Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Light Protocol

Poseidon Hash

Veridise Inc.
July 13, 2023

► **Prepared For:**

Light Protocol
https://www.lightprotocol.com/

► **Prepared By:**

Kostas Ferles
Daniel Domínguez Álvarez
Alp Bassa

► **Contact Us:** contact@veridise.com

► **Version History:**

May 1, 2023	Initial Draft
Jul 13, 2023	V1

Contents

1 Executive Summary

From April 17, 2023 to April 28, 2023, Light Protocol engaged Veridise to review the security of their Poseidon Hash implementation. The review covered a library, implemented by Light Protocol, which exposes an API for calculating Poseidon Hashes over sequences of bytes. Veridise conducted the assessment over 4 person-weeks, with 2 engineers reviewing code over 2 weeks on commit `de1f8e8`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The Light Protocol developers provided the source code of the Poseidon Hash implementation for review. To facilitate the Veridise auditors' understanding of the code, the Light Protocol developers also provided documentation of the codebase in the form of README files and source code comments. As this is an implementation of a well-known hash function, these forms of documentation were adequate for understanding the Light Protocol's hash implementation.

The source code contained a test suite, which the Veridise auditors studied to estimate its coverage and understand the library's expected API usage.

Summary of issues detected. The audit uncovered 4 issues, 1 of which is assessed to be of high severity by the Veridise auditors. Specifically, the issue pointed out an undocumented design decision in Light Protocol's API that can lead to hash collisions in client applications. The Veridise auditors also identified two medium-severity issues that also relate to the API's usablity as well as one minor issue.

Recommendations. After auditing the library, the auditors had a few suggestions to improve the Light Protocol's Poseidon Hash implementation. Specifically, they suggested certain API changes that would prevent client applications from misusing the Light Protocol's API. They also suggested some additions to their API that are standard for Poseidon hash and will enable more client applications to utilize their library.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Poseidon Hash	`de1f8e8`	Rust	Solana

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
April 17 - April 28, 2023	Manual & Tools	2	4 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	1	1
Medium-Severity Issues	2	2
Low-Severity Issues	0	0
Warning-Severity Issues	0	0
Informational-Severity Issues	1	1
TOTAL	4	4

Table 2.4: Category Breakdown.

Name	Number
Usability Issue	3
Maintainability	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Light Protocol's Poseidon Hash library. In our audit, we sought to answer the following questions:

- Does the API conform to the standard?
- Does the library generate the Poseidon hash configuration parameters correctly?
- Does the Light Protocol's implementation deviate from existing reference implementations?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* We used `cargo-audit`, an open-source static analysis tool used to audit `Cargo.lock` files for crates with security vulnerabilities reported to the RustSec Advisory Database.
- *Fuzzing/Property-based Testing.* We also leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we fuzz tested Light Protocol's implementation against a reference (audited) implementation. Our fuzz tests did not uncover any cases where the two implementations differ. We ran a fuzzing campaign that lasted two and a half days and did nearly 40 million executions.

Scope. The scope of this audit is limited to the `light-poseidon/src` folder of the source code provided by the Light Protocol developers, which contains the implementation of the Poseidon Hash.

Methodology. Veridise auditors inspected the provided tests and read the Poseidon Hash documentation. They then began a manual audit of the code assisted by both static analyzers and automated testing. During the audit, the Veridise auditors regularly met with the Light Protocol developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowleged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-TODO-VUL-00	Function hash_bytes is prone to collisions	High	Fixed
V-TODO-VUL-00	Unconventional padding during hashing	Medium	Fixed
V-TODO-VUL-00	Domain Tag is not Configurable	Medium	Fixed
V-TODO-VUL-00	Typo in error message	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-LGHP-VUL-001: Function hash_bytes is prone to collisions

Severity	High	**Commit**	de1f8e8
Type	Usability Issue	**Status**	Fixed
File(s)	`light-poseidon/src/lib.rs`		
Location(s)	PoseidonBytesHasher::hash_bytes		

