SMART CONTRACTS REVIEW

zokyo

April 28th 2025 | v. 1.0

Security Audit Score

PASS

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

ZOKYO AUDIT SCORING YIELDNEST

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 issues: 0 points deducted
- 1 Medium issue: 1 acknowledged = - 5 points deducted
- 3 Low issues: 2 acknowledged and 1 resolved = - 7 points deducted
- 6 Informational issues: 5 unresolved and 1 acknowledged = 0 points deducted

Thus, 100 - 5 - 7 = 88

TECHNICAL SUMMARY

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

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

Contract Status

There were 0 critical issue 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 Yieldnest 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 Yieldnest repository:
Repo: https://github.com/yieldnest/yieldnest-protocol/pull/240

The last commit - d8306f624222490a2e423d9811c2f4dfc2d24f4d

Contracts under the scope:

- src/ynEigen/EigenStrategyManager.sol
- src/ynEigen/TokenStakingNode.sol
- src/ynEigen/TokenStakingNodesManager.sol
- src/ynEigen/AssetRegistry.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 Yieldnest 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

The EigenStrategyManager is responsible for managing strategies within the Eigenlayer protocol, facilitating deposits and withdrawals into various strategies. The TokenStakingNodesManager oversees the operations of staking nodes, ensuring proper role management and interaction with the delegation manager. The TokenStakingNode contract implements staking functionalities, allowing token staking, delegation, and rewards management. It interacts with the Eigenlayer protocol to deposit assets, manage staking operations, and handle rewards, ensuring synchronization with the delegation manager for accurate state tracking. Together, these components enable efficient and secure token staking and strategy management within the Eigenlayer ecosystem.

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 Yieldnest team and the Yieldnest 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	Incorrect Max Node Count Enforcement in TokenStakingNodesManager	Medium	Acknowledged
2	Division by Zero Vulnerability in Asset Conversion Functions	Low	Acknowledged
3	Transfer Inside a loop may cause transactions to revert	Low	Acknowledged
4	Assets can not be enabled directly after getting disabled	Low	Unresolved
5	Floating Pragma	Informational	Acknowledged
6	Duplicate Modifier Execution	Informational	Unresolved
7	Unused Struct Declaration	Informational	Unresolved
8	Unused function parameter	Informational	Unresolved
9	Unused variables	Informational	Unresolved
10	Incorrect comment	Informational	Unresolved

Incorrect Max Node Count Enforcement in TokenStakingNodesManager

Location: TokenStakingNodesManager.sol

The setMaxNodeCount() function in the TokenStakingNodesManager contract allows setting a maximum node count (maxNodeCount) that is smaller than the number of currently active nodes. This creates an inconsistent state where:

1. New nodes cannot be created because the max limit is exceeded.

2. Existing nodes beyond the new limit remain active, violating the intended constraint.

This issue arises because there is no validation to ensure that _maxNodeCount is greater than or equal to the current number of active nodes (nodes.length).

Code Evidence
The vulnerable code in setMaxNodeCount():

```
function setMaxNodeCount(uint256 _maxNodeCount) public
onlyRole(STAKING_ADMIN_ROLE) {
    maxNodeCount = _maxNodeCount; // No validation for current node count
    emit MaxNodeCountUpdated(_maxNodeCount);
}
```

The function directly updates maxNodeCount without checking whether _maxNodeCount is sufficient to accommodate the existing nodes.

Recommended Fix
Add a validation check to ensure that _maxNodeCount is at least equal to the current number of active nodes:

```
function setMaxNodeCount(uint256 _maxNodeCount) public
onlyRole(STAKING_ADMIN_ROLE) {
    require(_maxNodeCount >= nodes.length, "Invalid max node count: fewer
than existing nodes");

    maxNodeCount = _maxNodeCount;
    emit MaxNodeCountUpdated(_maxNodeCount);
}
```

LOW-1 | ACKNOWLEDGED

Division by Zero Vulnerability in Asset Conversion Functions

Location: AssetRegistry.sol

there's no validation to ensure the assetRate is non-zero before performing division. If an oracle returns a zero rate (which can happen during market disruptions), this would cause a division by zero runtime error and potentially break core protocol functionality.

Impact

If a price oracle temporarily returns a zero value for any asset:

1. The convertFromUnitOfAccount function will revert with a division by zero error
2. Any contract functions that rely on this conversion will fail
3. Key protocol operations including total value calculations, deposits, withdrawals, and rebalancing could be disrupted
4. Users might be unable to withdraw funds during market turbulence

The issue is particularly concerning because market volatility (when users most need access to their funds) is exactly when price oracles are most likely to fail or return extreme values.

