SMART CONTRACTS REVIEW

zokyo

January 3rd 2025 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING SYMBIOSIS

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: 0 points

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 0 High issue: 0 points deducted
- 3 Medium issues: 3 resolved = 0 points deducted
- 5 Low issues: 3 resolved and 2 acknowledged = - 2 points deducted
- 1 Informational issue: 1 resolved = 0 points deducted

Thus, 100 - 2 = 98

TECHNICAL SUMMARY

This document outlines the overall security of the Symbiosis smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Symbiosis smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Symbiosis team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Symbiosis repository:
Repo: https://github.com/symbiosis-finance/symbiotic-middleware/commit/9dc212e5e079dea7ab89572a0cc4b43fc722d708

Last commit -ce819405360da2bd85bc7671980fd7d46c6be00c

Contract/s under the scope:

- SymbiosisMiddleware.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Symbiosis smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Symbiosis team and the Symbiosis team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Missing gap to Avoid Storage Collisions	Medium	Resolved
2	Missing call to _disableInitializers	Medium	Resolved
3	Incorrect Operator Validation in registerOperator Function	Medium	Resolved
4	Lack of Two-Step Ownership Transfer	Low	Resolved
5	The owner can renounce ownership	Low	Resolved
6	Risk of Centralisation	Low	Acknowledged
7	Excessive Loops in SymbiosisMiddleware Functions	Low	Acknowledged
8	Missing Event Emission	Low	Resolved
9	Lack of Operator Existence Verification in getOperatorCollaterals and getOperatorStake Functions	Informational	Resolved

MEDIUM-1 | RESOLVED

Missing `gap` to Avoid Storage Collisions

The `SymbiosisMiddleware` contract is intended to be an upgradeable smart contract, but do not have a __gap variable.

In upgradeable contracts, it's crucial to include a `_gap` to ensure that any additional storage variables added in future contract upgrades do not collide with existing storage variables. This is especially important when inheriting from multiple upgradeable contracts.

Recommendation:
Include a `_gap` as the last storage variable to `SymbiosisMiddleware` contract to reserve space for future storage variables and prevent storage collisions. This is a common practice to ensure compatibility and avoid issues when upgrading the contract in the future.

MEDIUM-2 | RESOLVED

Missing call to _disableInitializers

An uninitialized contract can be taken over by an attacker. This applies to both a proxy and its implementation contract, which may impact the proxy. To prevent the implementation contract from being used, you should invoke the `_disableInitializers()` function in the constructor to automatically lock it when it is deployed, more information can be found here.

Recommendation:
Consider calling `_disableInitializers()` in the constructor.

Incorrect Operator Validation in registerOperator Function

Description:
In the registerOperator function of the SymbiosisMiddleware contract, there is a critical bug in the operator validation process. The function is incorrectly checking if the msg.sender (the contract owner) is registered in the OPERATOR_REGISTRY instead of verifying the operator parameter. This bug allows the registration of potentially invalid operators, as long as the contract owner is registered in the OPERATOR_REGISTRY.

Current implementation:

```
if (!IRegistry(OPERATOR_REGISTRY).isEntity(msg.sender)) { // BUG HERE
    revert NotOperator();
}
```

Recommendation:
Replace the incorrect validation check with the following code:

```
if (!IRegistry(OPERATOR_REGISTRY).isEntity(operator)) {
    revert NotOperator();
}
```

This change ensures that the function verifies the validity of the operator being registered, rather than the contract owner.

LOW-1 | RESOLVED

Lack of Two-Step Ownership Transfer

The `SymbiosisMiddleware` contracts does not implement a two-step process for transferring ownership. In its current state, ownership can be transferred in a single step, which can be risky as it could lead to accidental or malicious transfers of ownership without proper verification.

Recommendation:
Implement a two-step process for ownership transfer where the new owner must explicitly accept the ownership. It is advised to use OpenZeppelin's Ownable2StepUpgradeable.

LOW-2 | RESOLVED

The owner can renounce ownership

The `OwnableUpgradeable` contracts includes a function named `renounceOwnership()` which can be used to remove the ownership of the contract.
If this function is called on the `SymbiosisMiddleware` contract, it will result in the contract becoming disowned. This would subsequently break several critical functions of the protocol that rely on `onlyOwner` modifier.

Recommendation:
override the function to disable its functionality, ensuring the contract cannot be disowned e.g.

```
function renounceOwnership() public override onlyOwner {
revert ("renounceOwnership is disabled");
}
```

Risk of Centralisation

The current implementation grants significant control to the owner through multiple functions that can alter the contract's state and behavior. This centralization places considerable trust in a single entity, increasing the risk of potential misuse.

If the owner's private key is compromised, an attacker could execute any function accessible to the owner, potentially leading to fund loss, contract manipulation, or service disruption.

Recommendation:
To enhance security and reduce the risk of a single point of failure, it is recommended to implement a multi-signature wallet for executing owner functions.

Client comments:
- The owner of the contract will be the multisig address;
- We have added a slashing mechanism to reduce the risk of an economic attack on our protocol from operators.

LOW-4 | ACKNOWLEDGED

Excessive Loops in `SymbiosisMiddleware` Functions

The contract contains functions, such as `getOperatorCollaterals` and `getOperatorStake`, that perform extensive iterations over the vaults EnumerableSet. These loops directly iterate over the total number of vaults and invoke external calls within each iteration. The gas cost for executing these functions increases linearly with the number of vaults in the set, potentially leading to out-of-gas errors or exceeding block gas limits in scenarios with a large number of vaults. This inefficiency could disrupt contract operations and prevent users from successfully executing these functions in production environments.

Recommendation:
Introduce mappings or indexed storage to facilitate direct access to operator-specific or collateral-specific data.

Client comment: Since there will be a limited number of registered vaults in the Middleware contract (highly likely < 20), loop through them in the view function should not be a problem

LOW-5 | RESOLVED

Missing Event Emission

functions in the `SymbiosisMiddleware` contract do not emit events to log state changes. This omission reduces the transparency and traceability of these operations, which can hinder off-chain monitoring.

Recommendation:
Introduce event definitions for vault and operator registration/unregistration actions. Emit these events at the end of each respective function to ensure that all state changes are logged on-chain.

Lack of Operator Existence Verification in getOperatorCollaterals and getOperatorStake Functions

Description:
The getOperatorCollaterals and getOperatorStake functions in the SymbiosisMiddleware contract do not verify whether the provided operator address is registered within the system before performing operations. This oversight could lead to misleading results and potential security risks.

Recommendation:
Modify both functions to include a check for operator registration before proceeding with the stake calculations. This can be done by using the operators EnumerableSet:

```
function getOperatorCollaterals(address operator) public view returns
(address[] memory, uint256[] memory) {
    require(operators.contains(operator), "Operator not registered");
    // ... rest of the function ...
}

function getOperatorStake(address operator, address collateral) public
view returns (uint256 amount) {
    require(operators.contains(operator), "Operator not registered");
    // ... rest of the function ...
}
```

Alternatively, if you want to maintain the current function signature, you could return zero values for unregistered operators:

	SymbiosisMiddleware.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying TRC-20)	Pass

We are grateful for the opportunity to work with the Symbiosis team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Symbiosis team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo