PUBLIC

Code Assessment

of the YieldNest Protocol Smart Contracts

April 19, 2024

Produced for

YieldNest

by

ChainSecurity

Contents

1 Executive Summary

Dear YieldNest team,

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

YieldNest implements a liquidity pooling system built on top of EigenLayer, where users can deposit ETH and LSD tokens and earn yield.

The audit found multiple severe issues (for a detailed description see the Resolved Findings section). All severe issues have been fixed accordingly. In summary, we find that the codebase now provides a good level of security.

Yet, the types of issues identified indicated that the code had an insufficient diligent internal review process and meaningful testing. E.g., the critical issues should have been caught as these issues are well-known in vaults. We highlight this to make YieldNest aware that in the event of contract updates, a thorough review and testing process is essential to ensure the security of the codebase. For the current version of the code, we are not aware of any further severe issues, but 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. These measures include, but are not limited to, further unit and integration testing, fuzzing, and a careful roll-out in case significant funds are expected to be held by the new code base.

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	0
High-Severity Findings	4
• Code Corrected	4
Medium-Severity Findings	2
• Code Corrected	2
Low-Severity Findings	8
• Code Corrected	8

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 YieldNest Protocol 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	06 Mar 2024	3061501d3e028ae5274ddeb928fedd02d8135b3a	Initial Version
2	27 Mar 2024	35ab00a645202c8952e32740b90dbcaa57a6af09	Version with fixes
3	28 Mar 2024	ced5c5ff841ecc3eed8896485c5be133e6730592	Second version with INFO fixes
4	08 Apr 2024	4c38ab84df14885fb31a73bf21c51286c895eac3	Third fixes round

For the solidity smart contracts, the compiler version `0.8.24` was chosen.

The following files are in the scope of the review:

```
LSDStakingNode.sol
PlaceholderContract.sol
RewardsDistributor.sol
RewardsReceiver.sol
StakingNode.sol
StakingNodesManager.sol
YieldNestOracle.sol
ynBase.sol
ynETH.sol
ynLSD.sol
interfaces:
    IEigenLayerBeaconOracle.sol
    ILSDStakingNode.sol
    IOracle.sol
    IRewardsDistributor.sol
    IRewardsReceiver.sol
    IStakingNode.sol
    IStakingNodesManager.sol
    IynETH.sol
    IynLSD.sol
external:
    etherfi:
        DepositRootGenerator.sol
```

In (Version 2), `external/etherfi/DepositRootGenerator.sol` was moved to `external/ethereum/DepositRootGenerator.sol`.

In (Version 4), `interfaces/IEigenLayerBeaconOracle.sol` and `interfaces/IOracle.sol` were removed from the codebase.

2.1.1 Excluded from scope

Any contracts that are not explicitly listed above are out of the scope of this review. Third-party contracts and libraries are out of the scope of this review.

3 System Overview

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

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

YieldNest offers a liquidity pooling system built on top of EigenLayer, where users can deposit `ETH` and `LSD` tokens and earn yield. It is implemented with two vaults, one for native staking and one for liquid staking, where the delegation and rewards management is managed by YieldNest. It is important to note that the current implementation integrates with EigenLayer v0.1.0, and thus does not implement withdrawals.

Unless explicitly specified, all the contracts are deployed behind a transparent upgradeable proxy.

3.1 Native (re)staking

3.1.1 ynETH

This is the main entry point for users with `ETH`, they can deposit `ETH` in `ynETH` and receive shares of the vault. By default, the shares are not transferable, this can be changed by the `PAUSER` role. The deposited `ETH` can be pulled by the `StakingNodesManager` to activate new validators on the beacon chain. The shares are expected to increase in value as consensus and execution layers rewards are distributed.

3.1.2 StakingNode

The `StakingNodes` are deployed behind a beacon proxy. Each `StakingNode` mirrors an `EigenPod`, from which it is the owner. The admin of a `StakingNode` is the address bearing the `STAKING_NODES_ADMIN` role in the `StakingNodesManager` contract. The admin can trigger the following actions:

- `verifyWithdrawalCredentials()`: triggers the EigenPod to verify the withdrawal credentials and activate some validators on EigenLayer. This will currently revert, as EigenLayer paused the functionality.
- `delegate()`: Delegates the staked amount to some operator. This will currently revert, as EigenLayer paused the functionality.
- `undelegate()`: undelegates the staked amount. This will currently revert, as EigenLayer paused the functionality.
- `withdrawBeforeRestaking()`: calls `withdrawBeforeRestaking` on the EigenPod, this can be done only before any restaking is done. This will start the delayed withdrawal process of the balance of the EigenPod. This will currently revert, as EigenLayer paused the functionality.

- `claimDelayedWithdrawals()`: claims up to `maxNumWithdrawals` for the EigenPod's owner (`StakingNode`), if any full withdrawal was to be claimed, the amount should be reflected in `withdrawnValidatorPrincipal`. The total claimed amount is then sent to the `StakingNodesManager`, where the principal is sent directly to `ynETH`, and the rewards are sent to the `consensusLayerReceiver` to be distributed.

3.1.3 StakingNodesManager

The `StakingNodesManager` is responsible for deploying new `StakingNodes` and registering new validators. Before any action, the `STAKING_ADMIN` must deploy the upgradeable beacon and set the implementation for the `StakingNodes` with `registerStakingNodeImplementationContract`. The `STAKING_ADMIN` can update implementation with `upgradeStakingNodeImplementation`. Once this is done, the `STAKING_NODE_CREATOR` can create `StakingNodes` with `createStakingNode`, up to `maxNodeCount`. Each of them has an id and creates their EigenPod during the creation process.

When at least one `StakingNode` has been deployed, the `VALIDATOR_MANAGER` can start registering validators with `registerValidators`. The function first checks that the current deposit root matches some expected root passed as arguments, then it will withdraw `#validators * 32 ETH` from `ynETH` and makes the deposit in the beacon chain deposit contract for each of the validators, with their withdrawal credentials pointing to one of the EigenPods managed by one of the `StakingNodes`.

3.1.4 Rewards distribution

The distribution of the consensus and execution layers rewards are managed by the `RewardsDistributor`. The execution layer rewards are first sent to the `executionLayerReceiver` by the validators, and the consensus layer rewards are sent to the `consensusLayerReceiver` by the `StakingNodesManager` through delayed withdrawal and `StakingNode.claimDelayedWithdrawals()` path.

Anyone can call `processRewards()` on the `RewardsDistributor`, the function will pull the `ETH` balances of the two `RewardsReceiver` mentioned above, take a fee on the rewards (10% by default), and send the remaining amount to the `ynETH` contract. For this, `RewardsDistributor` is assumed to have the `WITHDRAWER` role in both of the `RewardsReceivers`.

3.2 Liquid (re)staking

3.2.1 ynLSD

This is the main entry point for users with liquid staking derivatives (LSD), they can deposit their tokens in `ynLSD` and receive shares of the vault. By default, the shares are not transferable, this can be changed by the `PAUSER` role. The deposited tokens can be pulled by the `StakingNodesManager` to activate new validators on the beacon chain. The shares are expected to gain in value as consensus and execution layers rewards are distributed. The tokens that can be deposited are whitelisted and bound to an EigenLayer strategy.

The `ynLSD` contract is also responsible for deploying a new `LSDStakingNode` that will deposit the tokens into EigenLayer. But first, the `STAKING_ADMIN` must deploy the upgradeable beacon and set the implementation for the `LSDStakingNodes` with `registerLSDStakingNodeImplementationContract`. The `STAKING_ADMIN` can update implementation with `upgradeLSDStakingNodeImplementation`. Once this is done, the `LSD_STAKING_NODE_CREATOR` can create some `LSDStakingNodes` with `createLSDStakingNode`, up to `maxNodeCount`.

3.2.2 LSDStakingNode

The `LSDStakingNodes` are deployed behind a beacon proxy. The admin of a `StakingNode` is the address bearing the `LSD_STAKING_NODES_ADMIN` role in the `ynLSD` contract. The admin can trigger the following actions:

- `depositAssetsToEigenlayer()`: pulls from `ynLSD` and deposits the specified amount of the specified LSD into its EigenLayer strategy.
- `delegate()`: delegates the staked amount to some operator. This will currently revert, as EigenLayer paused the functionality.
- `undelegate()`: undelegates the staked amount. This will currently revert, as EigenLayer paused the functionality.

