SMART CONTRACTS REVIEW

zokyo

February 4th 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
- 0 Medium issues: 0 points deducted
- 0 Low issues: 0 points deducted
- 7 Informational issues: 3 resolved and 4 acknowledged = 0 points deducted

Thus, the score is 100

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/commit/efda4776f5a043003f45795853969cec90e0c2ac

Fixes - https://github.com/yieldnest/yieldnest-protocol/pull/188

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- StakingNode.sol
- StakingNodesManager.sol
- EigenStrategyManager.sol
- TokenStakingNode.sol
- TokenStakingNodesManager.sol
- ArrayLib.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

From Yieldnest Team: Yieldnest's contract assumes that delegation and undelegation to the operator can only be done by StakingNode and TokenStakingNode and so it first queueWithdrawals for all the tokens and increments queuedShares(https://github.com/yieldnest/yieldnest-protocol/blob/release-candidate/src/ynEIGEN/TokenStakingNode.sol#L157). Then, delegator role(trusted address) calls undelegate on StakingNode and TokenStakingNode(https://github.com/yieldnest/yieldnest-protocol/blob/release-candidate/src/ynEIGEN/TokenStakingNode.sol#L273-L282) to ensure that accounting of queuedShares is done properly. Calling undelegate directly on StakingNode and TokenStakingNode doesn't account for the queuedWithdrawals created which can break the accounting of queuedShares if all the strategies of the Node isn't queued before undelegating by Delegator role. Calling completeQueuedWithdrawals would also revert in case of undelegation since it's querying operator from eigenlayer's undelegationManager which maybe changed from the time the withdrawals were queued and hence withdrawal roots would differ in delegationManager.

But in eigenlayer's DelegationManager contract, operator can also call undelegation for staker(https://github.com/Layr-Labs/eigenlayer-contracts/blob/bda003385c5fec59e35196dc14d01f17d1eb7001/src/contracts/core/DelegationManager.sol#L228-L231). This will queue withdrawals for all the delegated strategies of the staker as mentioned above. Due to this, all the deposits of staker will be queued for withdrawals but this won't be accounted into queuedShares of StakingNode and TokenStakingNode contract. Hence, share mechanism is not synchronized which will break accounting. Calling undelegate on StakingNode and TokenStakingNode won't work after desynchronization because all the withdrawals for the previous operator which undelegated is queued and there is no operator to undelegate from.
Changes made:

Added a function to ensure that StakingNode and TokenStakingNode is synchronized before performing actions on StakingNode and TokenStakingNode. The contract will be considered as synchronized if the existing delegation in eigenlayer DelegationManager contract is same as delegatedTo stored in the contract. Apart from this, added a function by trusted actor which processes QueuedWithdrawals and increments queuedShares. Also, added a function to complete QueuedWithdrawals and reinvest the shares to new operator(if any).
Also added a new library named ArrayLib. It currently has one function named dedupilcate which takes array and returns array of unique elements in the array. This is being used in completeQueueWithdrawals in TokenStakingNode where we can batch withdrawals and if there is same strategies in different withdrawal struct, we dedupicate it for accounting purposes.

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	Excessive Loops in updateTotalETHStaked	Informational	Acknowledged
2	State Variable Synchronization	Informational	Acknowledged
3	Implementation Upgrade Issues	Informational	Acknowledged
4	Cache the state value into a memory	Informational	Resolved
5	Payable modifier could be removed	Informational	Resolved
6	Missing a zero address check	Informational	Resolved
7	Potential Out-of-Gas Error Due to Unbounded Memory Allocation	Informational	Acknowledged

Excessive Loops in `updateTotalETHStaked`

The `updateTotalETHStaked` function iterates over all staking nodes using a loop that grows linearly with the number of nodes. If the number of nodes becomes too large, this loop can lead to excessive gas consumption, potentially making the function fail due to out-of-gas errors. This can prevent updates to `totalETHStaked`, leading to inaccurate tracking of staked ETH. Additionally, the requirement that all nodes be synchronized before proceeding means that even a single unsynchronized node can revert the function, blocking updates entirely.

```
function updateTotalETHStaked() public {
    uint256 updatedTotalETHStaked = 0;
    IStakingNode[] memory allNodes = getAllNodes();
    for (uint256 i = 0; i < allNodes.length; i++) {
        if (!allNodes[i].isSynchronized()) {
            revert NodeNotSynchronized();
        }
        updatedTotalETHStaked += allNodes[i].getETHBalance();
    }

    emit TotalETHStakedUpdated(updatedTotalETHStaked);
    totalETHStaked = updatedTotalETHStaked;
}
```

Recommendation:
Implement a more gas-efficient way to track `totalETHStaked`

Client's Comment: This is design choice and not a vulnerability. We ensure that stakingNodes doesn't grow much and are aware of the iteration over each node to updateTotalETHStaked. We don't consider this as an issue

State Variable Synchronization

Location: TokenStakingNodesManager.sol

The initializeV2() function in the TokenStakingNodesManager contract initializes the rewardsCoordinator state variable:

```
function initializeV2(IRewardsCoordinator _rewardsCoordinator)
    external
    notZeroAddress(address(_rewardsCoordinator))
    reinitializer(2)
{
    rewardsCoordinator = _rewardsCoordinator;
}
```

However, this function does not emit an event to notify listeners of this important state change. This omission can lead to the following issues:

- Dependent contracts or off-chain applications may not be aware of changes to the rewardsCoordinator.
- It becomes difficult to track the history of rewardsCoordinator changes.
- The lack of synchronization notification could result in inconsistent state across the system.

Recommendation:
Add an event to the contract for rewardsCoordinator updates:

Emit this event in the initializeV2() function:

```
{
    rewardsCoordinator = _rewardsCoordinator;
    emit RewardsCoordinatorUpdated(address(_rewardsCoordinator));
}
```

Client Comment: This should be informational severity since there is no off-chain application dependent on rewardsCoordinator address. Not emitting this event doesn't cause any issues in smart contract code.

Implementation Upgrade Issues

Location: StakingNodesManager.sol

1. Insufficient Implementation Validation

While the contract does check for the existence of the upgradeable beacon and uses OpenZeppelin's UpgradeableBeacon contract (which includes basic implementation checks), there's a lack of additional, contract-specific validation for the new implementation.

The function only checks that the provided address is not zero, but doesn't perform any further validation specific to the expected interface or functionality of the new implementation.

2. Lack of Timelock Mechanism

The upgrade process allows for immediate execution of upgrades without any delay or timelock mechanism. This could potentially allow for malicious or accidental upgrades to be executed without giving stakeholders time to review or react.

```
function upgradeStakingNodeImplementation(address _implementationContract)
    public
    onlyRole(STAKING_ADMIN_ROLE)
    notZeroAddress(_implementationContract) {
    _upgradeStakingNodeImplementation(_implementationContract);
}
```

There's no built-in delay between proposing an upgrade and executing it, which is considered a security best practice for upgradeable contracts.

Recommendation:

1. Enhance Implementation Validation:
   - Add checks for implementation compatibility.
   - Verify interface conformance of the new implementation.
   - Implement version control checks to ensure proper upgrade sequencing.
2. Implement a Timelock Mechanism:

```
uint256 public constant UPGRADE_TIMELOCK = 48 hours;
mapping(address => uint256) public pendingImplementations;

function proposeUpgrade(address newImplementation) external
onlyRole(STAKING_ADMIN_ROLE) {
    pendingImplementations[newImplementation] = block.timestamp +
UPGRADE_TIMELOCK;
    emit UpgradeProposed(newImplementation);
}

function executeUpgrade(address implementation) external
onlyRole(STAKING_ADMIN_ROLE) {
    require(block.timestamp >=
pendingImplementations[implementation], "Timelock not expired");
    _upgradeStakingNodeImplementation(implementation);
    emit UpgradeExecuted(implementation);
}
```

3. Add Event Emissions:
Emit events for upgrade proposals and executions to improve transparency and allow for off-chain monitoring.

Client Comment: This issue should be informational as this upgrade only needs to be called by trusted role which is our security council.

INFORMATIONAL-4 | RESOLVED

Cache the state value into a memory

Location: StakingNode.sol

The deallocateStakedETH() function reads the withdrawnETH value from storage twice.

Recommendation:
Consider the withdrawnETH value into a memory to save the transaction cost.

INFORMATIONAL-5 | RESOLVED

payable modifier could be removed

Location: StakingNode.sol

The allocateStakedETH() and deallocateStakedETH() functions are made payable even if they are not supposed to receive native coins from the staking nodes manager contract.

Recommendation:
Remove payable modifier in the functions.

INFORMATIONAL-6 | RESOLVED

Missing a zero address check

Location: StakingNodesManager.sol

The initializeV3() function is missing a zero address check for the rewardsCoordinator variable.

Recommendation:
Add a zero address check for the rewardsCoordinator variable so that it can never be set to zero.

Potential Out-of-Gas Error Due to Unbounded Memory Allocation

Location: ArrayLib.sol
Description

The function allocates memory for the result array based on the count of unique elements, without any upper bound. For very large input arrays, this could consume excessive gas or cause out-of-gas errors.

Recommendation:
Implement a maximum array size limit and revert if exceeded:

```
uint256 constant MAX_ARRAY_SIZE = 1000; // Adjust as needed

function deduplicate(address[] memory arr) internal pure returns
(address[] memory) {
    require(arr.length <= MAX_ARRAY_SIZE, "Input array too large");
    // ... rest of the function
}
```

Client Comment: This won't be the issue in context of codebase because it's called in completeQueuedWithdrawals which doesn't contain large number of queuedWithdrawals. This function is called by trusted entity which will take care to not pass large array in argument.

	StakingNode.sol StakingNodesManager.sol EigenStrategyManager.sol TokenStakingNode.sol TokenStakingNodesManager.sol ArrayLib.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