Photon-messaging audit

Prepared on: 2024-01-31

Contract: AUD511

Prepared by:
charles Holtzkampf

Prepared for:
Entangle

Table of Contents

3 Executive Summary

4 Severity Description

5 Methodology

5 LOW

Executive Summary

This document outlines any issues found during the audit of the contracts:

- photon-cross-chain-messaging
- 683840899512af9f2da3543e5aa626788e6576e4

- The contract has 3 lows.
- 0 Gas security issues were found.
- 0 Medium security issues were found.
- 0 High security issues were found.
- The risk associated with this contract is low

LOW	GAS	MEDIUM	HIGH
3	0	0	0

Severity Description

LOW

lows are instances in the code that are worthy of attention, but in no way represent a security flaw in the code. These issues might cause problems with the user experience, confusion with new developers working on the project, or other inconveniences.
Things that would fall under lows would include:
- Instances where best practices are not followed
- Spelling and grammar mistakes
- Inconsistencies in the code styling and structure

GAS

Gas optimizations are crucial for the efficient execution of Ethereum smart contracts. Gas in Ethereum is a measure of computational effort. Each operation, including computations, transactions, or contract interactions, requires a certain amount of gas. Optimizing gas usage can significantly reduce the cost of executing smart contracts, making them more appealing to users.
Things that would fall under gas would include:
- Reducing storage operations.
- Simplifying computations.
- Batching operations.

MEDIUM

Issues of medium security can cause the code to crash unexpectedly, or lead to deadlock situations.
Things that would fall under medium would include:
- Logic flaws that cause crashes
- Timeout exceptions
- Reentrancy attack
- Sawndwich attacks

HIGH

High issues cause a loss of funds or severely impact contract usage.
Things that would fall under high would include:
- Missing checks for authorization
- Logic flaws that cause loss of funds
- Logic flaws that impact economics of system
- All known exploits (for example, on_notification fake transfer exploit)

Methodology

Throughout the review process, we check that the token contract:

- Documentation and code comments match logic and behaviour
- Is not affected by any known vulnerabilities

Our team follows best practices and industry-standard techniques to verify the proper implementation of the smart contract. Our smart contract developers reviewed the contract line by line, documenting any issues as they were discovered. Ontop of the line by line review, we also perform code fuzzzing.

Our strategies consist largely of manual collaboration between multiple team members at each stage of the review, including:

I. Due diligence in assessing the overall code quality of the codebase.
II. Testing contract logic against common and uncommon attack vectors.
III. Thorough, manual review of the codebase, line-by-line.

Our testing includes:

- Overflow Audit
- GAS optimizations
- Authority Vulnerability
- Re-entry
- Timeout
- RAM Attacks
- Fake contract
- Fake deposit
- Denial of Service
- Design Logic Audit
- RNG attacks
- Stack Injection attacks

Our Code Fuzzing methodology:

The contract is instantiated and it's public functions are called in random order, with random input, so as to explore the state space and find corner case inputs that might lead to undesired outcomes. Apart from detecting logic bugs, this approach allows us also to detect memory bugs, hangs, undefined behavior and crash bugs in a semi-automated manner. Since ETH contracts are usually designed to run and complete within a short amount of time, fuzzing them is very fast as well, and therefore an effective instrument for teasing out bugs within the duration of an audit.

LOW

1 Ignored state of deducing fee might lead to affect the protocol's accounting system

In BetManager::releaseBetsAndReward(), the protocol relies on the successful return of true from the deduceFee() function to register the agent. However, insufficient consideration has been given to handling the scenario where deduceFee() returns false. This oversight could potentially disrupt the protocol's accounting system, as it fails to account for situations where fee deduction encounters errors or fails. Without proper handling, such scenarios could lead to discrepancies in accounting records or unexpected protocol behavior, jeopardizing the integrity and reliability of the system.

2 Native assets could be lost in contract

The AggregationSpotter contract features two payable functions, yet lacks explicit mechanisms for transferring native assets. Consequently, if users inadvertently or maliciously send native assets along with calls to these functions, the contract may inadvertently lock these assets within its balance, making them irretrievable. This oversight poses a significant risk of asset loss and underscores the importance of implementing robust asset transfer mechanisms within payable functions.

3 Missing input check validation

The initial audit report, which advised implementing input validation checks, has not been fully followed. Specifically, while some input validation may exist within the codebase, it appears incomplete or insufficiently applied across all recommended inputs. This oversight leaves certain pathways vulnerable to unexpected or malicious inputs, potentially leading to exploitation or erroneous behavior within the system. It's crucial to thoroughly review and implement comprehensive input validation strategies to ensure the robustness and security of the protocol.