3.3 Trust Model

PROXY_ADMIN_OWNER

The role is assumed to be **fully trusted**.

Responsible for managing and upgrading all transparent upgradeable proxy contracts. This role can change the implementation behind a proxy, allowing for contract upgrades while preserving the contract's address and state.

YieldNest informed us that this will be controlled by a core-team 3/5 multisig.

DEFAULT_ADMIN_ROLE

The role is assumed to be **fully trusted**.

A general administrative role with broad permissions, including potentially managing other roles, updating system parameters, or performing critical system functions in the contracts.

- `RewardsDistributor.setFeesReceiver`
- `RewardsDistributor.setFeesBasisPoints`
- `ynETH.setExchangeAdjustmentRate`

YieldNest informed us that this will be controlled by a core-team 3/5 multisig.

STAKING_ADMIN

The role is assumed to be **fully trusted**.

Responsible for managing staking-related parameters or operations. This address can register and upgrade the implementation contracts `StakingNode``in ``StakingNodesManager` and `LSDStakingNode` in `ynLSD`. It's also able to set the `maxNodeCount`.

Hence, the following functions can be called by the role:

- `StakingNodesManager.registerStakingNodeImplementationContract`
- `StakingNodesManager.upgradeStakingNodeImplementation`
- `StakingNodesManager.setMaxNodeCount`
- `ynLSD.registerLSDStakingNodeImplementationContract`
- `ynLSD.upgradeLSDStakingNodeImplementation`
- `ynETH.setMaxNodeCount`

YieldNest informed us that this will be controlled by a core-team 3/5 multisig.

STAKING_NODES_ADMIN

The role is assumed to be **fully trusted**.

The role is specifically focused on the administration of staking nodes. Manages node-specific administrative tasks where nodes are the contracts with `StakingNode` as the implementation contract. This role is tracked within the `StakingNodesManager` and acts verified using the `onlyAdmin` within `StakingNode`.

The actions triggered by this role are keeper-style actions meant to handle restaking-related operations within `StakingNode.sol`:

- `withdrawBeforeRestaking`
- `claimDelayedWithdrawals`
- `verifyWithdrawalCredentials`
- `delegate`
- `undelegate`

YieldNest informed us that this will be controlled by a 2/3 multisig that is controlled by off-chain backend processes in an automated fashion that decides upon when to skim rewards, process validator withdrawals or delegate/undelegate to different operators.

LSD_RESTAKING_MANAGER

The role is assumed to be **fully trusted**.

This role is specifically focused on the administration of LSD staking nodes. Manages node-specific administrative tasks where nodes are the contracts with implementation `LSDStakingNode`. This role is tracked within the `ynLSD` and acts verified using the `onlyLSDRestakingManager` within `LSDStakingNode`.

The actions triggered by this role are keeper-style actions meant to handle restaking-related operations within `LSDStakingNode`:

- `LSDStakingNode.depositAssetsToEigenlayer`
- `LSDStakingNode.Delegate`
- `LSDStakingNode.Undelegate`

This will be controlled by a 2/3 multisig that is controlled by off-chain backend processes in an automated fashion that decide upon when to batch deposit LSD assets that are deposited into `ynLSD` into Eigenlayer, delegate and undelegate.

VALIDATOR_MANAGER_ROLE

The role is assumed to be **fully trusted**.

Manages validator-specific functions, such as registering or deregistering validators, managing validator sets, or handling validator performance and slashing conditions.

Currently, only `StakingNodesManager.registerValidators` can be called by the role.

Additionally, the following information was provided by YieldNest:

There are several modules available to add on to Gnosis that are relevant to the `VALIDATOR_MANAGER` role. A time lock delay modifier creates a delay between when a transaction is approved and when it can be executed. This delay would provide a safety period to enable transaction rejection and mitigation of potential front-running of root deposits.

Furthermore, transaction role modifiers that restrict the capability of `VALIDATOR_MANAGER EOA` signers to only a specific set of transaction parameters would assist with limiting the capability of this role to a confined set of actions.

YieldNest informed us that this will be controlled by a 2/3 multisig that is controlled by off-chain backend processes in an automated fashion that uses the figment API and those of other staking service providers to provision validators on-chain.