The function `PoseidonByteHasher::hash_bytes` has a flaw that can cause it to return the same hash for different sequences of bytes. As the attached code snippet shows, `hash_bytes` takes an array of byte sequences with arbitrary length and returns an array of `HASH_LEN` bytes as the hash. The first step of the function is to map each sequence of bytes from the inputs to an element of the finite field `F`, which is a type parameter of `PoseidonByteHasher`. However, the mapping simply interprets each byte sequence as a big integer and then takes the modulo over the field's prime. Since all byte sequences whose big integer representation are congruent modulo `F`'s prime will map to the same field element, multiple byte sequences can result in the same hash value.

```
1    fn hash_bytes(&mut self, inputs: &[&[u8]]) -> Result<[u8; HASH_LEN],
     PoseidonError> {
2        let inputs: Vec<F> = inputs
3            .iter()
4            .map(|bytes| F::from_be_bytes_mod_order(bytes))
5            .collect();
6        let hash = self.hash(&inputs)?;
7
8            ...
9    }
```

Impact The current design decision is undocumented. As a result, clients of the library may inadvertently introduce vulnerabilities in their codebase by not restricting the byte sequences they pass to `hash_bytes`.

Recommendation We recommend changing the API to accept field elements instead of byte sequences, which is the common practice for Poseidon hash implementation. Typically, in order to achieve a desired level of security, client applications require control over the mapping of input data to field elements (see here). Therefore, changing the API to accept field elements would enable even more applications to use the library.

Developer Response Developers acknowledged the issue and implemented the recommended changes in commit 5e2905b.

4.1.2 V-LGHP-VUL-002: Unconventional padding during hashing

Severity	Medium	**Commit**	de1f8e8
Type	Usability Issue	**Status**	Fixed
File(s)	`light-poseidon/src/lib.rs`		
Location(s)	PoseidonHasher::hash		

If the number of field elements passed to the `PoseidonHasher` function is smaller than the number of inputs specified in the hasher's configuration, the implementation pads the inputs with zeros to match the expected length.

Impact Although this behavior is documented, it may still introduce issues in client applications as it leaves room for users to misuse the API.

Recommendation We recommend changing the API to accept exactly as many elements as the hasher's configurations. You can leverage Rust's type system to achieve this.

Developer Response Developers acknowledged the issue and implemented the recommended changes in commit 4294ecc.

4.1.3 V-LGHP-VUL-003: Domain Tag is not Configurable

Severity	Medium	**Commit**	de1f8e8
Type	Usability Issue	**Status**	Fixed
File(s)	`light-poseidon/src/lib.rs`		
Location(s)	PoseidonHasher::hash		

The authors of the Poseidon hash recommend using domain separation when an application utilizes multiple instances of the Poseidon hash (Section 4.2 in the paper). However, the current implementation hardcodes the domain tag to zero.

Impact Applications that require multiple instantiations of Poseidon hash may not be able to utilize this library.

Recommendation We recommend adding a configuration parameter to all functions that create a `Poseidon` instance and adjust the implementation of `PoseidonHasher::hash` accordingly.

Developer Response Developers acknowledged the issue and implemented the recommended changes in commit 8680c4e.

4.1.4 V-LGHP-VUL-004: Typo in error message

Severity	Info	**Commit**	de1f8e8
Type	Maintainability	**Status**	Fixed
File(s)	`light-poseidon/src/lib.rs`		
Location(s)	enum PoseidonError		

There is a small typo in the error message for `U64Tou8` in `PoseidonError` (”a the number”).

Consider fixing it like follows:

```
    - #[error("Failed to convert a the number of inputs to a u8")]
    + #[error("Failed to convert the number of inputs to a u8")]
    U64Tou8,
```

Developer Response Developers acknowledged the issue and implemented the recommended changes in commit e4c46ad.