Symbiotic

Security Assessment

September 9th, 2024 — Prepared by OtterSec

Robert Chen r@osec.io

Naoya Okanami minaminao@osec.io

Table of Contents

01 — Executive Summary

Overview

Symbiotic Finance engaged OtterSec to assess the `core` and `rewards` programs. This assessment was conducted between August 26th and September 3rd, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 3 findings throughout this audit engagement.

In particular, we identified an issue concerning the dependency of the slashed amount on the execution order of slashing operations, which may result in underslashing if previous slashes are not accounted for (OS-SYM-SUG-00).

We also made recommendations to ensure adherence to coding best practices (OS-SYM-SUG-02) and advised against enforcing sequential slashing, as legitimate slashes in close proximity have the potential to abort if processed out of order (OS-SYM-SUG-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.dev/symbioticfi/core. This audit was performed against commits 788f2d7 and d44b600.

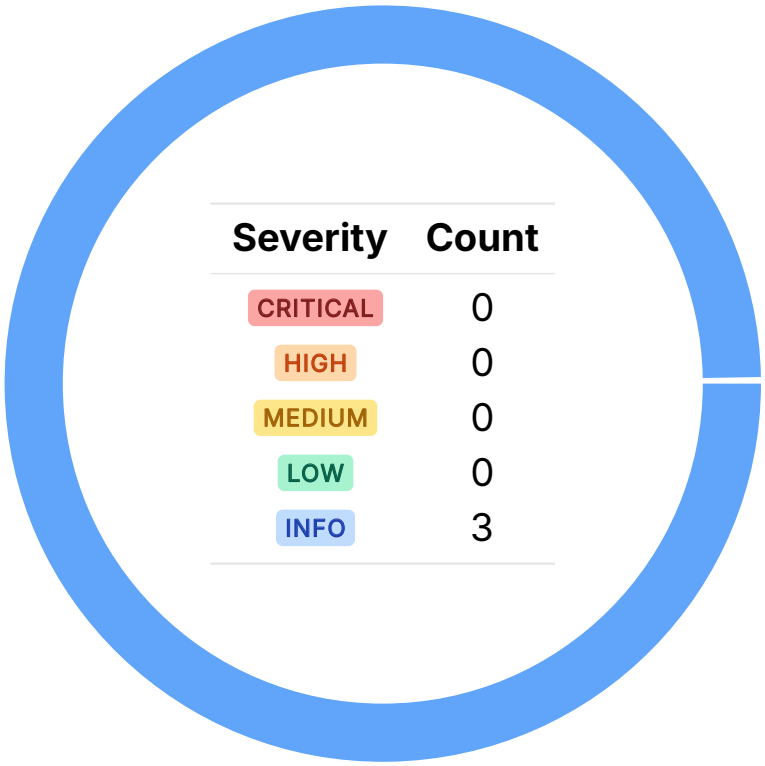
A brief description of the programs is as follows:

Name	Description
core	The core implementation of Symbiotic, a shared security protocol that allows decentralized networks to manage and customize their multi-asset restaking systems.
rewards	This module contains the logic for the Symbiotic Staker Rewards interface.

03 — Findings

Overall, we reported 3 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	3

04 — 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-SYM-SUG-00	The slashed amount is dependent on the execution order of slashing operations, which may result in underslashing if previous slashes are not accounted for.
OS-SYM-SUG-01	Enforcing sequential slashing in `_checkLatestSlashedCaptureTimestamp` may be too strict, as legitimate slashes in close proximity have the potential to abort if processed out of order.
OS-SYM-SUG-02	Suggestions regarding inconsistencies in the slashing logic.

Underslashing Due to Dependency on the Execution Order OS-SYM-SUG-00

Description

In `Slasher`, the slashed amount is computed based on the order of execution of slashes, which may result in underslashing. This is due to the fact that the slashing logic depends on the remaining `slashableStake` of an operator, which may change as multiple slashes are applied sequentially. `slashableStake` is calculated by subtracting the cumulative slashed amount up to the `captureTimestamp` from the total stake.

