

Produced for



by



# Contents

<b>1</b>	<b>Executive Summary</b>	<b>3</b>
<b>2</b>	<b>Assessment Overview</b>	<b>5</b>
<b>3</b>	<b>System Overview</b>	<b>7</b>
<b>4</b>	<b>Limitations and use of report</b>	<b>16</b>
<b>5</b>	<b>Terminology</b>	<b>17</b>
<b>6</b>	<b>Findings</b>	<b>18</b>
<b>7</b>	<b>Resolved Findings</b>	<b>19</b>
<b>8</b>	<b>Informational</b>	<b>26</b>
<b>9</b>	<b>Notes</b>	<b>28</b>

# 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 review was initially performed for both parts of the system. The native `ETH` staking and the liquid staking part. Both parts are not directly connected and updates were performed independently. Hence, our report will be split into two parts from here on. We extracted the relevant issues concerning either the `ETH` or `LSD` part. This report is issued for the `LSD` implementation. To still have a complete view of YieldNest's system, the system overview will still contain a summary of the `ETH` part.

The first audit performed on both parts found three `LSD` related high-severity issues (for a detailed description see the [Resolved Findings](#) section). The next iterations had little and less severe issues. All issues were always fixed and addressed accordingly.

Hence, we conclude that the code base has a high level of security. Still, 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.

-Severity Findings	0
-Severity Findings	3
•	3
-Severity Findings	2
•	2
-Severity Findings	8
•	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.

Starting with the fixes submission of                      onwards, two reports will exist that share a common history. One report will cover native ETH staking and this report will cover liquid staking (LSDs).

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
5	15 July 2024	48371285141dba4b93b78913abddd0087c51322e	Withdraw feature
6	22 July 2024	a1249b900307009a5cebc0925f27fb80496a1e9f	ynEigen code
7	06 Aug 2024	064c1b1aa8423b585d565ffbae9b7f8a9bfe82bf	ynEigen fixes