Code

```
 function convertFromUnitOfAccount(IERC20 asset, uint256 amount) public
view returns (uint256) {
    uint256 assetRate = rateProvider.rate(address(asset));
    uint8 assetDecimals = IERC20Metadata(address(asset)).decimals();
    return assetDecimals != 18
        ? amount * (10 ** assetDecimals) / assetRate  // Division by
assetRate with no zero check

        : amount * 1e18 / assetRate;                  // Division by
assetRate with no zero check
}
```

Recommended Fix

Add a zero-check in both conversion functions to ensure division safety:

```
function convertFromUnitOfAccount(IERC20 asset, uint256 amount) public
view returns (uint256) {
    uint256 assetRate = rateProvider.rate(address(asset));
    require(assetRate > 0, "Zero rate not allowed");  // Add this check

    uint8 assetDecimals = IERC20Metadata(address(asset)).decimals();
    return assetDecimals != 18
        ? amount * (10 ** assetDecimals) / assetRate
        : amount * 1e18 / assetRate;
}
```

Transfer Inside a loop may cause transactions to revert

The function `_stakeAssetsToNode` in EigenStrategyManager performs `safeTransfer` calls inside a loop. If any transfer fails, the entire transaction reverts, potentially causing issues when handling multiple assets. This can lead to inefficiencies, as a single failing transfer prevents all other valid transfers from proceeding.

```
        for (uint256 i = 0; i < assetsLength; i++) {
            // NOTE: approving also token that will not be transferred
            IERC20(assets[i]).forceApprove(address(_wrapper), amounts[i]);
            (uint256 depositAmount, IERC20 depositAsset) = _wrapper.unwrap(amounts[i],
assets[i]);
            depositAssets[i] = depositAsset;
            depositAmounts[i] = depositAmount;

            // Transfer each asset to the node
            depositAsset.safeTransfer(address(node), depositAmount);
        }
```

Recommendation:
Ensure proper handling of partial failures.

LOW-3 | UNRESOLVED

Assets can not be enabled directly after getting disabled

Description:
The `addAsset()` function within the `AssetRegistry.sol` smart contract does not allow adding an asset whose status is different from Unavailable:

```
if (_assetData[asset].status != AssetStatus.Unavailable) {
            revert AssetAlreadyAvailable(address(asset));
}
```

However, when an asset gets disabled, its status is updated to disabled.

Impact:
If an asset gets disabled, it would not be possible to add it again directly as its status would be different from `Unavailable`. For adding the asset again, it should first get removed from the system by executing `deleteAsset()` which will only work once there is `0` balance in every strategy. This means that it is not possible to disable and enable assets again directly.

Recommendation:
Update the `addAsset()` function to allow disabled assets getting enabled again.

INFORMATIONAL-1 | ACKNOWLEDGED

Floating Pragma

The smart contract uses a floating pragma version `(^0.8.24)`. Contracts should be deployed using the same compiler version and settings as were used during development and testing. Locking the pragma version helps ensure that contracts are not inadvertently deployed with a different compiler version.

Recommendation:
Consider locking the pragma version to a specific, tested version to ensure consistent compilation and behavior of the smart contract.

INFORMATIONAL-2 | UNRESOLVED

Duplicate Modifier Execution

Location: TokenStakingNode.sol

The TokenStakingNode contract has 2 completeQueuedWithdrawals() functions with the same name but different parameters (first one has a struct array parameter and second one has a struct parameter) and both functions have the same modifiers (onlyTokenStakingNodesWithdrawer and onlyWhenSynchronized).
However, the second function internally calls the first function.
As a result, the modifiers are executed twice during a single call to the second function.

Recommendation:
To avoid duplicate modifier execution, consider removing modifiers from the second function (having a struct parameter).

INFORMATIONAL-3 | UNRESOLVED

Unused Struct Declaration

Location: TokenStakingNode.sol

The contract declares OperatorStrategyPair struct but it's never used in the contract.
This unused struct increases the codebase size unnecessarily and may cause confusion for developers. While it does not pose a direct security risk, it can reduce code clarity and maintainability.

Recommendation:
Remove the unused struct declaration to improve code clarity and maintainability.

INFORMATIONAL-4 | UNRESOLVED

Unused function parameter

Description:
The `_decreaseQueuedSharesOnCompleteWithdrawals()` function within the `TokenStakingNode.sol` smart contract receives a `_allocationManager` parameter which is not used within the function implementation.

Recommendation:
Remove the unused and not needed function parameters.

INFORMATIONAL-5 | UNRESOLVED

Unused variables.

Description:
The `synchronize()` function within the `TokenStakingNode.sol` smart contract defines 3 unused variables:

```
IAllocationManager allocationManager =
delegationManager.allocationManager();
IStrategy[] memory uniqueStrategies = new
IStrategy[](withdrawals.length);
uint256 uniqueStrategiesLength = 0;
```

Recommendation:
Remove the variables that are not used and not needed for the function operation.

Incorrect comment.

Description:
The `deallocateTokens()` function within the `TokenStakingNode.sol` smart contract contains the following comment:

```
* @notice Deallocates tokens from the withdrawn balance and approves them
for transfer.
```

However, the function implementation is not approving tokens for transfers but directly transferring them.

```
function deallocateTokens(IERC20 _token, uint256 _amount) external
onlyYieldNestStrategyManager {
        withdrawn[_token] -= _amount;
        _token.safeTransfer(msg.sender, _amount);

        emit DeallocatedTokens(_amount, _token);
    }
```

Recommendation:
Fix the comment to describe the actual function implementation.

	src/ynEigen/EigenStrategyManager.sol src/ynEigen/TokenStakingNode.sol src/ynEigen/TokenStakingNodesManager.sol src/ynEigen/AssetRegistry.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 ERC-20)	Pass

We are grateful for the opportunity to work with the Yieldnest 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 Yieldnest team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo