Bob Collective

Security Assessment

March 30th, 2024 — Prepared by OtterSec

Nicholas R. Putra nicholas@osec.io

Robert Chen notdeghost@osec.io

Matteo Oliva matt@osec.io

Table of Contents

01 — Executive Summary

Overview

Bob Collective engaged OtterSec to assess the `fusion-lock` program. This assessment was conducted between March 18th and March 22nd, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 2 findings throughout this audit engagement.

We recommended addressing the inadequate flexibility in bridge address configuration (OS-BOB-SUG-00) and advised clarifying in the comment that the gas limit parameter applies to each withdrawal individually, not to all withdrawals combined. This clarification would help avoid user confusion. (OS-BOB-SUG-01).

Scope

The source code was delivered to us in a Git repository at https://github.com/bob-collective/fusion-lock. This audit was performed against commit e4f25ee.

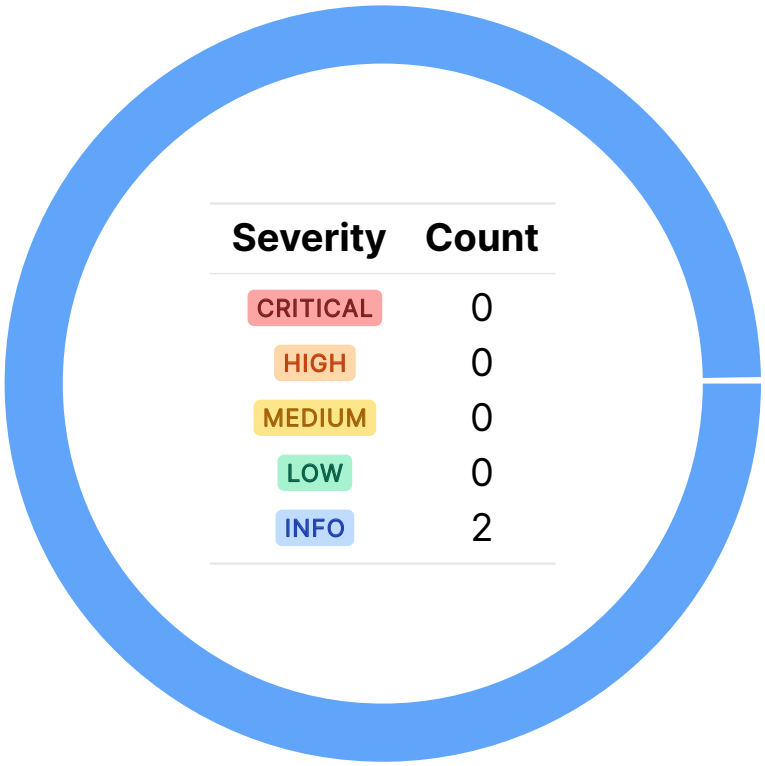
A brief description of the programs is as follows:

Name	Description
fusion-lock	Facilitates the management of token deposits and withdrawals across various blockchain layers, ensuring users maintain access to their funds while fostering interoperability between them.

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-BOB-SUG-00	The flexibility in bridge address configuration within `FusionLock` is insufficient.
OS-BOB-SUG-01	Specify in `withdrawSingleDepositToL2` that the `minGasLimit` parameter applies to each withdrawal individually.

Inconsistency In Address Flexibility OS-BOB-SUG-00

Description

In `FusionLock`, `l2TokenAddress` denotes the address of the corresponding token on layer two. This address may be modified dynamically at any point, facilitating updates to accommodate alterations in the token contract on layer two. This flexibility empowers the contract owner to adapt token mappings as necessary, guaranteeing compatibility with any upgrades or migrations of token contracts on the layer two network.

>_ FusionLock.sol solidity

```
function changeMultipleL2TokenAddresses(TokenAddressPair[] memory tokenPairs) external onlyOwner
    ↪ {
    for (uint256 i = 0; i < tokenPairs.length; i++) {
        TokenAddressPair memory pair = tokenPairs[i];
        // Ensure the token is allowed for deposit before changing its L2 address
        require(allowedTokens[pair.l1TokenAddress].isAllowed, "Need to allow token before
            ↪ changingL2 address");
        // Update the L2 address of the token
        allowedTokens[pair.l1TokenAddress].l2TokenAddress = pair.l2TokenAddress;
        emit TokenL2DepositAddressChange(pair.l1TokenAddress, pair.l2TokenAddress);
    }
}
```

Conversely, `l1BridgeAddressOverride` serves as an optional override address for the bridge contract used when bridging tokens to layer two. However, once set, this address cannot be modified during the withdrawal period, remaining static until the withdrawal period concludes. This lack of flexibility restricts the contract owner's ability to adapt to changes in bridge contracts or transition to alternative bridge implementations if necessary.

Remediation

Allow the `l1BridgeAddressOverride` field to be changed during the withdrawal period, mirroring the behavior of `l2TokenAddress`.

Patch

Fixed in da35903.

Gas Limit Clarification OS-BOB-SUG-01

Description

In the NatSpec for `withdrawSingleDepositToL2`, the comment regarding `minGasLimit` should clarify that this parameter sets the minimum gas limit for each withdrawal transaction. This ensures that every withdrawal has sufficient gas to execute independently. Users must understand that the gas limit is specific to each withdrawal rather than being shared among multiple withdrawals.

>_ *FusionLock.sol* solidity

```
/**
* @dev Internal function to withdraw tokens to Layer 2.
* @param token Address of the token to withdraw.
* @param minGasLimit Minimum gas limit for the withdrawal transaction.
* @param receiver The receiver of the funds on L2.
*/
function withdrawSingleDepositToL2(address token, uint32 minGasLimit, address receiver) internal
    ↪ {
    [...]
}
```

Remediation

Modify the comment for `minGasLimit` as described above.

Patch

Fixed in 2db42bb.

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.