PAUSER_ROLE

The role is assumed to be **fully trusted**.

The role has the authority to pause and unpause certain functions within the system. This role is critical for emergency response or system maintenance, allowing for a halt to operations without affecting the underlying state.

The following functions can be called by the role:

- `ynBase.unpauseTransfers`
- `ynBase.addToPauseWhitelist`
- `ynBase.removeFromPauseWhitelist`
- `ynETH.updateDepositsPaused`

YieldNest informed us that this will be controlled by a core dev 2/3 multisig.

LSD_STAKING_NODE_CREATOR_ROLE

The role is assumed to be **fully trusted**.

Authorized to create new staking nodes within the system (`StakingNodesManager.createStakingNode`, `ynLSD.createLSDStakingNode`).

YieldNest informed us that this role will be controlled by a core dev 2/3 multisig.

ORACLE_MANAGER

The role is assumed to be **fully trusted**.

Manages oracles or data feeds that provide external information to the system. This role is crucial for systems that rely on accurate, real-time data for price feeds from outside the chain.

Allows setting asset price feeds by calling `setAssetPriceFeed`.

YieldNest informed us that this role will be controlled by a core team 3/5 multisig.

WITHDRAWER_ROLE

The role is assumed to be **fully trusted**.

The role is allowed to move all Ether and tokens out of the `RewardsReceiver` contract by calling `transferETH` and `transferERC20`.

YieldNest did not specify the role in more detail.

We also assume the following:

- Chainlink price will always be checked to be returned in 18 decimals. Otherwise, the price feed will not be used.
- `ynLSD` and `ynEth` are behind proxy contracts.
- Yieldnest contracts will not be frozen by EigenLayer
- Slashing is expected to be a very rare event and will not have a significant impact on the project's funds. YieldNest informed us they will create an insurance fund to cover for user's funds in case slashing happens.

3.4 Changes in V2

- the deposits in `ynETH` and `ynLSD` can be paused/unpaused by the `PAUSER` role
- the flow for withdrawals claims was updated and now relies on arbitrary addresses to claim on behalf of the `StakingNode`. The admin of the node is responsible for processing the withdrawals by calling `processWrithdrawals`.

- During deployment of `ynLSD`, a special trusted address `depositBootstrapper` receives the initial shares. This address is trusted to keep its shares as long as the protocol is running to avoid inflation attack in the case withdrawals become available and the pool is emptied.

3.5 Changes in V4

- a function to retrieve the assets locked in the `LSDSTakingNodes` has been added. The function `LSDSTakingNodes.recoverAssets()` can only be called by the `LSD_RESTAKING_MANAGER` and sends the token balance to `ynLSD`.
- the function `StakingNode.processWithdrawals()` has been updated to process only `expectedETHBalance` if available. The difference `balance - expectedETHBalance` will be processed by another transaction.

4 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.

5 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.

6 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
- Correctness: Mismatches between specification and implementation

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	0

7 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	0

High-Severity Findings	4

- Computation of ynLSD.getTotalAssets() Is Wrong Code Corrected
- First Depositor Gets More Shares Code Corrected
- Withdrawals That Are Not Self-Claimed Break the Accounting Code Corrected
- ynLSD Is Vulnerable to Donation Attack Code Corrected

Medium-Severity Findings	2

- Incorrect Balance Transfers With Rebasing Tokens Code Corrected
- Partial Withdrawals Claims Will Fail Code Corrected

Low-Severity Findings	8

- Processing of Withdrawals Can Be DOSed Code Corrected
- Discrepancy in the Value Check for maxAge Code Corrected
- Ignored Return Values Code Corrected
- Initializer Not Disabled Code Corrected
- Oracle Price Sanity Check Code Corrected
- Redundant Functionality Code Corrected
- Uninitialized Reentrancy Guard Code Corrected
- Unused Code Code Corrected

Informational Findings	5

- Remaining Todos Code Corrected
- Overcomplicated Expression Code Corrected
- Complete Events Code Corrected
- Incorrect Natspec Code Corrected
- Missing Natspec Param Definition Code Corrected

7.1 Computation of `ynLSD.getTotalAssets()` Is Wrong

Correctness High Version 1 Code Corrected

CS-YNPROTO-001

The computation of `ynLSD.getTotalAssets()` has two issues:

1. The index used in the inner loop to get the `asset` should be `j` instead of `i`. The current implementation will either revert with an out-of-bound exception, or double count some assets, by adding the balance of token `X` as what should be the balance of token `Y` and ignore others.
2. The current implementation of `LSDStakingNode` does not allow it to use its own token balance, as it will always pull tokens from `ynLSD` and deposit that exact same amount to EigenLayer. This means that outside of a call to `depositAssetsToEigenlayer()` the tokens in each of the `LSDStakingNodes` are locked. If counting them to the `totalAssets` is correct and intended should be re-evaluated.

Put together, the two issues result in a wrong price calculation of the `ynLSD` shares.

Code corrected:

1. The correct index `j` is now used in the loop.
2. The function `LSDStakingNode.recoverAssets()` has been added. The function sends the token balance from an `LSDStakingNode` to `ynLSD`, allowing them to be unlocked from the `LSDStakingNodes` and counted towards the `totalAssets`.

7.2 First Depositor Gets More Shares

Correctness | High | Version 1 | Code Corrected

CS-YNPROTO-002

In `ynETH` and `ynLSD`, the first depositor sees its shares minted `1:1` to its deposited amount. If `exchangeAdjustmentRate > 0`, the following depositors will have their shares minted at a lower ratio, basically gifting some of their deposited amount to the first depositor.

Please provide a detailed description of why would `exchangeAdjustmentRate` be needed and what was the intention.

Code corrected:

The variable `exchangeAdjustmentRate` has been removed from the codebase.

7.3 Withdrawals That Are Not Self-Claimed Break the Accounting

Design | High | Version 1 | Code Corrected

CS-YNPROTO-003

EigenLayer allows claiming withdrawals on behalf of arbitrary addresses. The current implementation of the YieldNest Protocol does not take this into account, and thus any withdrawn amounts that are not claimed through `StakingNode.claimDelayedWithdrawals()` are locked in the `StakingNode`.

The implementation of `StakingNode` can be updated, but the shares of `ynETH` would be underpriced until the accounting is corrected.

Code corrected:

The function `StakingNode.claimDelayedWithdrawals` has been removed from the codebase. A new function `StakingNode.processWithdrawals` has been added, this function expects that the withdrawals are claimed by a third party and will simply send the balance of the `StakingNode` contract to the `stakingNodesManager` for further processing as before. This new function can only be called by the admin.

7.4 `ynLSD` Is Vulnerable to Donation Attack

Design | High | Version 1 | Code Corrected

CS-YNPROTO-004

It is possible for the first user of the pool to steal the next deposited amount. The `ynLSD` reads the balances of its supported `assets` to compute `totalAssets` and the `_convertToShares` function does not add an offset. This makes the contract vulnerable to a donation attack, where the first user mints `1` share and front-runs the second user in their deposit by transferring the deposited amount to `ynLSD` in order to force a minting of `0` shares to the second user.

Example:

1. `ynLSD` is deployed and accepts token `T`. The oracle is assumed to return the following price: `1 T = 1 ETH`.
2. Alice deposits `1 wei` of `T` and receives `1` share for it. Now `totalSupply = 1` and `totalAssets = 1`.
3. Bob sends a transaction to deposit a big amount `X` of `T`.
4. Alice sees the transaction in the mempool and front-runs it with a transfer of amount `X`. Now `totalSupply = 1` and `totalAssets = 1 + X`.
5. Bob's transaction gets executed and the number of shares he receives is $\lfloor \frac{X * totalSupply}{totalAssets} \rfloor = \lfloor \frac{X * 1}{1 + X} \rfloor = 0$.
6. Alice has now `1` share valued at `1 + 2 * X`, and Bob lost his deposit.

Note that if `exchangeAdjustmentRate > 0`, this attack would be cheaper to conduct as the `totalSupply()` would be considered smaller than what it actually is. Then the amount needed to round down to zero the shares of the next deposit is also reduced.

Code corrected:

Upon deployment, the code now enforces a bootstrap deposit of 10 units of `assets[0]` that must be worth at least `1 ether`. The shares of this initial deposit are sent to a trusted address `depositBootstrapper`. Moreover, `exchangeAdjustmentRate` has been removed from the codebase.

