating system device lock and fully terminate Safely before allowing another person to control or view an unlocked device. The recovery phrase reveal screen and any populated import forms should be closed before backgrounding Safely. These steps reduce exposure but do not make the in-app lock reliable.

Remediation

We recommend rendering only the lock UI while `isLocked` is `true` and keeping all protected navigation content unmounted or otherwise unreachable. Protected screens should be made available only after the authenticated `unlock` callback succeeds.

Status

The Safely team has addressed the reported Android Back bypass by consuming Back events while the lock screen is focused.

Verification

Resolved.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

Issue C: Mnemonic May Be Recovered From Keyboard's Text Prediction

Location

apps/mobile/src/features/add-wallet/SeedPhraseImportForm/SeedPhraseImportForm.tsx#L92

Synopsis

Mnemonic seed phrases entered via the keyboard may be captured by the keyboard's text prediction, allowing all or part of the seed phrase to be recovered outside the application.

Preconditions

To access the keyboard's text prediction state, an attacker must have temporary access to the unlocked mobile device. Any application that allows text input can be used to query the text prediction state. The attacker must guess at least one word of the mnemonic to have subsequent words predicted. If a sufficiently long fragment of the mnemonic can be recovered from the text prediction state, it may be possible to guess the unknown leading and/or trailing bits of the seed by brute force.

Likelihood

Low.

Impact

High.

A successful attack would give the attacker full control of the user's wallet.

Severity

Medium.

Technical Details

A mnemonic seed phrase is composed of words from a well-known word list. These words are individually common, but rarely used in sequence in any other context than seed phrases. Therefore, if a word from the word list is entered on a device's keyboard and another word from the word list is suggested by the keyboard's text prediction, it is likely that the words have been typed in the past as part of a mnemonic seed phrase.

Mitigation

We suggest that users disable the keyboard's text prediction feature and clear the text prediction history.

Remediation

We recommend setting the following properties on all React Native text inputs that handle sensitive text:

- `autoComplete = 'off'`
- `autoCorrect={false}`
- `keyboardType='visible-password'`

Status

The Safely team has implemented the recommended platform keyboard controls, resolving the reported predictive input issue.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

Verification

Resolved.

Issue D: Device Onboarding Is Vulnerable to Man-in-the-Middle (MITM) Attack by Server

Location

packages/sync/src/onboarding/crypto.ts#L74

packages/sync/doc/spec.md?plain=1#L160

Synopsis

When onboarding a new device, an attacker who controls the server and can observe the QR code displayed on the new device's screen can enroll the new device into an account controlled by the attacker, thereby gaining access to any wallet state subsequently created on the new device.

Preconditions

The attacker must control the Safely server and also must be able to observe the QR code displayed on the new device's screen. The user may detect the attack by noticing that the wallet state from the existing device has not been synchronized to the new device or that the new device has not been added to the existing device's list of enrolled devices.

Likelihood

Low.

Impact

High.

If the attack is successful, the wallet state on the new device will be synchronized using keys known to the attacker, allowing the attacker to read and modify the state. In particular, the seeds for any wallets created on the new device will be exposed to the attacker.

Severity

Medium.

Technical Details

To onboard a new device to a user's account, a QR code is shown on the new device's screen and scanned by any of the user's existing devices. This QR code contains the public key of an ephemeral key pair generated by the new device. The existing device that has scanned the QR code generates its own ephemeral key pair, derives the ephemeral shared secret, encrypts the account's master key under a key derived from the shared secret, creates an onboarding message containing the existing device's ephemeral public key and the encrypted master key, and uploads the onboarding message to the server. The new device downloads the onboarding message, derives the ephemeral shared secret, and decrypts the master key.

If an attacker controls the server and can observe the QR code, the attacker can replace the onboarding message with one containing its own ephemeral public key and master key encrypted under the shared secret of its own ephemeral key pair and the new device's ephemeral key pair. The new device will not be able to detect the attack as it does not have any basis on which to authenticate the onboarding message.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

The keys used for encrypting wallet state during synchronization are derived from the master key, so the attacker will be able to decrypt any wallet state created by the new device when it is uploaded to the server for synchronization.

