PUBLIC

Code Assessment

# of the Default Rewards Smart Contracts

August 20, 2024

Produced for

SYMBIOTIC

by

CHAINSECURITY

Contents

1 Executive Summary 3

2 Assessment Overview 5

3 Limitations and use of report 9

4 Terminology 10

5 Findings 11

6 Resolved Findings 12

7 Informational 16

8 Notes 17

1 Executive Summary

Dear all,

Thank you for trusting us to help Symbiotic with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Default Rewards according to Scope to support you in forming an opinion on their security risks.

Symbiotic provides default contracts for standardizing the distribution of rewards to operators and stakers.

The most critical subjects covered in our audit are asset solvency, function correctness and access control. The general subjects covered are specification and trustworthiness.

The most notable issue uncovered is the possibility of Stealing Operator Rewards. The finding has been addressed through code correction.

In summary, we find that the codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	1
• Code Corrected	1
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	3
• Code Corrected	2
• Risk Accepted	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Default Rewards repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	15 Jul 2024	35072e1ba48451e99a2e990c3acddc4b50463857	Initial Version
2	01 Aug 2024	d44b600ec423f8ba86347325fa32fd3d56738097	After Intermediate Report
3	15 Aug 2024	03e5d612926bc13297e350b6d455daabcdeb7fb9	Final Version

For the solidity smart contracts, the compiler version `0.8.25` was chosen.

The contracts listed below are in scope:

```
contracts:
    defaultOperatorRewards:
        DefaultOperatorRewards.sol
        DefaultOperatorRewardsFactory.sol
    defaultStakerRewards:
        DefaultStakerRewards.sol
        DefaultStakerRewardsFactory.sol
interfaces:
    defaultOperatorRewards:
        IDefaultOperatorRewards.sol
        IDefaultOperatorRewardsFactory.sol
    defaultStakerRewards:
        IDefaultStakerRewards.sol
        IDefaultStakerRewardsFactory.sol
    stakerRewards:
        IStakerRewards.sol
```

2.1.1 *Excluded from scope*

Generally, all contracts not listed above are out of scope. Further,

- all interacted-with external systems (e.g. security of tokens),
- the correctness of the core system,
- rebasing tokens and very obscure tokens,
- and off-chain computations

are out of scope.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

At the end of this report section, we have added subsections for each of the changes according to the versions.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Symbiotic implements contracts for distributing rewards to stakers and operators. Note that the usage of these contracts is optional and may differ if other contracts are deployed. For details regarding the core, please refer to our core audit.

2.2.1 Default Operator Reward

The `DefaultOperatorRewards` allow the networks' corresponding middleware to distribute rewards by publishing a root of a Merkle tree with `distributeRewards`. That will pull the specified amount and update the root. Then, operators may claim their rewards with `claimRewards` by specifying the amount in the tree that is encoded with their address along with the proof. As a result, funds are transferred out. Further, note that the `recipient` is the address receiving funds but not the eligible address. Note that a leaf in the tree is encoded as the hash of a user eligible to claim encoded with the claimable amount. Note that the claimable amount should be the sum of all rewards received so far. Also, see Root update considerations.

The `DefaultOperatorRewardsFactory` allows deploying a minimal proxy pointing to the `DefaultOperatorRewards` implementation in a standardized way. Note that the factory acts as a `Registry` (see core audit) which allows querying for entities.

2.2.2 Default Staker Rewards

The `DefaultStakerRewards` are vault-specific. Namely, the networks' corresponding middleware can distribute rewards to stakers of a vault with `distributeRewards`. Note that the distribution is based on timestamps. More specifically, the middleware must provide the timestamp encoded in `data`. Then, the amounts will be claimable according to the distribution of the shares at the given time.

Note that

- only whitelisted networks may distribute rewards (networks trusted by the vault owner)
- rewards can only be distributed retrospectively (due to the accounting)
- rewards can only be distributed if and only if shares and stake in the vault existed in the vault (due to DoS vectors and meaningfulness)

All distributions are tracked per token in a list of distributions. The rewards can be claimed by stakers with `claimRewards` which iterates through the list of reward distributions for a given token, starting from the earliest (by push time) reward distribution and iterating up to the last reward. Note that a user can encode a maximum number of iterations as part of `data` (`maxRewards` in code). Also, note that when a user claims rewards, the `activeSharesOfHints` (hints provided for more efficient checkpointing queries) should optimally be the size of `maxRewards` and that `maxRewards` should be less than or equal to the number of rewards available to the user when sending the transaction (otherwise the transaction may revert unnecessarily). Further, the `recipient` is the address receiving funds but not the eligible address.