7.5 Incorrect Balance Transfers With Rebasing Tokens

Correctness | Medium | Version 1 | Code Corrected

CS-YNPROTO-005

Rebasing tokens like stETH might transfer less than expected. This might get problematic when a contract expects to receive the amount specified in the transfer. E.g., in `ynLSD.deposit` the `safeTransferFrom` might transfer one or two wei less than specified in `amount`. But before, all calculations and the share distribution were done on the assumption that `amount` would be later an asset of the contract. In consequence, this might break the invariant between the amount of shares and assets such that there are shares but no assets.

Code corrected:

In ynLSD YieldNest accepted the risk. In the case of `LSDStakingNode.depositAssetsToEigenlayer` the issue was fixed by querying the pre- and post-balance of the contract before calling `depositIntoStrategy`. We rate this issue as fixed but created a note to document the behavior for `ynLSD`.

7.6 Partial Withdrawals Claims Will Fail

Design | Medium | Version 1 | Code Corrected

CS-YNPROTO-006

The function `StakingNode.claimDelayedWithdrawals` allows the caller to specify the `maxNumWithdrawals`, but `totalClaimable` will always be computed as if `maxNumWithdrawals` was set to `type(uint256).max`. If the caller does not want to claim all the claimable withdrawals, the condition `totalClaimable > claimedAmount` will be evaluated to `true` and the function will revert.

Code corrected:

The function `StakingNode.claimDelayedWithdrawals` has been removed from the codebase. A new function `StakingNode.processWithdrawals` has been added, this function expects that the withdrawals are claimed by a third party and will simply send the balance of the `StakingNode` contract to the `stakingNodesManager` for further processing as before. This new function can only be called by the admin.

7.7 Processing of Withdrawals Can Be DOSed

Security | Low | Version 2 | Code Corrected

CS-YNPROTO-021

The function `StakingNode.processWithdrawals()` expects a precise amount of ETH as its balance, if it differs from this amount the call will revert. The contract assumes it can only receive `ETH` from the `DelayedWithdrawalRouter`, but it is possible to force send `ETH` to the contract with `selfdestruct`.

Code corrected:

The function `StakingNode.processWithdrawals()` has been updated such that only `expectedETHBalance` is processed. The difference between the balance and `expectedETHBalance` can be processed in another transaction.

7.8 Discrepancy in the Value Check for `maxAge`

Design | Low | Version 1 | Code Corrected

CS-YNPROTO-007

In YieldNestOracle, the value of maxAge is required to be > 0 in setAssetPriceFeed, but no such check is done in the constructor.

Code corrected:

The value of maxAge is now checked during construction of YieldNestOracle and redundancies have been resolved by reusing the code that was present in setAssetPriceFeed.

7.9 Ignored Return Values

Design | Low | Version 1 | Code Corrected

CS-YNPROTO-008

The following calls ignore the returned value:

- LSDStakingNode.depositAssetsToEigenlayer does not check the return value by asset.approve
- ynLSD.retrieveAsset does not check the return value from IERC20(asset).transfer

---

Code corrected:

The OpenZeppelin library SafeERC20 is used for ERC20 interactions in LSDStakingNode.depositAssetsToEigenlayer (forceApprove) and in ynLSD.retrieveAsset (safeTransfer).

7.10 Initializer Not Disabled

Security | Low | Version 1 | Code Corrected

CS-YNPROTO-009

Proxy implementation contracts inheriting OpenZeppelin's Initializable contract should call _disableInitializers in their constructor to prevent initialization and re-initialization of the implementation contract.

Code corrected:

All contracts inheriting OpenZeppelin's Initializable contract now call _disableInitializers in their constructor.

7.11 Oracle Price Sanity Check

Design | Low | Version 1 | Code Corrected

CS-YNPROTO-010

In YieldNestOracle.getLatestPrice the price returned by the oracle is not further checked if it might be, e.g., zero.

Code corrected:

The function has been updated such that the call reverts if `price <= 0`.

7.12 Redundant Functionality

Design | Low | Version 1 | Code Corrected

CS-YNPROTO-011

The functions `ynBase.pauseWhiteList` and `ynBase.isAddressWhitelisted` have the same logic.