Mitigation

We suggest that users verify that the newly enrolled device appears in the device list on the existing device before performing any account actions on the new device, particularly before creating new wallets.

Remediation

We recommend using a short authentication string to confirm that the new and existing devices have derived the same shared secret before allowing the new device to perform any actions that could expose sensitive information to an attacker.

Status

The Safely team has added an Account ID display to the onboarding UI. The user is instructed to compare the Account IDs on the trusted existing device and the new device to confirm a match before proceeding.

Verification

Resolved.

Issue E: Temporary Cryptographic Secrets Are Not Explicitly Cleared After Use

Location

packages/sync/src/crypto/service/sync-key-service.ts#L8-L23

packages/sync/src/crypto/service/vault-key-service.ts#L8-L23

packages/sync/src/crypto/service/ik-service.ts#L19-L27

packages/sync/src/crypto/service/dmk-signer-service.ts#L29-L37

packages/core/src/entities/seed/bip39-seed-producer.ts#L8-L12

packages/core/src/entities/derivation/btc/implementations/bip39/btc-bip32-node-producer.ts#L20-L30

packages/core/src/entities/signer/btc/btc-keypair-signer.ts#L8-L31

Synopsis

Safely retrieves and derives cryptographic keys into mutable process memory but does not explicitly clear every buffer when its operation ends. Four synchronization services clear fetched keys only after encryption, decryption, or signing succeeds, so an exception skips cleanup. Bitcoin software signing does not clear its BIP39 seed, BIP32 private nodes, or derived input private keys after either success or failure. An attacker with a separate ability to inspect Safely's process memory may recover material that remains available until the runtime reclaims or overwrites it.

Preconditions

For this issue to occur, the following must hold true:

- Safely must perform a synchronization cryptographic operation or sign with a software wallet.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

- To reach the synchronization service failure case, the cryptographic operation must throw after the service retrieves the key.
- The attacker must separately be able to inspect Safely's process memory while the uncleared material remains recoverable, such as through a process memory disclosure, injected code, privileged debugging or instrumentation, a compromised dependency, or device forensic access.
- Runtime behavior must leave the relevant bytes recoverable long enough for inspection. JavaScript garbage collection and memory reuse make this interval unpredictable.

Likelihood

Low.

Impact

High.

Severity

Medium.

Technical Details

`SyncKeyService`, `VaultKeyService`, `IkService`, and `DmkSignerService` clear fetched keys only after their cryptographic operation succeeds. An exception exits before cleanup. `MasterKeyService.withMasterKey` demonstrates the safer pattern by clearing its key after both success and failure.

The Bitcoin software signing path derives a `BIP39` seed, a `BIP32` root and account node, and private input keys. These values are not cleared after signing or an exception, even though the pinned `@scure/bip32` implementation provides `HDKey.wipePrivateData()`.

Garbage collection does not overwrite these values at a defined time. Explicit overwriting in JavaScript remains best effort because the runtime or dependencies may create copies, but clearing buffers owned by Safely removes known copies and shortens the exposure window.

Mitigation

There is no reliable user action that forces Safely's JavaScript heap to clear these buffers. Until the application is updated, we suggest that users avoid running Safely on rooted or jailbroken devices, attaching untrusted debugging or instrumentation tools, or installing untrusted software that can execute within the application process. Fully terminating the application after sensitive operations may reduce the exposure window but does not guarantee that every copy has been overwritten.

Remediation

We recommend using `finally` blocks or a scoped cleanup helper to clear keys used by the affected synchronization services after both successful and failed operations. For software signing, references to the `BIP39` seed and every `BIP32` private node should be retained, with the seed cleared and `wipePrivateData()` called as soon as each value is no longer needed. Cleanup should also be tested after successful operations and forced exceptions, and the documentation should clarify that zeroization in JavaScript is a best-effort measure.

Status

The Safely team has implemented the recommended cleanup, tests, and documentation.

