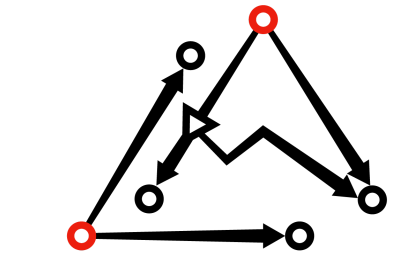

Least Authority

PRIVACY MATTERS

Saferoot Snap

Security Audit Report

Staging Labs

Final Audit Report: 28 August 2023

Table of Contents

Overview

Background

Staging Labs has requested that Least Authority perform a security audit of their MetaMask Snap.

Project Dates

- **August 15, 2023 - August 22, 2023:** Initial Code Review *(Completed)*
- **August 23, 2023:** Delivery of Initial Audit Report *(Completed)*
- **August 28, 2023:** Verification Review *(Completed)*
- **August 28, 2023:** Delivery of Final Audit Report *(Completed)*

Review Team

- Jehad Baeth, Security Researcher and Engineer
- Steven Jung, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the MetaMask Snap 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:

- Staging Labs Saferoot Snap:
  https://github.com/Staging-Labs/Saferoot-Snap

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

- `f5c48bc6ef9fc4c2ce6da1ca015ad9b0f4fadaae`

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

- Staging Labs Saferoot Snap:
  https://github.com/LeastAuthority/StagingLabsSaferoot-Snap

For the verification, we examined following Git revisions:

- `5013cbd39bdec7360e5a1d0abc3a9563ed52c6c8`
- `20a71e3e27d340377c7263cfc85593e1a9bd59c0`
- `852b3f16102eb594fa754b02d7feda68fa8e15b1`
- `5b4022e1514c14e3c024371cb7786873ba01193b`

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:

- Website:
  https://www.staginglabs.io

In addition, this audit report references the following documents:

- M. Dworkin, “Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC.” *NIST Special Publication 800-38D*, 2007, [Dworkin07]
- `localstorage`:
  https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage
- `snap_getEntropy`:
  https://docs.metamask.io/snaps/reference/rpc-api/#snap_getentropy
- Snaps execution environment:
  https://docs.metamask.io/snaps/concepts/execution-environment
- Snaps design guidelines:
  https://docs.metamask.io/snaps/concepts/design-guidelines

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the Snap implementation;
- Potential misuse and gaming of the Snap;
- Attacks that impacts funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Adversarial actions and other attacks on the network;
- Denial of Service (DoS) and other security exploits that would impact the intended use of the Snap or disrupt the execution of the Snap capabilities;
- Vulnerabilities in the Snap code;
- Protection against malicious attacks and other ways to exploit Snap code;
- 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

Our team performed a comprehensive security assessment of Saferoot, a web application designed to identify and intercept malicious transactions by redirecting assets to a secure destination. Saferoot's system consists of smart contracts and a decentralized application (dApp), which serves as a user interface between the Ethereum blockchain and the Saferoot backend, allowing its users to interact with the system.

In our review, we investigated Saferoot’s storage and encryption of sensitive metadata, input and response validation, in addition to the implementation of custom API for signing with Ethereum. We also reviewed Saferoot’s utilization of the MetaMask security framework and adherence to the security best practices.

System Design

Our team identified an Issue relating to sensitive information and user metadata being stored in `localStorage` in plaintext, which could be extracted easily in case the system is compromised. We recommend encrypting all sensitive data and privacy-relevant metadata using encryption (Issue A).

In addition, there are several instances of errors not handled properly, which could result in a denial of service. We recommend that error handling be improved. (Suggestion 1).

Code Quality

Our team identified several suggestions that would improve the quality of the code and contribute to the overall security of the implementation (Suggestion 2, Suggestion 4, Suggestion 6).

Tests

During our review, our team found no tests within the codebase. We recommend implementing a test suite, which helps identify implementation errors that could lead to security vulnerabilities (Suggestion 7).

Documentation and Code Comments

The project documentation provided for this review offers a generally sufficient overview of the system. However, we found that the documentation within the codebase is not comprehensive and can be further improved (Suggestion 5).