Code corrected:

The function `isAddressWhitelisted` was removed.

7.13 Uninitialized Reentrancy Guard

Security | Low | Version 1 | Code Corrected

CS-YNPROTO-012

The `StakingNode` contract inherits `ReentrancyGuardUpgradeable`. But does not call `__ReentrancyGuard_init()` in `initialize`.

Code corrected:

The `initialize` function has been updated to call `__ReentrancyGuard_init()`.

7.14 Unused Code

Design | Low | Version 1 | Code Corrected

CS-YNPROTO-013

Some parts of the codebase are never used. To ease the comprehension of the code, it is good practice to keep it in its minimal form. Here is a non-exhaustive list of unused code:

1. the errors `MinimumStakeBoundNotSatisfied`, `StakeBelowMinimumynETHAmount`, `DepositAllocationUnbalanced` in `StakingNodesManager`
2. the errors `MinimumStakeBoundNotSatisfied`, `StakeBelowMinimumynETHAmount` in `ynETH`
3. the errors `error InvalidConfiguration`, `error NotOracle` and `error Paused` in `RewardsDistributor`
4. the errors `StrategyIndexMismatch` and `WithdrawalAmountTooLow` in `StakingNode`
5. the storage variable `pendingWithdrawnValidatorPrincipal` and the constant `GWEI_TO_WEI` in `StakingNode`
6. the storage variable `allocatedETHForDeposits` in `ynETH`
7. the storage variables `maxBatchDepositSize, stakeAmount` in `StakingNodesManager`
8. the constant `BASIS_POINTS_DENOMINATOR` in `ynBase`
9. the event `FeeReceiverSet` in the interface definition of `RewardsDistributorEvents` in `RewardsDistributor.sol`

10. the events `WithdrawalStarted` and `RewardsProcessed` in the interface definition of `StakingNodeEvents` in `StakingNode.sol`
11. the interfaces `IOracle` and `IEigenLayerBeaconOracle`

Version 2:

1. the error `ValueOutOfBounds` in ynETH

Code corrected:

All the listed issues have been resolved.

7.15 Complete Events

Informational Version 1 Code Corrected

CS-YNPROTO-014

We assume YieldNest checked when to emit events. Without clear specification on when events shall be emitted, we cannot verify if the events are emitted correctly. We encourage YieldNest to review if all relevant state changes emit events as intended (e.g., `RewardsDistributor.processRewards`, `ynBase._updatePauseWhitelist`).

The above also applies to indexing events. Most events index relevant fields but some don't. E.g., `RewardsDistributorEvents.FeeReceiverSet`.

Core corrected:

Events have been added throughout the codebase where important state changes are made.

7.16 Incorrect Natspec

Informational Version 1 Code Corrected

CS-YNPROTO-017

The natspec of `StakingNodesManager.validateDepositDataAllocation` claims the function does:

```
/**
* @notice Validates the allocation of deposit data across nodes to ensure the distribution does not increase the disparity in balances.
* @dev This function checks if the proposed allocation of deposits (represented by `_depositData`) across the nodes would lead to a more
* equitable distribution of validator stakes. It calculates the current and new average balances of nodes, and ensures that for each node,
* the absolute difference between its balance and the average balance does not increase as a result of the new deposits
* @param newValidators An array of `ValidatorData` structures representing the validator stakes to be allocated across the nodes.
*/
```

The implementation deviates from the description. The only check done is `nodeId >= nodes.length` for each node in `newValidators`.

Code corrected:

The function name and natspec have been changed to reflect the implementation.

7.17 Missing Natspec Param Definition

Informational | Version 1 | Code Corrected

CS-YNPROTO-018

The natspec definition for the second parameter `withdrawnValidatorPrincipal` of the function `StakingNode.claimDelayedWithdrawals` is missing.

Code corrected:

The function `StakingNode.claimDelayedWithdrawals` was removed from the codebase.

7.18 Overcomplicated Expression

Informational | Version 1 | Code Corrected

CS-YNPROTO-019

Expressions like `assetDecimals < 18 || assetDecimals > 18` in `ynLSD.convertToETH` can be replaced by simpler variants, they add unnecessary complexity and should be avoided.