Verification

Resolved.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

Issue F: Watch-Only Import Does Not Communicate Its BIP84 Requirement

Location

packages/core/src/entities/blockchain/btc/btc-network.ts#L6-L8

packages/core/src/blockchain-api/btc/btc-xpub.ts#L9-L77

apps/mobile/src/features/add-wallet/WatchOnlyAddressForm/WatchOnlyAddressForm.tsx#L23-L83

apps/mobile/src/shared/i18n/translations/en.json#L460-L466

packages/core/src/entities/portfolio/portfolio-watch-only.ts#L15-L48

Synopsis

The watch-only import accepts a generic xpub but always derives BIP84 native SegWit addresses. If the key represents a different account policy, the displayed addresses may not be monitored by the source wallet. As a result, funds remain recoverable with the original seed but may appear missing and require manual recovery.

Preconditions

For this issue to occur, the following must hold true:

- A user must import an xpub that represents an account using a script policy other than BIP84 native SegWit.
- The user or another party must send funds to the receive address displayed by the application.
- The source wallet must scan only the script policy associated with the original account.

Likelihood

Medium.

Impact

Medium.

Severity

Medium.

Technical Details

`BtcWalletType` defines only `NATIVE_SEGWIT`, and `BtcXpub.validate` accepts a structurally valid extended key without establishing the account's derivation purpose or output script.

`WatchOnlyAddressForm` accepts xpub and zpub, rejects ypub, and directs the user to use xpub. However, a bare xpub does not communicate whether the source account uses BIP44, BIP49, or BIP84.

After import, `PortfolioWatchOnlyBtc.create` always derives a P2WPKH address using `BtcWalletType.NATIVE_SEGWIT`. Structural validation of the extended key does not establish that this address belongs to the account policy monitored by the source wallet.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

Mitigation

We suggest that users import an explicit address or an extended key that they know represents a `BIP84` native SegWit account.

Remediation

We recommend clearly stating that watch-only extended key import supports only `BIP84` native SegWit accounts and refraining from instructing users to replace `ypub` with `xpub`. For stronger enforcement, confirmation that a bare `xpub` represents a `BIP84` account should be required, or an import format that identifies the intended script policy should be used.

Status

The Safely team has resolved the issue as recommended.

Verification

Resolved.

Suggestions

Suggestion 1: Use Locale-Insensitive String Comparisons in the CRDT

Location

`packages/slottree/src/core/slot-revision.ts#L15`

`packages/slottree/src/core/encoder/cbor/authors.ts#L77`

`packages/slottree/src/core/encoder/cbor/encode.ts#L52`

`packages/slottree/src/core/slots/slot-json.ts#L145`

Synopsis

The CRDT implementation uses local-sensitive string comparisons, which could result in the CRDT state diverging on devices that have different locales configured.

Mitigation

We recommend using locale-insensitive string comparisons whenever consistent results across devices are needed. This can be done by using the default locale-insensitive ordering provided by JavaScript's comparison operators or by specifying a locale to be used for these comparisons on all devices.

Status

The Safely team has implemented the recommended locale-independent comparisons, resolving the suggestion.

Verification

Resolved.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

Suggestion 2: Specify How the CRDT Should Handle Extreme Timestamps

Location

packages/slottree/src/core/merge-protocol.ts#L26

packages/slottree/src/core/encoder/cbor/format.ts#L37

Synopsis

The CRDT uses a logical clock that increases monotonically, so any locally made change is recorded with a higher timestamp than any remotely made change that has been observed so far. However, any finite number, including `Number.MAX_VALUE`, is accepted as a valid timestamp when parsing a snapshot. Once the logical clock has been set to `Number.MAX_VALUE`, subsequent attempts to increment the clock have no effect, breaking the assumption of monotonicity.

Mitigation

The synchronization protocol is not designed to handle malicious or compromised devices. Therefore, we do not suggest requiring the CRDT to continue operating as intended when a device makes extreme timestamp changes. Instead, we recommend documenting the issue and specifying how the CRDT should behave in this scenario (for example, by rejecting the snapshot as invalid).

Status

The Safely team has defined timestamp validation and overflow behavior as recommended, resolving the suggestion.

Verification

Resolved.

