Eclipse-syzygy

Security Assessment

October 31st, 2024 — Prepared by OtterSec

Nicholas R. Putra nicholas@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Eclipse engaged OtterSec to assess the `syzygy` program. This assessment was conducted between October 17th and October 21st, 2024. For more information on our auditing methodology, refer to chapter 05.

Key Findings

We produced 2 findings throughout this audit engagement.

We made recommendations around refactoring the code to mitigate possible security issues (OS-ECS-SUG-00) and suggested fixing certain inconsistencies in the codebase to ensure adherence to coding best practices (OS-ECS-SUG-01).

Scope

The source code was delivered to us in a Git repository at https://github.com/Eclipse-Laboratories-Inc. This audit was performed against commit b631617.

A brief description of the program is as follows:

Name	Description
syzygy	The Eclipse canonical bridge relayer.

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-ECS-SUG-00	Recommendation for modifying the codebase for improved functionality, efficiency, and maintainability.
OS-ECS-SUG-01	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.

Code Refactoring OS-ECS-SUG-00

Description

1. In `Treasury`, the `feeWei` is deducted from the `amountWei`, implying the fee is a portion of the withdrawal amount, not an additional amount on top of it. Thus, by adding `amountWei` and `feeWei` together, the contract is effectively checking if there is enough balance for the withdrawal amount plus an additional fee when only the total withdrawal amount is required. Therefore, the `totalAmount` check in `withdrawEth` is overly strict.

   >_ *Inc–syzygy/smart-contracts/src/v1/Treasury.sol* solidity

```solidity
function withdrawEth(
    address to,
    uint256 amountWei,
    address feeReceiver,
    uint256 feeWei
)
   [...]
{
    uint256 totalAmount = amountWei + feeWei;
    if (address(this).balance < totalAmount) {
        revert InsufficientFunds();
    }
   [...]
}
```

2. Update `deleteWithdrawMessage` to include the clearing of `withdrawMsgIdProcessed` to ensure consistency and accuracy in the contract's state.

   >_ *smart-contracts/src/v1/CanonicalBridge.sol* solidity

```solidity
function deleteWithdrawMessage(bytes32 messageHash) public
    ↪  onlyRole(WITHDRAW_CANCELLER_ROLE) {
    WithdrawStatus status = withdrawMessageStatus(messageHash);
    if (status != WithdrawStatus.PENDING && status != WithdrawStatus.PROCESSING) {
        revert CannotCancel();
    }
    startTime[messageHash] = 0;
    emit WithdrawMessageHashDeleted(msg.sender, messageHash);
}
```

3. Maintain consistency in the logic by either ensuring that `feeReceiver` is never zero or by handling the zero case appropriately. If the business logic requires a valid `feeReceiver`, add a check in `CanonicalBridge::validWithdrawMessage`. If the business logic allows zero for `feeReceiver`, adjust `Treasury::withdrawEth` to handle zero addresses gracefully without attempting to transfer the fee. This prevents potential errors or inconsistencies in how withdrawal messages and fees are processed.

Remediation

1. Ensure that the contract has enough ETH to cover the full withdrawal amount (`amountWei`) since the fee is already accounted for within that amount.
2. Implement the above refactor.
3. Update `withdrawMsgIdProcessed` in `deleteWithdrawMessage`.

Patch

1. Issue #1 resolved in f631034.
2. Issue #2 resolved in faac5da.
3. Issue #3 resolved in 8aa0e94.

Code Maturity OS-ECS-SUG-01

Description

1. In `CanonicalBridge::claimWithdrawal`, update `startTime` before interacting with the external contract (`withdrawEth` call) to prevent possible reentrancy exploits by following the check-effects-interaction pattern.

   >_ *lnc–syzygy/smart-contracts/src/v1/Treasury.sol* solidity

   ```solidity
   function claimWithdrawal(
       WithdrawMessage calldata message
   )
       [...]
   {
       [...].
       bool success = ITreasury(TREASURY).withdrawEth(
           message.destination,
           message.amountWei,
           message.feeReceiver,
           message.feeWei
       );
       if (!success) revert WithdrawFailed();
       startTime[messageHash] = NEVER;
       emit WithdrawClaimed(message.destination, message.from, messageHash, message);
   }
   ```

2. In the current implementation, multiple variables, such as `Treasury::StorageV1` and role constants, are not utilized. Ensure to remove any unutilized variables for improved readability.
3. To maintain the integrity of the withdrawal process in `CanonicalBridge`, add a check for `withdrawMsgIdProcessed` in the `_authorizeWithdraw` to prevent the reuse of withdrawal IDs, maintaining the uniqueness of each transaction.

   >_ *smart-contracts/src/v1/CanonicalBridge.sol* solidity

   ```solidity
   function _authorizeWithdraw(
       WithdrawMessage memory message
   ) private validWithdrawMessage(message) {
       [...]
       if (withdrawMessageStatus(messageHash) != WithdrawStatus.UNKNOWN) {
           revert CanonicalBridgeTransactionRejected(message.withdrawId, "Message already
               ↪ exists");
       }
       [...]
   }
   ```

Remediation

Implement the above-mentioned suggestions.

Patch

1. Issue #1 resolved in 26c409b.
2. Issue #2 resolved in 7ff4a81.
3. Issue #3 resolved in f1609c0.

04 — 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.

05 — 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.