Scope

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

Dependencies

We examined all the dependencies implemented in the codebase and identified several instances of vulnerable dependencies. We recommend improving dependency management (Issue B).

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: Data Is Stored Unencrypted in localstorage	Resolved
Issue B: Usage of Vulnerable Dependencies	Resolved
Suggestion 1: Improve Error Handling and Reporting	Resolved
Suggestion 2: Update Deprecated Functions	Resolved
Suggestion 3: Adhere to MetaMask Best Practices	Resolved
Suggestion 4: Remove Unused Code	Resolved
Suggestion 5: Improve Code Comments	Partially Resolved

Suggestion 6: Review the Linter Reported Issues	Resolved
Suggestion 7: Implement a Test Suite	Partially Resolved

Issue A: Data Is Stored Unencrypted in localstorage

Location

`src/utils/localStorage.ts`

Synopsis

The SafeRoots Snap uses `localstorage` property to store application-specific data (namely, the `authenticated-address` and theme). The data is stored unencrypted in plaintext format.

Impact

Storing sensitive information in local storage is not advisable, as any user with local machine privileges or any browser extension with enough permission can potentially bypass or access the stored data.

Preconditions

This Issue is likely if an attacker gains access to the filesystem with enough privilege or if a malicious browser extension with enough permission is installed.

Remediation

We recommend always encrypting application-specific data and metadata stored in `localStorage` with a strong encryption scheme, as recommended by the National Institute of Standards and Technology (NIST) in [Dworkin07]. MetaMask's `snap_getentropy` function can be utilized as a source of entropy to create the encryption secret key, as it is both Snap and user account specific.

Alternatively, we recommend reevaluating the overall design of the system, such that it better utilizes the security features provided by the MetaMask Snaps API. This can be achieved by moving some of the sensitive dApp components that handle the storage of data and authentication into the Snaps execution environment. This approach would allow the dApp to utilize all of the security features provided by MetaMask's Snap API, in addition to allowing developers to impose restrictions by configuring permissions within a sandboxed execution environment.

Status

The Saferoot Snap development team has changed the design of the system by eliminating the need for `localStorage`. Instead, the team currently solely relies on the backend to fully manage session-based authentication.

Verification

Resolved.

Issue B: Usage of Vulnerable Dependencies

Location

`packages/site/package.json`

Synopsis

Analyzing `package.json` for dependency versions using `npm audit` shows that the dependencies used in the MetaMask Snap have 9 reported known vulnerabilities (2 Moderate, 6 High, 1 Critical).

Impact

Using unmaintained dependencies or packages with known vulnerabilities may lead to critical security vulnerabilities in the codebase.

Remediation

We recommend following a process that emphasizes secure dependency usage to avoid introducing vulnerabilities to the MetaMask Snap and to mitigate supply chain attacks, which includes:

- Manually reviewing and assessing currently-used dependencies;
- Upgrading dependencies with known vulnerabilities to patched versions with fixes;
- Pinning dependencies to secure versions when upgrading vulnerable dependencies to secure ones, including pinning build-level dependencies in the `package.json` file to a specific version;
- Only upgrading dependencies upon careful internal review for potential backward compatibility issues and vulnerabilities; and
- Including Automated Dependency auditing reports in the project's CI/CD workflow.

Status

According to the npm `audit`, some of the dependencies used are still considered vulnerable, despite the fact that the Saferoot Snap development team updated them to the latest versions.
Since the team has implemented a dependency management process and included it in their development workflow, we consider this Issue resolved.

Verification

Resolved.

Suggestions

Suggestion 1: Improve Error Handling and Reporting

Location

Examples (non-exhaustive):

`hooks/API/helpers.ts`

`src/components/AuthenticationAdapter.ts`

Synopsis

In various locations within the Snap code, errors are neither handled nor reported properly. For example, there is limited error handling in the `predefinedRequests` function. If a fetch request fails for any reason, the function may not handle the error correctly, thereby causing the application to crash.

Mitigation

We recommend including a `try-catch` block around the fetch requests to handle any errors that might occur. Additionally, we recommend providing useful user-oriented feedback that helps end users better understand and handle any possible user experience issues.