Suggestion 3: Improve Test Coverage

Location

packages/core/src/utils/format/pasted-amount-normalizer.ts

packages/slottree/src/core/merge-protocol.ts#L21

Synopsis

While test coverage is generally sufficient, additional testing would be beneficial in several areas.

Mitigation

We recommend adding tests for the code that parses and normalizes strings containing currency amounts. This code is clear and well-structured but necessarily complex.

The end-to-end tests for the CRDT and synchronization protocol cannot test scenarios in which the devices have divergent clocks because the system clock is used directly in the CRDT implementation. We recommend wrapping the clock in an interface that can be replaced for testing, allowing scenarios with divergent clocks to be tested on a single machine.

Status

The Safely team has added the recommended test coverage, resolving the suggestion.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

Verification

Resolved.

Suggestion 4: Avoid Logging Sensitive Information

Location

packages/core/src/blockchain-api/btc/btc-transaction-template.ts#L130

packages/core/src/blockchain-api/btc/btc-transaction-template.ts#L70

packages/core/src/blockchain-api/btc/btc-transaction-template.ts#L82

Synopsis

The application's approach to logging sensitive information demonstrates attention to security. The application uses different logging configurations for development and release environments. In the release environment, all logs are sanitized by detecting and redacting potentially sensitive information, such as long strings of hexadecimal digits that could contain keys or addresses. Some types of information are entirely redacted, while others are partially redacted, leaving a few leading and trailing characters to allow the same piece of information to be matched across log entries.

While this partial redaction is sufficient to prevent keys or addresses from being recovered by brute-force guessing, it may not be sufficient to prevent data from being matched against a list of items of interest, such as transactions or addresses that were active on the blockchain during a certain time period.

Another concern with pattern-based sanitization is that information that has already been partially redacted before logging may not be recognized by the sanitizer's patterns, leading to inconsistent redaction.

Mitigation

We recommend fully redacting sensitive information in the release environment.

As a defense-in-depth measure, we also recommend avoiding sensitive information in error or log messages, even if these messages are expected to be sanitized, so that pattern-based sanitization is relied on only to handle unplanned logging of sensitive information.

Status

The Safely team added logging mitigations but accepted the remaining risk of retaining transaction identifiers and wallet addresses.

Verification

Partially Resolved.

Suggestion 5: Fail Closed If Ledger Application Version Cannot Be Determined

Location

packages/core/src/ledger/ledger-controller.ts#L77

packages/ux/src/features/ledger/machine/actors.ts#L14

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

Synopsis

If a failure occurs while querying the version of the Bitcoin application installed on a Ledger device, the mobile application assumes the version is compatible.

Mitigation

We recommend failing closed in this situation to prevent possible miscommunication between the mobile and Ledger applications.

Status

The Safely team has implemented rejection of operations when the Ledger application version cannot be determined.

Verification

Resolved.

Suggestion 6: Add an Automated Dependency Advisory Gate

Location

`.github/workflows/ci.yml#L9-L89`

`pnpm-workspace.yaml`

Synopsis

Automated scanning of the complete pinned `pnpm` dependency graph reported 73 advisory records consisting of four critical, 48 high, 19 moderate, and two low records. Tracing the affected dependency paths revealed no critical or high advisory that could be triggered through the released wallet at runtime. The affected paths were limited to development and build tooling, React Native development code, or runtime packages for which the required trigger conditions were not met.

The continuous integration workflow installs the frozen lockfile and runs compilation, linting, and tests, but does not run an advisory check. The existing four-day minimum package age and lifecycle script allowlist reduce supply chain risk. However, these controls do not detect advisories affecting approved packages, including vulnerabilities disclosed after packages enter the lockfile. Without an automated baseline, new actionable advisories can be lost among the existing records.