>_ *src/contracts/slasher/Slasher.sol* RUST

```
function slash(
    bytes32 subnetwork,
    address operator,
    uint256 amount,
    uint48 captureTimestamp,
    bytes calldata hints
) external onlyNetworkMiddleware(subnetwork) returns (uint256 slashedAmount) {
   [...]
    slashedAmount =
            Math.min(amount, slashableStake(subnetwork, operator,
                ↪  captureTimestamp,slashHints.slashableStakeHints));
    if (slashedAmount == 0) {
        revert InsufficientSlash();
    }
    [...]
}
```

If multiple slashing operations are executed against the same operator for the same period, the order in which these operations are processed affects the amount that may be slashed in each operation. If an earlier slashing operation reduces the operator's stake, subsequent slashing operations may not be able to slash as much as they would have if the earlier operations had not taken place, as `slashedAmount` is assigned the minimum of `amount` (the requested amount to slash specified by the caller) and `slashableStake`.

Remediation

Ensure that slashing operations are executed in a controlled manner to prevent issues arising from the order of execution.

Patch

In practice, it is assumed that slashing execution will be independent of batches, as explained in the Conveyor approach, and the finalization delay for slashings will be negligible for most cases.

Overly Strict Slashing Enforcement OS-SYM-SUG-01

Description

Enforcing strict sequential slashing via `BaseSlasher::_checkLatestSlashedCaptureTimestamp` may be too restrictive, as it requires the `captureTimestamp` for a slashing event to be strictly greater than the `latestSlashedCaptureTimestamp`. This prevents the processing of slashing events if their `captureTimestamp` is older than the most recent slashing event.

>_ src/contracts/slasher/BaseSlasher.sol RUST

```
function _checkLatestSlashedCaptureTimestamp(bytes32 subnetwork, uint48 captureTimestamp)
    ↪  internal view {
    if (captureTimestamp < latestSlashedCaptureTimestamp[subnetwork]) {
        revert OutdatedCaptureTimestamp();
    }
}
```

Consequently, if two slashes occur within the same time window, with very close `captureTimestamp` values, and the second slashing event (with a slightly higher `captureTimestamp`) is processed first, the first slashing event (with a slightly lower `captureTimestamp`) will be rejected, even though it is still valid and should be processed. Thus, strictly enforcing sequential slashing, as it is implemented now, may result in valid slashing events being rejected because they are processed out of order.

Remediation

Implement a system that allows minor discrepancies in `captureTimestamp` values, such as permitting slashes that are within the same epoch.

Slashing Inconsistencies OS-SYM-SUG-02

Description

1. In the current implementation of the slashing logic, it is possible to get penalized on the stake that was added in the current epoch for a slash event recorded in the previous epoch. Thus, even though this stake did not exist during the epoch of the slashing event, it is unfairly penalized for past infractions.
2. `Slasher` enforces that the `captureTimestamp` is at most one epoch duration behind the current timestamp to ensure that slashing events are relatively recent, but it operates directly on the raw `captureTimestamp`. `Vault`, however, converts the raw timestamps into epochs before performing the comparison. This approach aligns the timestamps with the system's epoch structure, rendering the check more accurate.

>_ src/contracts/slasher/Slasher.sol RUST

```
function slash(
    bytes32 subnetwork,
    address operator,
    uint256 amount,
    uint48 captureTimestamp,
    bytes calldata hints
) external onlyNetworkMiddleware(subnetwork) returns (uint256 slashedAmount) {
    [...]
    if (
        captureTimestamp < Time.timestamp() - IVault(vault_).epochDuration() ||
            ↪  captureTimestamp >=Time.timestamp()
    ) {
        revert InvalidCaptureTimestamp();
    }
    [...]
}
```

3. Currently, in `ERC4626Math`, an unnecessary `+1` is added when converting between shares and assets in `convertToShares` and `convertToAssets`, which results in a small rounding error that unfairly disadvantages the user.

Remediation

1. Ensure that slashing penalties are only applied to stakes present during the epoch when the slashing occurred.
2. Convert the timestamps into epochs first in `Slasher` for increased accuracy.
3. Remove the unnecessary addition operation.

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 others may have 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.