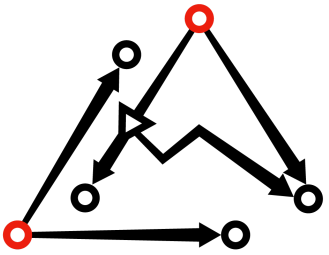

Least Authority

PRIVACY MATTERS

Mina Ledger Application

Security Audit Report

O(1) Labs

Updated Final Report Version: 5 February 2021

Table of Contents

Overview

Background

O(1) Labs has requested that Least Authority perform a security audit of the Mina Ledger Application and its integration for the Nano S/X Hardware Wallet. Mina is a cryptocurrency with a lightweight, constant-sized blockchain utilizing zk-SNARKs and aims to improve scaling while maintaining decentralization and security.

Project Dates

- **January 4 - 12**: Code review *(Completed)*
- **January 14**: Delivery of Initial Audit Report *(Completed)*
- **January 27 - 28:** Verification *(Completed)*
- **January 29:** Delivery of Final Audit Report *(Completed)*
- **February 5:** Delivery of Updated Final Audit Report *(Completed)*

Review Team

- Ramakrishnan Muthukrishnan, Security Researcher and Engineer
- Meejah, Security Researcher and Engineer
- Sajith Sasidharan, 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 Mina Ledger Application followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

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

- Mina Ledger Application: https://github.com/jspada/ledger-app-mina

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

`536062b323b326efd04fbe79b4c0f1d8b0c06944`

For the verification, we examined the Git revisions:

`70a46fd7dc1cc7631563a8b1ebe658798ca72763`

`0e59862c1639e93dfe20b2744d0549014b6bcd11`

`27eab1bcd29505bf7856946c61484ce66561dd26`

`e998834f4835acaa72ca31a6eb6243b4332b8574`

`4e11b3656f0b31661d0609993acf0cad52df04e1`

`982483d6857bc0bc822282f4985164f28d87f3c9`

`7712a61e8ec6a89d942b9158e6340dffb073c0b5`

`262669fe6982b83079de59ce2e36befa6a90c32b`

`14797f1ebd9453eb1cef96c64003e3eeb0ecbd4e`

`92b5c8c25fcaee71c9c8d4d8698f03d183954ca9`

`51d8d27c11b2bb1cd0a98c3c05b0472801c0e9ef`

`81d322580478a86de7b9877d607ecd40a7c8c0ac`

For the review, this repository was cloned for use during the audit and for reference in this report: https://github.com/LeastAuthority/ledger-app-mina

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:

- `README.md`: https://github.com/jspada/ledger-app-mina/blob/master/README.md
- API Documentation: https://github.com/jspada/ledger-app-mina/blob/master/doc/api.asc
- Poseidon paper: https://eprint.iacr.org/2019/458.pdf
- C-reference Implementation: https://github.com/MinaProtocol/c-reference-signer.git
- OCAML reference implementation: https://github.com/MinaProtocol/signer-reference

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Potential compromise of secrets;
- Common C programming pitfalls;
- Input validation;
- Other methods that attackers can utilize to render the program useless;
- Vulnerabilities within each component;
- Secure interaction between the application and libraries;
- Adversarial actions and malicious attacks on the app;
- Key management implementation: secure private key storage and proper management of encryption and signing keys;
- Exposure of any critical information during user interactions with the app;
- Attacks that impacts funds, such as draining or manipulating of funds;
- Mismanagement of funds via transactions;
- Protection against malicious attacks and other methods of exploitation;
- 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

System Design

Overall, it is clear that the Mina team has taken security considerations into account when designing the Mina Ledger Application system. This is especially apparent by the care that has been taken to reduce heap allocations and to clear private keys after use.

Code Quality

The code is very well organized and follows development best practices. The Mina Ledger Application code is simple without undue complexity, making it easy to audit, which is an important security consideration. We found that the functions are easy to read and names of variables and functions clearly convey their intended use. The indentation is largely consistent, with the exception of the areas noted in Suggestion 1.

We also found that functions that do not accept parameters are declared with empty parentheses, which has the unintended consequence that any number of parameters might be passed to these functions without errors or warnings. As a result, we recommend declaring such functions that are intended to take no parameters with `(void)` rather than with empty parentheses (Suggestion 2).

The use of the `const` type qualifier is particularly commendable, as it clearly states the assignment structure within a function. Furthermore, we looked for common pitfalls in memory allocations and string handling, and identified no concerns in these areas.

Test coverage is extensive with unit tests, integration tests, and a command line wallet script. However, we believe there is a case for additional coverage that can be used to test code off-device, check for a variety of edge cases, prove correctness, and run memory checkers (Suggestion 3).

Documentation

Although code comment coverage is minimal, the content provided in the comments is sufficient, providing the necessary information and useful pointers to relevant papers where needed.

The existing project documentation is accurate and helpful. It provides both a guide for building the software and an overview of the project and its intended functionality, in addition to facilitating an understanding of the application and its integration for the Nano S/X Hardware Wallet.

At present, the round constants used in the Add Round Constants (ARC) steps of the Poseidon hash are provided without a method to verify that the values are not chosen in a way that makes the hash function vulnerable to attacks from the party who originally generated the values. A well-documented script that reproducibly generates the values would alleviate any concerns around this topic (Issue A).

In addition, we recommend providing documentation on the Poseidon hash round count parameters to justify the deviation from Posseidon defaults (Suggestion 4). Furthermore, the scripts referenced in the Poseidon paper should be referenced in the repository in order to easily verify the security of the Maximum Distance Separable (MDS) matrix (Suggestion 5).

Scope

We found the Mina Ledger Application audit scope to be sufficient for examining the areas of concern defined above. One major dependency we have identified is the underlying BOLOS operating system, which is out of scope for this audit. While our team has no immediate concerns about the use of this dependency, BOLOS is a custom operating system that is not well-known and, to our knowledge, has not

been previously audited. As a result, we recommend that the dependency be reviewed and carefully maintained and updated as necessary.

Signatures

Following the completion of our initial review, the Mina team introduced a new commit to the repository. In addition to other minor changes to the code base, the commit includes a change that differentiates signatures on testnet from signatures on mainnet, in order to prevent replay attacks across both network types. The scalar value k used in the Point multiplication (k*g) is now derived both from the input message as well as the network type (mainnet or testnet). Similarly, the scalar nonce value used to multiply with the secret key is also derived by a hash function, which uses a different Poseidon initial state based on the network type. These functions ensure that signatures are different in testnet and mainnet for the same message. We commend this approach as the inclusion of this code helps to avoid cross-network replay attacks.

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: Clarify Generation of Poseidon ARC Round Constants	Resolved
Suggestion 1: Fix Code Indentation	Resolved
Suggestion 2: Declare Functions That Accept No Parameters With (void)	Resolved
Suggestion 3: Expand Unit Test Coverage To `src/crypto.c` and `src/poseidon.c`	Resolved
Suggestion 4: Document Rationale of Poseidon Round Counts Choice	Resolved
Suggestion 5: Make Security of Poseidon MDS Matrix Easily Verifiable	Partially Resolved

Issue A: Clarify Generation of Poseidon ARC Round Constants

Location

poseidon.c#L17-L1298

/pasta_params.sage

Synopsis

The way in which the Poseidon hash round constants were generated is currently unclear. As such, we must assume that the numbers may have been chosen in a way such that they have properties that open up a security vulnerability that is only exploitable by the party generating the numbers. A script to generate the parameters was found, however, it returns different numbers from those used in the code. Instead, nothing-up-my-sleeve numbers should be used.

Impact

It may be possible for the Mina team to find hash collisions or second preimages in a reasonable amount of time. Since the hashes are used in a signature scheme, breaking the hash function may allow forging signatures.

Preconditions

In order for the attack to be possible, the numbers must have been chosen in a particular way. There is no evidence that this is the case, but there is also no evidence that it is not.

