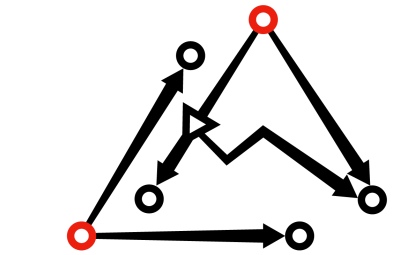

Least Authority

PRIVACY MATTERS

Namada Interface

Security Audit Report

Heliax

Final Audit Report: 3 July 2023

Table of Contents

Overview

Background

Heliax has requested that Least Authority perform a security audit of the Namada Interface and extension for the Anoma Protocol. Anoma is an intent-centric, privacy-preserving protocol for decentralized counterparty discovery, solving, and multi-chain atomic settlement.

Project Dates

- **January 11, 2023 - February 21, 2023:** Initial Code Review *(Completed)*
- **February 23, 2023:** Delivery of Initial Audit Report *(Completed)*
- **June 6, 2023:** Verification Review *(Completed)*
- **July 3, 2023:** Delivery of Final Audit Report *(Completed)*

Review Team

- Jehad Baeth, Security Researcher and Engineer
- Nicole Ernst, Security Researcher and Engineer
- Alejandro Flores, Security Researcher and Engineer
- Mehmet Gönen, Cryptography Researcher and Engineer
- Steven Jung, Security Researcher and Engineer
- Jan Winkelmann, Cryptography Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Namada Interface and extension for the Anoma Protocol followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

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

- Namada Interface:
  https://github.com/anoma/namada-interface

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

- `69e9be0e6a7451bbefdce2ce0c696897475d9dc3`

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

- Namada Interface:
  https://github.com/LeastAuthority/namada-interface

For the verification, we examined the Git revision:

- `43024f4b2cb38d61396286e2640d6f432ab086cd`

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:

- Heliax Website:
  https://heliax.dev
- Namada Website:
  https://namada.net
- Namada Documentation:
  https://docs.namada.net
- Specs:
  https://specs.namada.net
- Anoma whitepaper.pdf

In addition, this audit report references the following documents:

- A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, "RFC 9106: Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications." *IRTF*, 2021, [BDK+21]
- D. Khovratovich and J. Law, "BIP32-Ed25519: Hierarchical Deterministic Keys over a Non-linear Keyspace." *IEEE*, 2017, [KL17]
- D. L. Wheeler, "`zxcvbn`: Low-Budget Password Strength Estimation." *USENIX*, 2016, [Wheeler16]
- Key recovery attack on BIP32-Ed25519:
  https://web.archive.org/web/20210513183118/https://forum.w3f.community/t/key-recovery-attack-on-bip32-ed25519/44
- `Zeroize` Library:
  https://crates.io/crates/zeroize

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the wallet;
- Attacks that impacts funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Malicious attacks and security exploits that would impact the wallet;
- Vulnerabilities in the wallet code and whether the interactions between the related and network components are secure;
- Exposure of any critical or sensitive information during user interactions with the wallet;
- Use of external libraries and dependencies;
- Proper management of encryption and storage of private keys, including the key derivation process;
- Whether the workflow, as it relates to dependencies, is secure;
- Whether the workflow, as it relates to the internal review process of code during the development process, is secure;
- 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

Namada is a Proof-of-Stake (PoS) blockchain that utilizes the Tendermint BFT Consensus. The Namada Interface and Browser Extension composes a multi-asset shielded transfer wallet intended to facilitate user interaction with the Namada blockchain. Our team performed a comprehensive review of the Namada interface and browser extension as well as the dependencies packages in the application repository (with the exception of `masp-web`).

In addition to examining the areas of concern listed above and those shared by the Heliax team, we investigated the key generation mechanism and its susceptibility to key recovery attacks. We also attempted to circumvent the locking mechanism of the wallet extension and could not find a way to achieve this.

Our team found that the architecture of the application and the libraries generally follows security best practice, and the application is generally well-implemented. However, we identified issues in the use of cryptography, in addition to insufficient safeguards implemented to optimize the security of users. We also found the tests, code comments, and project documentation are in need of improvement.

System Design

Our team found that security has generally been taken into consideration in the design of the interface and wallet extension as demonstrated by the detailed areas of concern presented by the Heliax team at the beginning of the code review. Our team identified ways to further optimize the design choices implemented in the system. If the remediations are implemented, this will decrease the vulnerability of users of the wallet to attacks. For example, the wallet allows users to store secret data to the clipboard (Issue C). We also found that the wallet does not implement any functionality to allow users to easily change their password (Issue E), or delete secret data from the device (Issue F).