The rewards are distributed based on the checkpointed active-shares accounting at a given timestamp. Namely, they are distributed as follows (simplified version not considering precision):

$$\frac{rewardAmount * shares_{user, t_i}}{totalShares_{t_i}}$$

Note that admin fees, which can be set to up to 100% by ADMIN_FEE_SET_ROLE with setAdminFee, are set aside for the staker rewards when rewards are distributed, and can be claimed by ADMIN_FEE_CLAIM_ROLE with claimAdminFee. Further, networks can be whitelisted with setNetworkWhitelistStatus by NETWORK_WHITELIST_ROLE.

Notably, the staker rewards include a version to standardize different implementation interfaces with versions. The version of the implemented contract is 1.

The DefaultStakerRewardsFactory allows to permissionlessly deploy a minimal proxy pointing to the DefaultStakerRewards implementation. Note that all roles are initially given to the specified vault's owner.

See also Staker reward distribution considerations

2.2.3 Changes in Version 2

Notable changes in version 2 are:

- The reward distributions are now tracked per network-token pair.
- The whitelisting of networks has been removed as the accounting is done per network now (and not aggregated in a shared array).
- The data parameter now encodes more data for distributeRewards and claimRewards (additional hints and fee slippage protection for the first one, and the target network for the second one).

2.2.4 Roles and Trust Model

The default operator rewards and its factory have the following roles defined:

- Middleware: The middleware is the address distributing the rewards. Generally, it is untrusted - no middleware should affect other networks' distributions. However, given these boundaries, it is trusted to behave as expected. Otherwise, funds may be unclaimable (or special tokens might purposefully be designed to revert (or similar). However, with the constraint that other tokens should not be affected). Thus, this relates also to the off-chain computations done.
- Middleware service: Trusted to honestly return the middleware of a given network.
- Users: Users are untrusted.
- Tokens: The contract will only work with normal tokens. Very exotic tokens might lead to reverts but should not affect executions with other tokens. Note that blacklistable tokens may block the interaction of users with the token contract (or the contract itself). Note that rebasing tokens are not supported.

The default staker rewards and its factory have the following roles defined:

- Middleware: The middleware is the address distributing the rewards. Generally, it is untrusted - no middleware should affect other networks' distributions. However, given these boundaries, it is trusted to behave as expected. Otherwise, funds may be unclaimable (or special tokens might purposefully be designed to revert (or similar). However, with the constraint that other tokens should not be affected).
- Middleware service: Trusted to honestly return the middleware of a given network.
- Users: Users are untrusted.
- Vault: Expected to work correctly and honestly. Namely, the checkpointed accounting is assumed to be correct and expected to not be modifiable.
- DEFAULT_ADMIN_ROLE: Trusted to assign the fee setter and claimer roles.

- `ADMIN_FEE_SET_ROLE`: Trusted to set reasonable fees.
- `ADMIN_FEE_CLAIM_ROLE`: Trusted to specify a proper recipient. However, claiming admin fees should have no impact on the system.
- Tokens: The contract will only work with normal tokens. Very exotic (e.g. token that always reverts) tokens might lead to reverts but should not affect executions with other tokens. Note that blacklistable tokens may block the interaction of users with the token contract (or the contract itself). Note that rebasing tokens are not supported.

Please also see Notes for further consideration.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1

- Unskippable Rewards Risk Accepted

5.1 Unskippable Rewards

Design Low Version 1 Risk Accepted

CS-SYMB-REW-004

While typically the networks integrating with a default staker contract are trusted, the networks might collectively (but unintentionally) DoS a user from claiming a reward (or technically would make it bothersome to claim rewards).

Consider the following scenario:

1. Rewards are published for a `token` 10000 times.
2. A staker joins the system.
3. A staker is eligible for a reward distribution.
4. The last claimable index is 0. However, his first interaction with the system has been just recently. For most rewards, the staker is not eligible.
5. Nevertheless, the staker must iterate over all items in the array for the to-be-claimed token which may inflict unnecessary gas costs.

Ultimately, users cannot skip large sets of reward distributions for which they might be ineligible for. That might disincentivize users from claiming certain rewards.

Risk accepted:

Symbiotic accepts the risk. However, note that impact of the issue has been reduced by separating the accounting of networks. Nevertheless, the scenario above can occur for regular networks.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	1

- Stealing Operator Rewards Code Corrected

High-Severity Findings	0

Medium-Severity Findings	0

Low-Severity Findings	2

- Potential Reverts Due to Strict Inequality Code Corrected
- Reentrant Tokens Code Corrected

Informational Findings	4

- DefaultOperatorRewards Contract Inherits From Initializable Code Corrected
- Incomplete and Inaccurate NatSpec Code Corrected
- Merkle Tree Leaf Best Practices Code Corrected
- Incomplete Sanity Check in Initializer Code Corrected

6.1 Stealing Operator Rewards

Security Critical Version 1 Code Corrected

CS-SYMB-REW-001

Operator rewards can be stolen by malicious parties. Namely, there is no enforcement that the sum of tokens transferred in by a network is greater than or equal to the sum of claims (see Root update considerations for further additional considerations). Thus, it is possible to publish a root where one user can claim all of the contract's balances.

Consider the following example:

1. Regular rewards are deposited into the contract (e.g. 1M USD in WETH).
2. An attacker fully registers a network.
3. The malicious network's middleware calls `distributeRewards` with `amount=0` and `token=WETH`.
4. The tree has only one leaf which is the leaf encoding of 1M USD in WETH with the attacker's address.
5. The attacker claims rewards with the corresponding amount.
6. The rewards contract is drained.

Ultimately, the operator rewards can be stolen.

Code corrected:

Deposited balances are now tracked and it is enforced that claims for a network can not exceed its "deposits".

6.2 Potential Reverts Due to Strict Inequality

Design Low Version 1 Code Corrected

CS-SYMB-REW-002

When rewards are claimed, the size of the hints must be equal to `rewardsToClaim`. However, `rewardsToClaim` may depend on the length of `rewards[token]`. Ultimately, the dynamic nature of the array length might lead to unnecessary reverts. Consider the following example:

1. User A sees that `rewards[token] == 1` and thus provides hints of length `1` (assume `maxRewards` is higher). A transaction is sent.
2. Right after, a transaction arrives that pushes to the array increasing its size to 2.
3. User A's transaction arrives and reverts due to only one hint being provided.

Ultimately, executions might unnecessarily revert.

Code correct:

While reverts can still occur, there is now also the possibility to specify empty hints which can also mitigate the issue.

6.3 Reentrant Tokens

Design Low Version 1 Code Corrected

CS-SYMB-REW-003

Reentrant tokens are expected to integrate with the system. However, reentrant tokens paired with slightly exotic middleware may lead to fewer tokens being distributed than intended.

Consider the following scenario and setup:

1. Assume the middleware supports the ERC777 before-transfer-hook which pulls funds from a pool and pays a keeper with a small amount.
2. The middleware is used by a malicious keeper, `DefaultStakerRewards.distributeRewards` is called and, ultimately, the keeper gains control over the execution. The intended amount by the middleware would be `X`.
3. Then, the keeper calls `claimRewards` (not reentrancy protected) to claim `X-1` rewards.
4. Ultimately, the balance difference to be distributed will be `1` instead of `X`.

Note that in very exotic setups such scenarios could occur. However, typically, such occurrences should not be expected.

Code corrected:

Reentrancy protection has been introduced for other relevant functions.

6.4 Incomplete Sanity Check in Initializer

Informational Version 2 Code Corrected

CS-SYMB-REW-010

The initializer of `DefaultStakerRewards` reverts as below

```
if (
    params.defaultAdminRoleHolder == address(0) && params.adminFee != 0
        && params.adminFeeClaimRoleHolder == address(0)
) {
    revert MissingRoles();
}
```

However, note that this does not cover the scenarios where

- `adminFee` is zero and there is no fee claimer but there is a fee setter. Ultimately, fees could be set but never could be claimed.
- `adminFee` is zero and there is no fee setter but there is a claimer. Ultimately, the claimer role would be meaningless.

Code corrected:

The code has been adjusted to revert in the scenarios mentioned above. Further, note that with `renounceRole` it could be possible to get into the unwanted scenarios. However, that is expected to be the responsibility of the roles.

6.5 Incomplete and Inaccurate NatSpec

Informational Version 1 Code Corrected

CS-SYMB-REW-005

The NatSpec is incomplete or potentially inaccurate in some cases. Below is a (potentially incomplete) list of examples:

1. The `@dev` of the event `DistributeRewards` describes that leaves of the Merkle tree must represent an amount, a token and a claimable amount. However, leaves do not represent tokens. Instead, each network-token pair has its own tree.
2. `IDefaultOperatorRewards.root` lacks a `@return`.
3. `IDefaultOperatorRewards.claimed` lacks a `@return`.
4. `IDefaultStakerRewards.isNetworkWhitelisted` lacks a `@return`.
5. `IDefaultStakerRewards.ADMIN_FEE_CLAIM_ROLE` lacks a `@return`.
6. `IDefaultStakerRewards.NETWORK_WHITELIST_ROLE` lacks a `@return`.
7. `IDefaultStakerRewards.ADMIN_FEE_SET_ROLE` lacks a `@return`.

Code correct:

The NatSpec has been adjusted.

6.6 Merkle Tree Leaf Best Practices

Informational **Version 1** **Code Corrected**

CS-SYMB-REW-006

Note that best practices exist for defining Merkle trees. Namely, that is due to second preimage attack possibilities if the attacker can control all of the 64 bytes of a 64-byte sized leaf. While that is not the case for the tree used, it is nevertheless considered best practice to implement protective mechanism. For further details, consider the following post.

Code corrected:

Now, a double-keccak256 is done.

6.7 `DefaultOperatorRewards` Contract Inherits From `Initializable`

Informational **Version 1** **Code Corrected**

CS-SYMB-REW-008

`DefaultOperatorRewards` does not implement an initializer, thus modifiers from the `Initializable` contract are never used. The call to `_disableInitializers()` in the constructor brings overhead that could be avoided.

Code corrected:

The contract implements reentrancy locks as of version 2. Thus, `Initializable` is now inherited through the reentrancy guard library. However, it is required.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Event Emission When Setting adminFee

Informational Version 2 Acknowledged

CS-SYMB-REW-009

In the `DefaultStakerReward` contract, the event `SetAdminFee` is emitted in the public function `setAdminFee` after the admin fee has been set via the private function `_setAdminFee`. This does not allow tracking the first time the admin fee is set in `initialize` via the respective event.

Acknowledged:

Symbiotic stated that other initializers similarly do not emit (intentional) events. Thus, there is no event emission for consistency reasons.

7.2 Reward Distribution Is Not Dustless

Informational Version 1 Risk Accepted

CS-SYMB-REW-007

The reward distribution distributes rewards as follows

```
uint256 claimedAmount = IVault(VAULT).activeSharesOfAt(msg.sender, reward.timestamp, activeSharesOfHints[i])
.mulDiv(reward.amount, _activeSharesCache[reward.timestamp]);
```

Note that the computation is not dustless. Consider the following scenario:

1. There are two users with Alice having `10**18` and Bob having `1` active shares at the given timestamp.
2. The reward at the given timestamp is `10**18`.
3. The computation for Alice would return `99...999`.
4. The computation for Bob would return `0`. Ultimately, dust could remain locked in the contract.

Risk Accepted:

Symbiotic accepted the risk.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Root Update Considerations

Note Version 1

The `DefaultOperatorRewards` receive a Merkle Tree root. To ensure correctness, the following considerations should be made off-chain when updating the root.

1. A root update should imply that `reward@root_i <= reward@root_{i+1}` since the leafs encode the total rewards accrued which should never decrease. Otherwise, the computation `totalClaimable - claimed_` may revert due to an underflow.
2. The sum of leafs should equal the sum of overall deposits. If the sum of amounts encoded in leafs is less than the sum of overall deposits for a given network-token pair, some of the rewards transferred in may remain unused. The opposite case, we described as part of issue Stealing Operator Rewards.
3. The tree encoding should consider special tokens (e.g. fees on transfer) to ensure that the encoding for a token is accurate.

8.2 Staker Reward Distribution Considerations

Note Version 1

The staker rewards are tracked on a per token basis in an ever-growing array which users have to iterate over. The middleware is trusted to not be malicious. However, given that it is undefined, we define some properties that should hold for the middleware below:

1. The middleware should be permissioned. Namely, the middleware could spam many 1 wei rewards for a token to discourage users from claiming real rewards. The vault owner should carefully consider which networks to whitelist. Otherwise, rewards may be lost.
2. The middleware should not allow for unintentional spamming.

As of Version 2, a networks middleware cannot affect the reward distribution of other networks. Thus, another network cannot spam another one. Nevertheless, networks should ensure that the middleware does not allow spamming rewards for their reward tokens.