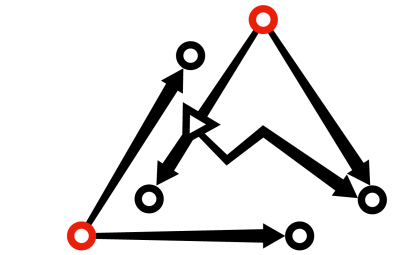

Least Authority

PRIVACY MATTERS

Keystone Hardware Wallet

Security Audit Report

ZCG

Final Audit Report: 20 March 2025

Table of Contents

This audit makes no statements or warranties and is for discussion purposes only.

Overview

Background

As the Zcash Ecosystem Security Lead, ZCG has requested that Least Authority perform a security audit of the Keystone Hardware Wallet, which supports Zcash users managing their assets. This wallet is designed with several security features, including air-gapped communication, open-source firmware, and a user-friendly interface.

Project Dates

- **January 6, 2025 - January 29, 2025:** Initial Code Review *(Completed)*
- **February 3, 2025:** Delivery of Initial Audit Report *(Completed)*
- **March 20, 2025:** Verification Review *(Completed)*
- **March 20, 2025:** Delivery of Final Audit Report *(Completed)*

Review Team

- Poulami Das, Security / Cryptography Researcher and Engineer
- Anna Kaplan, Cryptography Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor and Writer

Coverage

Target Code and Revision

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

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

- Keystone SDK:
  https://github.com/KeystoneHQ/keystone-sdk-rust/tree/0.0.45/libs/ur-registry/src/zcash
  - QRCode protocol definition
- Keystone SDK FFI:
  https://github.com/KeystoneHQ/keystone-sdk-rust/tree/0.0.45/libs/ur-registry-ffi
  - QRCode protocol definition FFI layer for iOS and Android
- Keystone SDK iOS:
  https://github.com/KeystoneHQ/keystone-sdk-ios/blob/0.8.5/Sources/KeystoneSDK/Chain/KeystoneZcashSDK.swift
  - iOS QR protocol wrapper
- Keystone SDK Android:
  https://github.com/KeystoneHQ/keystone-sdk-android/blob/0.7.8/library/src/main/kotlin/com/keystone/sdk/KeystoneZcashSDK.kt
  - Android QR protocol wrapper
- Keystone firmware (Zcash sign process):
  https://github.com/KeystoneHQ/keystone3-firmware/tree/1.8.2/rust/zcash_vendor
- Keystone firmware (Zcash key generation and message signing):
  https://github.com/KeystoneHQ/keystone3-firmware/blob/1.8.2/rust/keystore/src/algorithms/zcash/mod.rs
- Keystone firmware (Zcash API layer):
  https://github.com/KeystoneHQ/keystone3-firmware/tree/1.8.2/rust/apps/zcash/src

This audit makes no statements or warranties and is for discussion purposes only.

- Keystone firmware (wallet export): https://github.com/KeystoneHQ/keystone3-firmware/blob/1.8.2/rust/apps/wallets/src/zcash.rs
- Changes in: https://github.com/zcash/librustzcash/commit/9407f09208d5574a3ba7bf3e6963741114ba77c2
- Changes in: https://github.com/zcash/orchard/commit/e0cc7ac53ad8c97661b312a8b1c064f4cd3c6629

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

- keystone sdk rust: `42f5e05834c395b8d7ce5d1a233b371d2b74fa21`
- keystone sdk android: `34e9a701c52ab56d0844c27055d21fab14a5a434`
- keystone sdk ios: `1d826dff694cc0fbbf457bc05053fd9f6f8fe386`
- keystone firmware: `a162157390c498c1c36212a09cf6c9fcd6d16141`

For the verification, we examined the Git revisions:

- keystone sdk rust: `792eaafc0fb301d453f5ee7a84f7b8f9f90ec31f`
- keystone sdk ios: `bb599d52685d2aa607575f08f378b96f72ef925e`
- keystone firmware: `607c1c1bd404716c0c4c5ead23455b0c803b8a70`

For the review, these repositories were cloned for use during the audit and for reference in this report:

- keystone sdk: https://github.com/LeastAuthority/keystone-sdk-rust
- keystone sdk android: https://github.com/LeastAuthority/keystone-sdk-android
- keystone sdk ios: https://github.com/LeastAuthority/keystone-sdk-ios
- keystone firmware: https://github.com/LeastAuthority/keystone3-firmware

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:

- ZCG Forum: https://forum.zcashcommunity.com/t/keystone-hardware-wallet-support-grant-application/48508/79
- MetaMask and Keystone Mobile Tutorial: https://www.youtube.com/watch?v=ixRIoGfbmTI
- SDK here:
    - https://www.npmjs.com/package/@keystonehq/keystone-sdk 1
    - https://dev.keyst.one/docs/integration-guide-basics/install-the-sdk#webextensionreact-native
- Posts about the development and security of the wallet:
    - https://forum.zcashcommunity.com/t/keystone-hardware-wallet-support-grant-application/48508/25

This audit makes no statements or warranties and is for discussion purposes only.

    - https://forum.zcashcommunity.com/t/keystone-hardware-wallet-support-grant-application/48508/35
    - https://forum.zcashcommunity.com/t/keystone-hardware-wallet-support-grant-application/48508/36
    - https://forum.zcashcommunity.com/t/keystone-hardware-wallet-support-grant-application/48508/42
    - https://forum.zcashcommunity.com/t/keystone-hardware-wallet-support-grant-application/48508/47
    - https://forum.zcashcommunity.com/t/keystone-hardware-wallet-support-grant-application/48508/49
- Website: https://keyst.one

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Vulnerabilities within each component and whether the interaction between the components is secure;
- Key management, including secure private key storage and management of encryption and signing keys;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the execution;
- Protection against malicious attacks and other ways to exploit;
- 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

The Keystone Hardware Wallet acts as a wrapper wallet for an online wallet–in this case, Zashi. To enable the functionalities of the Zashi wallet, the Keystone Wallet team developed a QR code protocol and implemented Zcash functionalities to ensure compatibility within the Keystone Hardware Wallet context. In particular, the Keystone Hardware Wallet allows signing a Zcash *transparent* transaction or an *orchard* transaction.

Our team examined the design of the Keystone Hardware Wallet and found that security has generally been taken into consideration. However, we found areas of improvement that would contribute to the overall security of the system (Suggestion 1, Suggestion 2, Suggestion 3, Suggestion 4, and Suggestion 5).

Our team examined the QR code protocol components (definition, FFI layer, iOS, and Android wrappers), which ensure the correct generation and parsing of the QR code from and to the Keystone hardware, the Zcash signing process, the key generation and message signing process within the Keystone firmware, as well as changes to the `librustzcash` and `orchard` crates related to the PCZT format relevant for Keystone.

This audit makes no statements or warranties and is for discussion purposes only.

During our review, we found that the risk of security attacks, such as signature forgeability and transaction malleability, is largely mitigated due to the air-gapped nature of Keystone operations.

We analyzed the correctness of Keystone operations and the potential for sensitive data leakage and did not identify any critical issues in this regard.

Additionally, we assessed the generation and use of randomness as well as the use of cryptographic building blocks, and did not identify any areas of concern.

Code Quality

We performed a manual review of the repositories in scope and found the code to be clean, well-organized, and in adherence to development best practices.

Tests

The repositories in scope include some tests; however, our team found that test coverage, especially with regards to some of the complex functionalities, can be significantly improved (Suggestion 1).

Documentation and Code Comments

The project documentation provided for this security review, particularly the tutorial for the Keystone Hardware Wallet integration with a software wallet, was insufficient. We recommend that the project documentation be improved to include additional information about the signing workflow, mapped with the codebase. Our team also found that while Sapling is no longer supported, there are some functionalities that still refer to Sapling-related parameters (for instance in `src/pczt/parse.rs`). We recommend clarifying this in the documentation to avoid confusion (Suggestion 3). Additionally, there are insufficient code comments describing security-critical components and functions in the codebase. We recommend improving code comments (Suggestion 2).

Scope

The scope of this review was sufficient and included all security-critical components.

Dependencies

We examined the dependencies implemented in the codebase and found that the Keystone Hardware Wallet code utilizes Zcash dependencies on the `librustzcash` library and the `orchard` crate. The `librustzcash` library and the `orchard` crate in turn use some insecure, unsound, and yanked dependencies. Since these two repositories were out of scope, except for minor code changes related to Keystone as noted here and here, we advise the Keystone Wallet team to actively track changes impacting them and work in close coordination with the development teams of the `librustzcash` library and the `orchard` crate. Additionally, our team checked the correctness of dependencies used in the Zashi Wallet repositories for Android and iOS, and did not identify any issues.

