Lighthouse

Security Assessment

May 3rd, 2024 — Prepared by OtterSec

Thibault Marboud thibault@osec.io

Robert Chen notdeghost@osec.io

Table of Contents

01 — Executive Summary

Overview

Lighthouse engaged OtterSec to assess the `lighthouse` program. This assessment was conducted between April 18th and April 25th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 2 findings throughout this audit engagement.

We made recommendations to enhance the overall code efficiency (OS-LHS-SUG-00) and also to modify the code base to enhance its security and ensure adherence to coding best practices (OS-LHS-SUG-01).

Scope

The source code was delivered to us in a Git repository at https://github.com/Jac0xb/lighthouse. This audit was performed against commit 293470d.

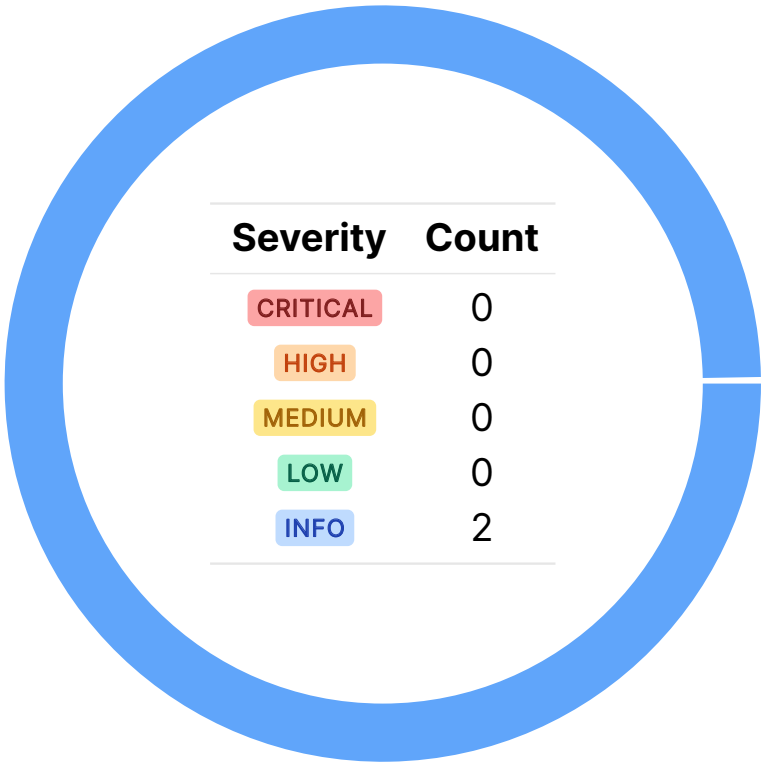
A brief description of the programs is as follows:

Name	Description
lighthouse	A Solana program that provides assertion instructions that may be added to transactions.

02 — Findings

Overall, we reported 2 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	0
LOW	0
INFO	2

03 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-LHS-SUG-00	Recommendations to enhance the overall code efficiency.
OS-LHS-SUG-01	Recommendations for modifying the code base to enhance its security and ensure adherence to coding best practices.

Code Optimization OS-LHS-SUG-00

Description

1. `sol_memcmp` is utilized within `keys_equal` to compare two `Pubkeys` by invoking the `sol_memcmp` system call instead of using the native equality operator (`==`). Although unconventional, comparing byte slices via `sol_memcmp` is a more economical approach compared to utilizing the native equality operator. Similarly, we suggest substituting `sha2_const_stable` with the `sol_sha256` system call to benefit from platform-specific optimizations inherent in the low-level call for calculating the `SHA-256` hash.
2. As Lighthouse is designed to be invoked alongside other instructions, the transaction's size limit poses a challenge. Implementing Run-Length Encoding (RLE) to encode integers, especially lengths will be a beneficial optimization strategy for reducing the size of transaction instructions by detecting and compressing repetitive data patterns. Run-Length Encoding aids in minimizing the overall transaction data size.

Remediation

Apply the above optimizations for increased efficiency in the code base.

Patch

The second issue related to Run-Length Encoding was fixed in PR#42.

Code Refactoring OS-LHS-SUG-01

Description

1. In the `MemoryWrite` instruction, the `WriteType` enumeration encompasses various variants to accommodate different types of data writes. Each variant in the enumeration contributes to the instruction's size, as the variant discriminant and associated data must be serialized and transmitted as components of the instruction.

   >_ *instructions/memory_write.rs* rust

   ```rust
   pub fn write_type(&mut self, write_type: WriteType) -> &mut Self {
       self.instruction.write_type = Some(write_type);
       self
   }
   ```

2. Maintaining separate single and multi versions of instructions duplicates code and adds unnecessary complexity to the code base. With the multi-version instruction, developers may still perform single assertions by passing a vector with only one element. This approach maintains flexibility while eliminating the need for separate single-version instructions.
3. Instead of creating separate assertion instructions for each account type, the existing `AssertAccountData` instruction may be utilized to dynamically specify the offset within the account data where the assertion should be performed. This allows the same instruction to assert different fields or sections of account data across various account types.

Remediation

1. Reduce the `WriteType` enumeration to include only the `DataValue::Bytes` variant to represent any arbitrary byte data. With the utilization of Run-Length Encoding, there should be a very small instruction size tradeoff.
2. Remove redundant single-version instructions and utilize the multi-instruction to improve the efficiency of the code base by reducing its size and complexity.
3. Reuse `AssertAccountData` instead of implementing the `Assert` trait for every account type that needs to be supported. The user or the `SDK` will be responsible for determining the appropriate offset for the field.

Patch

The second issue related to the utilization of separate single and multi versions of instructions was fixed in PR#42.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.