Mitigation

We suggest the following measures:

- Adding dependency scanning to continuous integration and scheduled maintenance workflows, covering the complete `pnpm` graph and the resolved iOS and Android dependency graphs.
- Failing builds when a change introduces a new critical or high-severity advisory or when an approved exception expires, while baselining the existing backlog so the gate does not require unrelated vulnerabilities to be fixed in a single change.
- Recording each exception with the advisory identifier, dependency path, execution context, reachability rationale, owner, and expiration date.
- Upgrading, replacing, or removing affected dependencies where practical, and using overrides only after mobile compatibility and release testing.
- Generating and retaining a software bill of materials and the resolved native dependency graphs for each release.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

Status

The Safely team has implemented delta-based advisory gates covering the complete pinned JavaScript dependency graph and the resolved native dependency graphs.

Verification

Resolved.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

About Least Authority

Founded in 2011 and based in Berlin, Germany, since 2016, Least Authority provides security consulting for privacy-preserving technologies and decentralized systems. We are committed to transparency, openness, and advancing secure digital infrastructure.

We believe that people have a fundamental right to privacy and that secure solutions enable people to use the internet and other connected technologies more freely. Our security consulting services help teams make their solutions more resistant to unauthorized access and unintended system manipulation. We support teams from the design phase through production launch and beyond.

Our team has experience reviewing code for security vulnerabilities and specific attack vectors in multiple languages, including Rust, Go, Python, C, C++, Solidity, Cairo, Circom, Noir, Haskell, OCaml, TypeScript, Swift, ZoKrates, and others. We have reviewed implementations of cryptographic protocols and distributed system architecture in blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, we use various tools to scan code and networks and build custom tools as necessary.

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

Our Methodology

We use a transparent review process that provides the project team visibility into our findings, assumptions, and remediation considerations throughout the audit. The goals of our security audits are to improve the quality of the systems we review and support sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Code Review

In manually reviewing all of the code, we identify potential issues, including those related to code logic, error handling, cryptographic implementations, and correctness with respect to specifications, standards, and best practices. We also evaluate areas where more defensive programming could reduce the risk of future mistakes and accelerate 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.

When appropriate, we also use artificial intelligence (AI) to support the efficiency of our security audits without reducing rigor or reliability. AI-assisted outputs are carefully guided, reviewed, and validated by our team so that conclusions remain accurate and defensible. This supports stronger audit decisions, not only faster review.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and white box penetration testing, supplemented by various tools, including AI, that support the code review process. Discussions with the development team help clarify their vision for the software. Installing and using the relevant software helps us understand user interactions and roles, while also informing our analysis of threat models and attack surfaces. Design documentation, relevant papers, other audit results, similar projects, source code dependencies, and open issue tickets may also be reviewed to inform the analysis. We identify potential vulnerabilities, and, when appropriate, create issue entries. For each entry, we follow the issue investigation and remediation process described below.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and supporting successful remediation. Whenever a potential issue is identified, we raise it with the client through a dedicated secure communication channel to discuss its validity and impact, to the extent these can be determined at that stage. This process allows potentially relevant findings to be reviewed with the client early, before they are documented in greater detail in the initial audit report. For each confirmed issue, we assess impact and feasibility and use those ratings to determine the overall severity, following the OWASP Risk Severity Matrix.

Suggested Solutions

We identify immediate and comprehensive mitigations available to live deployments and then suggest remediation requirements for future releases. When code is already deployed in production, we include mitigation strategies in the report to provide interim support until appropriate remediation is available. The mitigation and remediation recommendations should be reviewed by the development team and deployment engineers. After delivering the Initial Audit Report, we remain available to support implementation of the proposed mitigations and remediations before the verification review.

Development teams should address findings in an order and with an urgency that prioritizes the security of their users. Findings should be addressed as soon as possible, but remediation should be carried out pragmatically and should not cause unnecessary disruption to project timelines.

Although issues must be handled on a case-by-case basis, we work toward a resolution timeline that balances the impact on users and the needs of the project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess whether the issues and suggestions have been sufficiently addressed. When this analysis is complete, we update the report and provide a Final Audit Report that can be published in full. If any critical issues remain, we recommend that the Final Audit Report not be published and that users and other stakeholders be alerted to the impact. We encourage all findings to be addressed and the Final Audit Report to be shared publicly to support transparency and advance security knowledge within the industry.

This Security Audit Report makes no statements or warranties and is for discussion purposes only.