In addition, our team found that user-selected passwords are not constrained, which would enable users to select passwords that are insufficiently secure. This could make these users' devices vulnerable to brute-force attacks (Issue D). Although the application uses a recommended memory-hard key derivation function, the function is not configured to provide optimal security and could make weak passwords vulnerable to attacks (Issue A).

In addition, our team found a pattern of insufficient zeroization of secret data being implemented to prevent the leakage of secrets (Issue G). We also identified an instance of insufficient input validation to prevent unintended inputs from causing unexpected behavior (Issue H).

To further improve the system design, we recommend implementing a linter into the CI (Suggestion 3).

Code Quality

Our team performed a manual review of the codebase and found it to be well-organized, easy to read, and generally adhering to Rust development best practices. However, our team identified many instances of unused code in the system, which can cause issues with the encryption and the derivation of keys used for encryption. We recommend this code be removed (Issue B).

Tests

The in-scope repository included unit tests, a significant amount of which failed. Sufficient test coverage allows development and security teams to identify implementation errors and verify that the implementation behaves as expected. We recommend making all tests succeed (Suggestion 1).

Documentation

The project documentation provided for this security review was insufficient in describing the general architecture of the system, each of the components, and how those components interact with each other. We recommend that the project documentation be improved to include additional information about dependencies related to Rust and the `OpenSSL` library needed to test and run the project. Moreover, the instructions for installation outlined in the README were not accurate for MacOS, making the process of installation difficult for the auditors. Documentation on how components of the system function serves as a critical reference point that can be compared against what has been implemented in the codebase (Suggestion 2).

Code Comments

During our review, we found that there is a lack of code comments in the browser extension. Furthermore, there are significant amounts of code related to cryptographic functions that do not have any code comments explaining the system design and workflows used. We recommend that code comments be improved (Suggestion 4).

Scope

The scope of this review included all security-critical components of the application. However, our findings show that Namada Interface is still in its early stages of implementation. We recommend that the Heliax team commission a comprehensive security audit of the entire system once development is finalized and design features are complete.

Dependencies

Relevant code for this project, while out of scope for this audit, is included in `namada-interface/packages/masp-web`. This includes the `librustzcash` library, which was previously audited by Least Authority. However, it is not clear if `librustzcash` has been recently audited. As such, we recommend this be reaudited alongside new design and feature implementations.

The Heliax team stated that they want to remove some of the functionality from its current location and implement it into the Software Development Kit (SDK). As a result, we recommend that these changes and implementations be audited by an independent team familiar with the design and implementation of the interface and browser extension, once the functionalities are reimplemented.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Memory Parameter of Argon2 Is Too Low	Resolved
Issue B: Several Cases of Problematic Unused Code	Resolved

Issue C: Seed Phrase Stored in Clipboard	Resolved
Issue D: Usage of Strong Passwords Not Enforced	Resolved
Issue E: No Option To Change User Passwords	Resolved
Issue F: No Option To Delete Wallets	Resolved
Issue G: Secrets Stored in Memory in Rust Code	Resolved
Issue H: Insufficient Sanitization of Parameters	Resolved
Suggestion 1: Make All Tests Succeed (Known Issue)	Resolved
Suggestion 2: Include Additional Instructions for Running and Testing the Project	Resolved
Suggestion 3: Add a Linter Into the CI	Resolved
Suggestion 4: Improve Code Comments	Unresolved

Issue A: Memory Parameter of Argon2 Is Too Low

Location

`src/config/argon.ts#L5`

`lib/src/utils.rs#L11`

Synopsis

The Namada Interface uses an `Argon2` memory parameter of 4 MiB, which is far below the recommendations prescribed in [BDK+21].

Impact

A low memory parameter reduces the attacker's costs of brute-forcing weak passwords.

Preconditions

This Issue is likely if the user-selected password has low entropy, and the attacker has access to the encrypted wallet data.

Feasibility

The attack is straightforward, if profitable. Given measurements of around 19 ms/hash (on a single core virtual private server) and VPS service offerings of around 1 cent per hour for a single-core VM, we estimate the costs of brute-forcing a wallet to be below 17,000 USD. There are likely better value offers, so the real cost is probably lower.

Technical Details

`Argon2` is a memory-hard function commonly used for deriving keys and hashes from passwords and is particularly resistant to brute-force attacks. However, the memory parameter must be high enough to sufficiently slow down an attacker.

Remediation

We recommend following the guidelines detailed in [BDK+21] and using 64 MiB for memory-constrained applications. Hashing with these parameters takes between 600 and 1500 ms. However, even if some computers are slower, this might still be acceptable.

Status

The Heliax team has increased the memory parameter to 64MiB.

Verification

Resolved.

Issue B: Several Cases of Problematic Unused Code

Location

`packages/utils/src/crypto/index.ts`

`packages/crypto/lib/src/utils.rs`

`packages/crypto/lib/src/crypto/scrypt.rs`

`packages/crypto/lib/src/crypto/aead.rs`

Synopsis

There are several pieces of code that have problems of varying severity but are not used anywhere.

Impact

All problems relate to either encryption or the derivation of keys used for encryption. As such, the most likely impact would be a failure of confidentiality or authenticity of encrypted data.

Preconditions

Since the problematic code is not currently being used, it only poses a vulnerability if it remains in the repository. In such a case, if new code is added in the future that uses this problematic code, then the leftover code could be exploitable.

Technical Details

- The encryption and decryption functions in `packages/utils/src/crypto` use `crypto-js` without a message authentication code (MAC). Since the Cipher Block Chaining (CBC) mode is malleable, this is not secure.
- The parameters used in `packages/crypto/lib/src/crypto/scrypt.rs` and `packages/crypto/lib/src/utils.rs` are too low.
- The encryption functions in `packages/crypto/lib/src/crypto/aead.rs` use the code in `packages/crypto/lib/src/utils.rs` and inherit the vulnerabilities.

Remediation

We recommend removing problematic and unnecessary lines of code.

Status

The Heliax team has removed all of the problematic code in question.

Verification

Resolved.

Issue C: Seed Phrase Stored in Clipboard

Location

`AccountCreating/Steps/SeedPhrase.components.ts#L89`

Synopsis

The wallet currently features a button that allows users to copy the seed phrase to the system clipboard. Since all processes running on the system can read the clipboard, this presents a security risk when malicious programs are present on the system.

Impact

The seed phrase allows the attacker control over the wallet, which can result in a complete loss of funds.

Preconditions

For this Issue to occur, the attacker would need to have a malicious program or browser extension running on the target's system.

Feasibility

The attack itself is trivial. For example, scripts that scan for seed phrases are available online, and restoring a wallet from a seed phrase is possible using the Namada Interface Extension.

Remediation

We recommend removing the function from the extension and warning users against screenshotting the seed phrase.

Status

The Heliax team has implemented the remediation as recommended.

Verification

Resolved.

Issue D: Usage of Strong Passwords Not Enforced

Location

`Steps/Password/Password.tsx`

Synopsis

In the current implementation, there are no restrictions on user-selected passwords. This could result in a user selecting an insufficiently secure password, which would enable an attacker with access to the target's machine to decrypt secrets or unlock the wallet.

Impact

A decrypted seed phrase gives full control over the wallet. Similarly, an attacker with access to the browser of the target can unlock the wallet and gain full access.

Preconditions

This issue is likely to occur if the user selects an insufficiently secure password, and the attacker is either in possession of the encrypted secrets or in control of the target's browser.

Feasibility

Attackers with encrypted secrets can easily guess the password using a brute-force algorithm.

Remediation

We recommend using a library such as `zxcvbn`, as noted in [Wheeler16], or libpwquality to estimate the strength of user-selected passwords, and requiring passwords to have the maximum strength of 4.

Status

The Heliax team has incorporated the `zxcvbn`-based password strength feedback into the password creation process.

Verification

Resolved.

Issue E: No Option To Change User Passwords

Synopsis

Once a user selects a password when creating an account, they are indefinitely unable to change their password. This presents an issue when a user's password gets compromised.

Impact

See Issue D.

Preconditions

As in Issue D, the attacker would need access to the target's browser or to have their encrypted seed phrase in their possession. Additionally, in this case, they would also need to know the target's compromised password.

Feasibility

If the preconditions are met, the attack is trivial.

Remediation

We recommend adding an option allowing users to change their password, and re-encrypt the secrets, using the new password.

Status

The Heliax team has implemented the remediation as recommended.

Verification

Resolved.

Issue F: No Option To Delete Wallets

Synopsis

Users are only able to delete wallets and associated secrets from their machine by deleting the browser extension. While this is technically possible, it is also difficult and unintuitive.

Impact

A user who wants to delete their secrets may find themselves unable to do so. Consequently, this might lead to their secrets being stolen from the hard drive, which could further result in a complete loss of funds.

Preconditions

An attacker would need to either infiltrate the target machine remotely with malicious software or come into physical possession of a discarded hard drive.

Feasibility

Once the attacker has the secret in their possession, they would have to break the encryption, which might be easy given Issue D and Issue E.

Remediation

We recommend adding an option to the interface that deletes any secrets associated with the wallet from the user's hard drive.

Status

The Heliax team has added a button to the settings page, allowing users to delete their accounts.

Verification

Resolved.

Issue G: Secrets Stored in Memory in Rust Code

Location

`packages/crypto`

Synopsis

Our team found instances of secret data not being cleared from memory appropriately. Clearing secrets from memory once they are no longer needed is a known mitigation to memory-dump-based attacks.

Impact

The attacker may learn secrets handled in the application, including the mnemonic or the keys used to encrypt it, which would enable full control of the wallet.

Preconditions

The attacker would need to have access to a memory snapshot of the WebAssembly (wasm) sandbox.

Feasibility

Although this type of exploit requires skill, no computational or financial resources are needed.

Mitigation

We recommend using the `zeroize` library, which is relatively easy to do in Rust.

Status

The Heliax team has implemented the remediation as recommended.

Verification

Resolved.

Issue H: Insufficient Sanitization of Parameters

Location

`src/helpers/index.ts#L98`

Synopsis

There is a sanitization in place for URL parameters to strip invalid characters. However, this list is not extensive and does not cover all possible unwanted characters.

Impact

The possible impact of this short sanitization is the bypass of said list to perform attacks through the URL parameters, which might lead to a vulnerability.

Preconditions

Any endpoint using the `getUrl` function, which joins sanitization functions, would be affected by the lack of sanitization.

Feasibility

Any attacker that launches a fuzzing scan against an unprotected URL would discover the lack of sanitization easily.

Technical Details

By using a fuzzer or dynamic scan on a URL that is poorly sanitized, an attacker would try different characters that could trigger different responses from the backend of the extension. Based on the input of the user, this could either result in a disruption of the actual extension workflow or even a faulty response that would give an attacker more information than intended.

Mitigation

Using Web Application Firewall solutions that sit in front of the extension and receive all kinds of inputs will help the application be secure from unwanted inputs.

Remediation

We recommend creating a list of expected or wanted inputs from users and using it as a whitelist approach to sanitize inputs against it.

Status

The Heliax team has started using `DOMPurify` to sanitize URL parameters.

Verification

Resolved.

Suggestions

Suggestion 1: Make All Tests Succeed (Known Issue)

Synopsis

Our team ran the tests and found that 5 out of 7 test suites and 1 out of 32 tests failed in the README.

Mitigation

We recommend making all tests succeed to confirm that every unit works as intended.

Status

The Heliax team has upgraded the existing tests to make them succeed.

Verification

Resolved.

Suggestion 2: Include Additional Instructions for Running and Testing the Project

Location

`namada-interface#-run-namada-interface`

Synopsis

The instructions included in the README are not sufficient, as auditors and maintainers would need to install dependencies related to Rust and the OpenSSL library in order to successfully test and run the project.

Mitigation

Robust and comprehensive documentation helps security teams assess the in-scope components and understand the expected behavior of the system being audited. As a result, we recommend supplementing the document with all the information required to run and test the project.

Status

The Heliax team has extended the build instructions.

Verification

Resolved.

Suggestion 3: Add a Linter Into the CI

Synopsis

We identified several cases of unused functions and variables as well as a general lack of code comments.

Mitigation

We recommend adding linters, for all components of the code, to the CI pipeline to help make these issues visible and keep the code clean.

Status

The Heliax team has implemented `ESLint` based linting in their CI pipeline.

Verification

Resolved.

Suggestion 4: Improve Code Comments

Synopsis

There are little to no comments explaining the rationale behind certain critical lines of code. 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 within the codebase to better describe the intended functionality of the code, facilitating reasoning about the security properties of the system.

Status

The Heliax team has added the improvement of code comments to their roadmap. However, it remains an open task at the time of verification.

Verification

Unresolved.

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.