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: Improve Test Coverage	Resolved

This audit makes no statements or warranties and is for discussion purposes only.

Suggestion 2: Improve Code Comments	Resolved
Suggestion 3: Improve Documentation	Resolved
Suggestion 4: Correct Error Messages	Resolved
Suggestion 5: Update Silent Skip in Signature Generation	Resolved

Suggestions

Suggestion 1: Improve Test Coverage

Location

`ur-registry/src` (no tests)

`ur-registry-ffi/src/zcash` (no tests)

`zcash_vendor/src`

Synopsis

The functionalities related to the QR code protocol in `ur-registry/src` and `ur-registry-ffi/src/zcash` contain no tests at all, and those in `zcash_vendor/src` contain insufficient tests.

Mitigation

We recommend adding unit tests to check the correctness of the functionalities in the aforementioned repositories.

Status

The Keystone Wallet team has added correctness tests in PR1652 to `zcash_vendor/src`, `keystone-sdk-rust/tree/master/libs/ur-registry-ffi/src/zcash`, and `keystone-sdk-rust/tree/0.0.45/libs/ur-registry/src/zcash`.

Verification

Resolved.

Suggestion 2: Improve Code Comments

Location

Examples (non-exhaustive):

`ur-registry/src`

`ur-registry-ffi/src/zcash`

`zcash_vendor/src`

`rust/apps/zcash/src`

This audit makes no statements or warranties and is for discussion purposes only.

Synopsis

The code lacks a significant number of explanatory comments. This reduces the readability of the code and, as a result, makes reasoning about the security of the system more difficult. Comprehensive in-line documentation helps provide reviewers of the code with a better understanding and ability to reason about the system design.

Mitigation

We recommend expanding and improving the code comments to facilitate reasoning about the security properties of the system.

Status

The Keystone Wallet team has added extensive code comments to `zcash_vendor/src`, `rust/apps/zcash/src`, `keystone-sdk-rust/tree/master/libs/ur-registry-ffi/src/zcash`, and `keystone-sdk-rust/tree/0.0.45/libs/ur-registry/src/zcash`.

Verification

Resolved.

Suggestion 3: Improve Documentation

Synopsis

The documentation currently lacks sufficient explanation of the QR code protocol and Zcash functionalities of key generation and signing for a Keystone Hardware Wallet.

Mitigation

We recommend creating relevant documentation explaining the QR code protocol and signing mechanism within the Keystone Hardware Wallet. Ideally, this should include mapping the relevant workflows of the QR code protocol, key generation, and signing process with the respective codebase. To avoid confusion, we further recommend clarifying why the Sapling parameters need to be handled in certain parts of the code (`src/pczt/parse.rs`), as the Keystone Hardware Wallet does not currently support Sapling transactions.

Status

The Keystone Wallet team has added documentation to `src/pczt/parse.rs`, along with several `README`s in PR1652.

Verification

Resolved.

Suggestion 4: Correct Error Messages

Location

`KeystoneZcashSDK.swift#L18`

`KeystoneZcashSDK.swift#L26`

Synopsis

The code incorrectly throws a PSBT error instead of a PSZT error.

This audit makes no statements or warranties and is for discussion purposes only.

Mitigation

We recommend updating the code to make it parse and generate a PSZT error instead of a PSBT error.

Status

The Keystone Wallet team has updated the code to generate a PSZT error.

Verification

Resolved.

Suggestion 5: Update Silent Skip in Signature Generation

Location

`algorithms/zcash/mod.rs#L96`

Synopsis

When one of the conditions in the `if` statement does not hold, the function emits an `Ok(())` and silently skips. Consequently, no signature is produced, and no error message is provided.

Mitigation

We recommend updating the `else` case to output an error message instead of an `Ok(())`.

Status

The Keystone Wallet team has updated the code in PR1652 to throw an appropriate error message after the `Ok(())`.

Verification

Resolved.

This audit makes no statements or warranties and is for discussion purposes only.

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.

This audit makes no statements or warranties and is for discussion purposes only.