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

The following files are in the scope of the review in                      :

```
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 `external/etherfi/DepositRootGenerator.sol` was moved to `external/ethereum/DepositRootGenerator.sol`.

In `interfaces/IEigenLayerBeaconOracle.sol` and `interfaces/IOracle.sol` were removed from the codebase.

has two different features. The withdraw feature that includes (Note, the report was split. Please see the separated report for the withdraw feature.):

- `WithdrawalQueueManager.sol`
- `ynETHRedemptionAssetsVault.sol`
- `Constants.sol`
- `ynETH.sol`
- `ynBase.sol`
- `RewardsDistributor.sol`
- `RewardsReceiver.sol`
- `StakingNodesManager.sol` (taken out of scope while the review was on-going)
- `StakingNode.sol` (taken out of scope while the review was on-going)

The LSD staking feature called `ynEigen` was also reviewed in `ynEigen`. The scope included:

- `AssetRegistry.sol`
- `EigenStrategyManager.sol`
- `LSDRateProvider.sol`
- `TokenStakingNode.sol`
- `TokenStakingNodesManager.sol`
- `ynEigen.sol`
- `ynEigenDepositAdapter.sol`
- `ynBase.sol` (inherited by `ynEigen.sol`)

In this report ( `ynEigen` ), only the LSD part of `ynEigen` remained in scope:

- `AssetRegistry.sol`
- `EigenStrategyManager.sol`
- `LSDRateProvider.sol`
- `TokenStakingNode.sol`
- `TokenStakingNodesManager.sol`
- `ynEigen.sol`
- `ynEigenDepositAdapter.sol`

- `ynBase.sol` (inherited by `ynEigen.sol`)

### 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 ( ) 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\_ROLE

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 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 decides 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 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` (later renamed to `createTokenStakingNode`), `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 Protocol 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 withdrawal 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 `processWithdrawals`.



- 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 an 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.

## 3.6 Changes in V5

YieldNest has refactored LSD-related contracts, like `ynLSD` and `LSDStakingNodes`, and introduced `ynEigen`.

### 3.6.1 AssetRegistry

Trusted users holding `ASSET_MANAGER_ROLE` can add, disable, or remove liquid staking tokens (LSTs) supported by the system, as long as the registry is not paused. This contract also implements a function to track all deposited assets in `ynEigen` and their corresponding balances.

### 3.6.2 LSDRateProvider

This contract provides us with a `rate()` function which calculates for 1 ether of a given asset, how much underlying token is backing it. For example, how much `frxETH` backs 1 ether of `sfrxETH`.

### 3.6.3 ynEigenDepositAdapter

To avoid complexities of using rebasing tokens in `ynEigen` (users receive `ynETH` against their deposited tokens. As we will see `ynEigen` is not rebasing), `ynEigen` itself does not support rebasing tokens. Hence, rebasing tokens should first be wrapped and deposited into `ynEigen`. This wrapping mechanism is implemented by this adapter. Upon depositing `oETH` and `stETH` they first get wrapped in `woETH` and `wstETH` respectively and then deposited into `ynEigen`.

### 3.6.4 EigenStrategyManager

This contract orchestrates of the `EigenLayer` strategies. `STRATEGY_ADMIN_ROLE` can register strategies, but deregistering them is not implemented. `STRATEGY_CONTROLLER_ROLE` can call `stakeAssetsToNode()` to forwards the deposited assets from `ynEigen` to the staking node.

### 3.6.5 TokenStakingNodesManager

To deploy new instances of the staking nodes, this contract provides us with a function that can be called by users holding `TOKEN_STAKING_NODE_CREATOR_ROLE` to launch a new staking node. The Number of staking nodes that this contract can deploy is capped by `maxNodeCount`.

### 3.6.6 TokenStakingNode

Staking nodes stand as a router between YieldNest Protocol and `EigenLayer`. It allows the following functionalities:



1. Depositing the assets invested in `ynEigen` into their corresponding strategy through EigenLayer's strategy manager.
2. Delegating to a registered operator on the EigenLayer.
3. Undelegating from a delegatee.

### 3.6.7 *Withdrawing feature*

A withdrawal feature was added to the system. To withdraw, users need to call `requestWithdrawal()`. This will take the user's specified `ynETH` amount and issue an NFT with the resulting withdrawal terms. The NFT is not soul-bound but cannot be canceled. After all waiting periods are over and the withdrawal has been processed by all intermediate systems, the user (or the approved address) can claim their funds by calling `claimWithdrawal()`. This will burn the `ynETH` and the corresponding NFT. In case the redemption asset vault has sufficient funds, it will transfer the ETH to the specified receiver. The action might accrue some fees that will be sent to the fee receiver.

The requirement that needs to be met by YieldNest Protocol for a user to be able to claim their withdrawal request, is that `finalizeRequestsUpToIndex()` is called by the `REQUEST_FINALIZER_ROLE`. This simply raises the index of the last finalized request, and users with a lower request index can claim their funds.

Therefore, it is up to this role to ensure that the funds are present or have been properly processed in the background by the Ethereum network or by EigenLayer. For a full withdrawal, this might take quite some time as the validator needs to exit the network first. The funds will then be sent to the `eigenPod` where they need to transition EigenLayer's withdrawal process.

The most relevant changes for this feature are:

The addition of `ynETHRedemptionAssetsVault` that allows the `REDEEMER_ROLE`, from which the EigenLayer's funds can be claimed by the staking node with `completeQueuedWithdrawals()` for full withdrawals or `withdrawNonBeaconChainETHBalanceWei()` for partial withdrawals, to move redemption assets to a given address with `transferRedemptionAssets()` and send it to `ynETH` through `withdrawRedemptionAssets()`.

The addition of `WithdrawalQueueManager` allows users to request withdrawals via `requestWithdrawal()` and claim them with `claimWithdrawal`. Furthermore, a `REDEMPTION_ASSET_WITHDRAWER_ROLE` can withdraw surplus funds not belonging to any user with `withdrawSurplusRedemptionAssets()`. Besides, the contract offers privileged roles to manage the withdrawal queue with `finalizeRequestsUpToIndex()`.

From EigenLayer the funds can be claimed by the staking node with `completeQueuedWithdrawals()` for full withdrawals or `withdrawNonBeaconChainETHBalanceWei()` for partial withdrawals.

### 3.6.8 *Role updates*

After `YieldNest`, some new roles are introduced, some of the existing ones are renamed, and some have more capabilities. In what follows, we explain the roles in the `YieldNest` that are related to `ynEigen`:

#### DEFAULT\_ADMIN\_ROLE

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

A general administrative role with broad permissions, including potentially managing other roles.

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

#### STAKING\_ADMIN\_ROLE

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 `TokenStakingNode` in `TokenStakingNodesManager`. It's also able to set the `maxNodeCount`.



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

- `TokenStakingNodesManager.registerTokenStakingNode`
- `TokenStakingNodesManager.upgradeTokenStakingNode`
- `TokenStakingNodesManager.setMaxNodeCount`

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

## PAUSER\_ROLE

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

The role has the authority to pause 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:

- `AssetRegistry.pauseActions`
- `ynEigen.pauseDeposits`

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

## UNPAUSER\_ROLE

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

The role has the authority to unpause certain functions within the system, namely:

- `ynBase.unpauseTransfers`
- `ynBase.addToPauseWhitelist`
- `ynBase.removeFromPauseWhitelist`
- `AssetRegistry.unpauseActions`
- `ynEigen.unpauseDeposits`

## TOKEN\_STAKING\_NODE\_CREATOR\_ROLE

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

Authorized to create new LSD staking nodes within the system  
`ynLSD.createTokenStakingNode`.

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

## TOKEN\_STAKING\_NODES\_DELEGATOR\_ROLE

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

Users holding this role are privileged to call:

- `TokenStakingNode.delegate`
- `TokenStakingNode.undelegate`

## ASSET\_MANAGER\_ROLE

This role is entitled to add, disable, and remove assets by calling

- `AssetRegistry.addAsset`
- `AssetRegistry.disableAsset`
- `AssetRegistry.deleteAsset`

Therefore, assumed to be **fully trusted**.

## STRATEGY\_CONTROLLER\_ROLE



Assumed to be **fully trusted** to be entitled to this role. Users with this role can retrieve the deposited assets in `ynEigen` and transfer them to the `TokenStakingNode` to be later deposited in the `EigenLayer`.

With this role, users can call

- `EigenStrategyManager.stakeAssetsToNodes`
- and its scalar version `EigenStrategyManager.stakeAssetsToNode`

## STRATEGY\_ADMIN\_ROLE

Assumed to be **fully trusted**.

Users with this role have the authority to set a corresponding strategy for an asset by calling

- `EigenStrategyManager.setStrategy`



## 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			
Medium			
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:

- : Related to vulnerabilities that could be exploited by malicious actors
- : Architectural shortcomings and design inefficiencies
- : Mismatches between specification and implementation

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

-Severity Findings	0
-Severity Findings	0
-Severity Findings	0
-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.

-Severity Findings	0
-Severity Findings	3
<ul style="list-style-type: none"><li>• <a href="#">Computation of <code>ynLSD.getTotalAssets()</code> Is Wrong</a></li><li>• <a href="#">First Depositor Gets More Shares</a></li><li>• <a href="#">ynLSD Is Vulnerable to Donation Attack</a></li></ul>	
-Severity Findings	2
<ul style="list-style-type: none"><li>• <a href="#">_getStakedAssetBalanceForNode Counts Node Balance</a></li><li>• <a href="#">Incorrect Balance Transfers With Rebasing Tokens</a></li></ul>	
-Severity Findings	8
<ul style="list-style-type: none"><li>• <a href="#">Validity of Strategies Not Enforced During Initialization</a></li><li>• <a href="#">delegate() Does Not Include the Approver Salt</a></li><li>• <a href="#">Discrepancy in the Value Check for <code>maxAge</code></a></li><li>• <a href="#">Ignored Return Values</a></li><li>• <a href="#">Initializer Not Disabled</a></li><li>• <a href="#">Oracle Price Sanity Check</a></li><li>• <a href="#">Redundant Functionality</a></li><li>• <a href="#">Unused Code</a></li></ul>	
Informational Findings	3
<ul style="list-style-type: none"><li>• <a href="#">Incorrect Comments</a></li><li>• <a href="#">Overcomplicated Expression</a></li><li>• <a href="#">Complete Events</a></li></ul>	

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

CS-YNPROTO-017

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

CS-YNPROTO-014

In `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 ynLSD Is Vulnerable to Donation Attack

CS-YNPROTO-011

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:  
 $1\ T = 1\ \text{ETH}$ .
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.

## 7.4 `_getStakedAssetBalanceForNode` Counts Node Balance

CS-YNPROTO-018

`EigenStrategyManager._getStakedAssetBalanceForNode()` fetches the amount of asset held by the staking node, which should be zero. However, an attacker can donate his assets to avoid removal of an asset through `AssetRegistry.deleteAsset()`.