Code corrected:

The expression has been simplified to `assetDecimals != 18`.

7.19 Remaining Todos

Informational | Version 1 | Code Corrected

CS-YNPROTO-020

In `StakingNodesManager.isStakingNodesAdmin` we found a left over to do comment `// TODO: define specific admin`.

Code corrected:

The todo was removed from the code.

8 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.

8.1 Gas Optimizations

Informational | Version 1 | Code Partially Corrected

CS-YNPROTO-015

We highlight gas inefficiencies when we see them but highly encourage YieldNest to check for more inefficiencies as we did find quite a lot and expect more to be present. The following list are examples we found:

1. In the function `LSDStakingNode.depositAssetsToEigenlayer`, `asset` can be used in place of `assets[i]`.
2. In the function `LSDStakingNode.depositAssetsToEigenlayer`, the check `address(strategy) == address(0)` is redundant with the one implemented in `ynLSD.retrieveAsset()`.
3. The substration in the function `ynETH.withdrawETH()` can be `unchecked`.
4. In the function `ynETH.depositETH()`, `msg.value` can be used in place of `assets`, as its gas cost is only 2.
5. In the functions `ynLSD.initializeLSDStakingNode()` and `StakingNodesManager.initializeStakingNode()`, a call is made to `node.getInitializedVersion()` but the returned value is never used.
6. When the functions `ynLSD.initializeLSDStakingNode()` and `StakingNodesManager.initializeStakingNode()` are used, `nodes.length` is read twice, passing the `nodeId` as a function argument can save an `SLOAD`.
7. In the function `StakingNodesManager.processWithdrawnETH()`, the call `ynETH.processWithdrawnETH()` can be done only if `withdrawnValidatorPrincipal > 0`, if partial withdrawals are expected to be more common than full withdrawals.
8. In the function `RewardsReceiver.initialize()`, the admin of the `WITHDRAWER` is explicitly set to be `DEFAULT_ADMIN`, but this is the case by default.
9. In `ynLSD.createLSDStakingNode` the state variable `nodes.length` is read multiple times including in `initializeLSDStakingNode` where it could be passed as argument.
10. `StakingNodesManager.validateDepositDataAllocation` the state variable `nodes.length` is read multiple times and could be cached.
11. When looping over `assets`, the `asset` array's length should be cached. When using the storage variable as bounded in `i < assets.length` it will be read multiple times.

Version 3

1. In `ynLSD.getTotalAssets()` the state variable `nodes.length` is read multiple times in the loop and could be cached to save `SLOAD`.

Code partially corrected:

1. Fixed.
2. Fixed.
3. No change.
4. The cached value in now used everywhere. It is now consistent within the function, but not gas-optimal.
5. Fixed. The value is used in the emitted event.
6. Fixed.
7. No change.
8. Fixed.
9. Fixed.
10. Fixed. Function name updated to `StakingNodesManager.validateNodes`.
11. Fixed.

Version 3

1. Fixed.

8.2 Incorrect Comments

Informational | Version 1 | Code Partially Corrected

CS-YNPROTO-016

1. In `ynLSD._convertToShares` the comment was copied from the same function that exists in `ynETH` and, hence, mentions `deltaynETH` instead of `deltaynLSD`.
2. In `ynLSD` the explanation of `retrieveAsset` is incorrect. It states `Retrieves a specified amount of an asset from the staking node.` But actually, transfers the asset to the staking node (as @dev correctly describes).

Code partially corrected:

1. The comment has been corrected.
2. No change

9 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.

9.1 Slightly Deviating Balance to Share in Case of Rebasing Tokens

Note **Version 1**

Rebasing tokens like stETH might transfer less than expected. This might get problematic when a contract expects to receive the amount specified in the transfer. E.g., in `ynLSD.deposit` the `safeTransferFrom` might transfer one or two wei less than specified in `amount`. But before, all calculations and the share distribution were done on the assumptions that `amount` would be later an asset of the contract. In consequence, this might break the invariant between the amount of shares and assets such that there are shares but no assets.

The issue was rated more severe in Incorrect balance transfers with rebasing tokens and fixed for `LSDStakingNode.depositAssetsToEigenlayer`. However, the behavior is still present in `ynLSD.deposit` but not considered severe enough to cause further issues.