Status

The Saferoot Snap development team has implemented improvements to error handling in various locations across the codebase, covering more possible edge cases and providing concise and helpful user-oriented error messages when needed. The team has also added extra type checking and input/output validations, effectively reducing the possibility of having uncaught runtime errors.

Verification

Resolved.

Suggestion 2: Update Deprecated Functions

Location

`blockchain/helpers/useWagmiWrite.ts#L58`

Synopsis

The functions used in the location noted above have been deprecated by the development team. Functions that are no longer supported could behave in unexpected ways.

Mitigation

We recommend replacing deprecated functions with up-to-date and actively maintained functions.

Status

The Saferoot Snap team has removed the instances where deprecated functions are used.

Verification

Resolved.

Suggestion 3: Adhere to MetaMask Best Practices

Location

`packages/snap/snap.manifest.json`

Synopsis

We found that the Snap implementation does not adhere to MetaMask naming conventions, which could result in unintended behavior.

Mitigation

We recommend adhering to all MetaMask implementation guidelines.

Status

The Saferoot Snap development team has implemented necessary updates to the `manifest.snap.json` file to comply with MetaMask Snap Design Guidelines.

Verification

Resolved.

Suggestion 4: Remove Unused Code

Location

src/components/AuthenticationAdapter.ts#L3

src/pages/management.tsx#L1-L15

molecules/WalletCard/index.tsx#L9

newComponents/atoms/WalletStatus.tsx#L16-L26

Synopsis

We identified several instances of unused code and unfinished features. This reduces readability and can confuse reviewers and maintainers. For example, `isConnected` and `isFetching` in the `WalletCard` component are not used. As a result, the `WalletStatus` feature is not complete.

Mitigation

We recommend that all unused code be removed from the codebase in order to improve readability and to prevent confusion or errors. We also recommend completing the development of the components and their features.

Status

The Saferoot Snap development team has removed instances of unused or commented-out code.

Verification

Resolved.

Suggestion 5: Improve Code Comments

Location

src/components/AuthenticationAdapter.ts

blockchain/helpers/useWagmiWrite.ts

Synopsis

Currently, the codebase lacks explanation in areas that handle sensitive functionalities, such as the use of the `AuthenticationAdapter` function and the helpers. 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 explaining, for example, expected function behavior and usage, input arguments, variables, and code branches can greatly benefit the readability, maintainability, and auditability of the codebase. This allows both maintainers and reviewers of the codebase to comprehensively understand the intended functionality of the implementation and system design, which increases the likelihood for identifying potential errors that may lead to security vulnerabilities.

Mitigation

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

Status

The Saferoot Snap development team has improved the code comments in some areas and stated that they will continue improving it in the future, as they consider this mitigation to be part of an ongoing and incremental process.

Verification

Partially Resolved.

Suggestion 6: Review the Linter Reported Issues

Synopsis

The implementation makes use of linter with properly set rules. However, our team found that a large number of issues were reported by linter. While linters tend to have some false positives, our team was unable to verify all of the findings reported by linter due to insufficient time.

Mitigation

We recommend running linter and assessing the findings to discard the false positives and implement remediations for the confirmed issues.

Status

The Saferoot Snap development team has addressed some of the issues reported by linter and stated that they will actively review and address these issues in future development iterations.

Verification

Resolved.

Suggestion 7: Implement a Test Suite

Synopsis

Our team found no tests in the repository in scope. Sufficient test coverage should include tests for success and failure cases, which helps identify potential edge cases, and protect against errors and bugs that may lead to vulnerabilities or exploits. A test suite that includes a minimum of unit tests and integration tests adheres to development best practices. In addition, end-to-end testing is also recommended to assess if the implementation behaves as intended.

Mitigation

We recommend creating a test suite for the Snap implementation to facilitate identifying implementation errors and potential security vulnerabilities by developers and security researchers.

Status

The Saferoot Snap development team has added functionality for implementing tests and stated that improving the test coverage will be an ongoing process.

Verification

Partially Resolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

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

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.