Feasibility

The attacker would need significant knowledge of the Poseidon hash function and the mathematics behind it. The amount of computational complexity required for the attack is unclear to our team.

Technical Details

Poseidon is a SNARK-friendly hash function. It executes in multiple rounds and in each round it performs computations based on a set of constants. These should be close to random. However, it should also be verifiable that these numbers have not been chosen to undermine the security of the algorithm. This is usually done by generating the numbers using hashes or pseudorandom functions, based on short strings that describe the use, and do not provide room for the party generating the constants to introduce bias.

While we did find a script in another repository of Mina, we have not been able to verify that the numbers match.

Remediation

If the numbers are already generated in a verifiable manner, this should be clearly documented and made easy to verify that the numbers are those present in the code.

Should the numbers not be generated in a verifiable manner, a nothing-up-my-sleeve generation scheme should be used to make the generation verifiable. For example, the designers of Poseidon chose a scheme based on the Grain cipher. The description of their algorithm can be found in appendix F of the Poseidon paper.

Status

Following further discussion with the Mina team, we determined this to be a non-issue. Our team was able to verify that the numbers used are generated such that they can be verified to be random. Previously, we compared the actual ARC values with the generated MDS matrix values. With further clarification from the Mina team, we compared ARC constants to the generated ARC constants and were able to successfully conduct the verification.

However, we recommend that the Mina team simplify the verification process (See Suggestion 5). During our review, we found that the existing script prints poorly formatted Rust code, which is not accepted by rustfmt. We had to employ a semi-manual process that included writing custom code to make comparing the values to those in `src/poseidon.c` feasible. The process for verifying that the parameters were generated honestly should be streamlined by adding a script will simplify the verification process. This will make it easier for reviewers, developers, and contributors to verify the system is working as expected.

Verification

Resolved.

Suggestions

Suggestion 1: Fix Code Indentation

Location

/src/main.c#L47_L75

/src/get_address.h

Synopsis

The switch case in /src/main.c#L47_L75 is not formatted correctly and we have identified another instance of incorrect indentation in /src/get_address.h. Incorrectly formatted code is hard to read and can make flow control difficult to understand.

Mitigation

We suggest passing the code through a formatting utility like GNU indent, after agreeing on a formatting convention acceptable to all the developers contributing to the project. We recommend doing this as soon as possible, as changing the code formatting at a later point in the project would make it difficult to track code history. We also recommend that the formatting is fixed for all project files simultaneously, for internal consistency.

Status

The Mina team fixed the improper indentation in the relevant parts of code in `src/get_address.h` and `src/main.c`. As a result, the suggestion is resolved as recommended.

Verification

Resolved.

Suggestion 2: Declare Functions That Accept No Parameters With `(void)`

Location

E.g. /src/main.c#L270

Synopsis

Functions that do not accept parameters are declared with empty parentheses (e.g. in /src/main.c#L270). This has the unintended consequence that any number of parameters might be passed to these functions without errors or warnings.

Technical Details

According to GNU C:

> *"You can also declare a function that has a variable number of parameters (see Variable Length Parameter Lists), or no parameters using void. Leaving out parameter-list entirely also indicates no parameters, but it is better to specify it explicitly with `void`."*

C99 (ISO/IEC 9899:1999) also specifies the following (footnote 126):

> *"As indicated by the syntax, empty parentheses in a type name are interpreted as ``function with no parameter specification'', rather than redundant parentheses around the omitted identifier."*

In actual implementation, this often means that the function with empty parentheses (i.e. empty parameter list) often takes any number of arguments, which is the opposite of the intention of the programmers in this case.

Consider the following example:

```
#include <stdio.h>

int foo()

{

    return 42;

}

int main(void) {

    int x = foo(1, 2);

    printf("%d\n", x);

}
```

Here we have a function `foo()` that is not supposed to take arguments, but instead, we are passing two parameters when `foo()` is called from `main()`.

Compiling this with -Wall does not throw any warnings:

```
$ gcc -Wall -Wextra -std=c99 foo.c && ./a.out

42
```

In contrast, changing the definition of `foo()` to `foo(void)` triggers a compiler error.

```
$ gcc -Wall -Wextra -std=c99 foo.c && ./a.out

foo.c: In function ‘main’:

foo.c:9:13: error: too many arguments to function ‘foo’

    9 |     int x = foo(1, 2);

      |             ^~~

foo.c:3:5: note: declared here

    3 | int foo(void)

      |     ^~~
```

Mitigation

We suggest declaring such functions that are intended to take no parameters with `(void)` rather than with empty parentheses.

Status

The Mina Team has [implemented a fix] resolving this suggestion, such that functions that accept no parameters are declared with `(void)`.

Verification

Resolved.

Suggestion 3: Expand Unit Test Coverage To `src/crypto.c` and `src/poseidon.c`

Location

/src/crypto.c

/src/poseidon.c

Synopsis

When running to the code coverage program `gcov`, `src/crypto.c` and `src/poseidon.c` are not covered in the unit tests, although integration tests for these files are run by `/tests/unit_tests.py` on the `Ledger` device (see the corresponding GitHub Issue for details).

Mitigation

We recommend including more granular unit tests that exercise the functions in `src/crypto.c` and `src/poseidon.c` fully and directly, off-device. These can be used to test code off-device, check for a variety of edge cases, prove correctness, and run memory checkers.

Status

The Mina team has integrated Speculos into the build system, in order to run the unit tests on the build host. The following commits implement this resolution:

27eab1b

e998834

4e11b36

In addition, the Mina team has refactored existing tests and added new tests for `src/crypto.c`. Furthermore, validation of inputs for signature validation and for address validation are now performed by the `utils/mina_ledger_wallet.py` program that interacts with the hardware wallet. Lastly, the documentation on running the tests has been updated.

Verification

Resolved.

Suggestion 4: Document Rationale of Poseidon Round Counts Choice

Location

/poseidon.h#L23-L24

Synopsis

The Poseidon round parameters used by Mina are sixty-three full rounds and one additional ARC step. This diverges from what is described in the Poseidon paper, where typically only eight full rounds are performed, plus around sixty partial rounds (i.e., with fewer S-Box evaluations). Since the parameters are chosen to be stronger than those described in Poseidon, we do not consider this an issue. However, an

explanation for why these parameters are used and why the additional S-Box evaluations are performed would be interesting for technical audiences, including security auditors.

Mitigation

Describe in the documentation how the parameters were chosen and the rationale and reasoning for choosing them.

Status

The Mina team resolved this suggestion by updating the documentation in `src/poseidon.c`.

Verification

Resolved.

Suggestion 5: Make Security of Poseidon MDS Matrix Easily Verifiable

Location

/poseidon.h#L23-L24

Synopsis

The Poseidon paper lists a number of checks to be performed when generating the MDS matrix in order to ensure its security and provides a link to a Sage script that can be used for this. Some of these checks are not performed in the Mina MDS matrix generation script. We modified the script so that, instead of finding a secure matrix, it verifies the security of the matrix actually used inside the Mina Poseidon variant.

Mitigation

We recommend including a similar script in the repository, in order to make security more easily verifiable. The script modified by our team is available and can be provided upon request to be used as a reference.

Status

Upon request from the Mina team, we provided the modified script to be used as a reference and recommend that it be modified and refined prior to its incorporation into the repository.

Verification

Partially Resolved.

Recommendations

We commend the Mina team for promptly addressing and resolving the *Issue* and *Suggestions* stated above and for the strong considerations for security demonstrated in the system design and implementation of this project.

Unit test coverage that exercises the functions in `src/crypto.c` and `src/poseidon.c` to test code off-device, check for a variety of edge cases, prove correctness, and run memory checkers has been expanded.

Finally, we recommend that documentation be incorporated on the way Poseidon hash round constants are generated so that the randomness of the generation is verifiable.

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 C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. 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. Although we are a small team, we believe that 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 included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site 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. While 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, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, 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 though we have not yet 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 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 report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, 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 resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.