---

#### Code corrected:

`EigenStrategyManager._getStakedAssetBalanceForNode()` only calculates the balance staked in the strategy.

## 7.5 Incorrect Balance Transfers With Rebasing Tokens

CS-YNPROTO-015

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 Validity of Strategies Not Enforced During Initialization

CS-YNPROTO-004

`AssetRegistry.addAsset()` fetches the strategy corresponding to an asset from the strategy manager and ensures that there exists one. Then, this asset gets added to the list of the registered assets. However, when getting initialized, `initialize()` does not enforce the corresponding strategies being registered in the strategy manager.

---

### Code corrected:

When initializing the asset registry, it gets enforced that the strategies of the assets are registered in the strategy manager as well.

## 7.7 `delegate()` Does Not Include the Approver Salt

CS-YNPROTO-012

`TokenStakingNode.delegate()` emits `Delegated` event with the `approverSalt` set to 0.

---

### Code corrected:

The `Delegated` event now includes the `approverSalt`.

## 7.8 Discrepancy in the Value Check for `maxAge`

CS-YNPROTO-010

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

CS-YNPROTO-016



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`
  - `ynEigenDepositAdapter` does not check the return value by `stETH.transferFrom`
  - `ynEigenDepositAdapter` does not check the return value by `stETH.approve`
  - `ynEigenDepositAdapter` does not check the return value by `oETH.transferFrom`
  - `ynEigenDepositAdapter` does not check the return value by `oETH.approve`
- 

#### Code corrected:

The OpenZeppelin library `SafeERC20` is used in the `ERC20` interactions mentioned above.

## 7.10 Initializer Not Disabled

CS-YNPROTO-007

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

CS-YNPROTO-006

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

CS-YNPROTO-008



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

---

#### Code corrected:

The function `isAddressWhitelisted` was removed.

## 7.13 Unused Code

CS-YNPROTO-009

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 constant `BASIS_POINTS_DENOMINATOR` in `ynBase`
2. the event `FeeReceiverSet` in the interface definition of `RewardsDistributorEvents` in `RewardsDistributor.sol`
3. the events `WithdrawalStarted` and `RewardsProcessed` in the interface definition of `StakingNodeEvents` in `StakingNode.sol`
4. the interfaces `IOracle` and `IEigenLayerBeaconOracle`

Version5:

The following imports are unused:

1. `AssetRegistry` imports `AccessControlUpgradeable` twice.

The following variables are unused

1. In `AssetRegistry` `IAAssetRegistryEvents` the event `AssetActivated(address indexed asset)` and the errors `UnsupportedAsset` and `ZeroAmount`.
2. In `TokenStakingNodesManager` the events `AssetRetrieved`, `DepositsPausedUpdated` from `ITokenStakingNodesManagerEvents` and the errors `UnsupportedAsset`, `Unauthorized`, `InsufficientFunds`, `Paused`, and `ZeroAmount`. Additionally, all declared events are unused.
3. In `ynEigen` `IynEigenEvents` the events `LSDStakingNodeCreated` and `MaxNodeCountUpdated`, as well as the error `LengthMismatch`.
4. `WithdrawalQueueManager.MAX_SECONDS_TO_FINALIZATION` is never used. Removing it can improve the code's readability

---

#### Code partially corrected:

Version 5 introduced new code parts that are not used. All others were corrected.

## 7.14 Complete Events

CS-YNPROTO-013





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.

## 7.15 Incorrect Comments

CS-YNPROTO-003

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 corrected:

1. The comment has been corrected.
2. The comment has been corrected.

## 7.16 Overcomplicated Expression

CS-YNPROTO-005

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`.

# 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

CS-YNPROTO-002

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. In the functions `ynLSD.initializeLSDStakingNode()` and `StakingNodesManager.initializeStakingNode()`, a call is made to `node.getInitializedVersion()` but the returned value is never used.
  4. 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.
  5. In `ynLSD.createLSDStakingNode` the state variable `nodes.length` is read multiple times including in `initializeLSDStakingNode` where it could be passed as argument.
  6. 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.
  7. In `ynLSD.getTotalAssets()` the state variable `nodes.length` is read multiple times in the loop and could be cached to save SLOAD.
- 
1. In one transaction when `EigenStrategyManager.stakeAssetsToNode()` is called, funds are transferred from `ynEigen` to the `EigenStrategyManager` via `retrieveAssets()`. From the `EigenStrategyManager` they are forwarded to the node. The node ultimately stakes these funds into the `EigenLayer` contract. There might be a reason we couldn't see but this detour seems quite inefficient.
  2. In `AssetRegistry`, `totalAssets()`, `assetIsSupported()`, and `assetData()` is never called internally and can be defined as external.
  3. `AssetRegistry._assetData` is defined a public. Therefore, a getter will automatically be generated by the compiler. Despite this, a getter function `assetData()` is implemented.
  4. In `AssetRegistry`, `assetIsDisabled()`, `assetIsActive()`, and `assetIsUnavailable()` are never called and hence dead code.
  5. `EigenStrategyManager.getStakedAssetBalanceForNode()` can be defined as external.
  6. `ynEigen._deposit()` is called always with `receiver` being set as `msg.sender`.

7. `ynEigen._convertToShares()` is only called with `Math.Rounding.Floor`.
  8. `ynEigen.previewDeposit()` can be defined as external.
- 

**Code partially corrected:**

1. Fixed.
2. Fixed.
3. Fixed. The value is used in the emitted event.
4. Fixed.
5. Fixed.
6. Fixed.
  
7. Fixed.
  
4. Fixed.

## 8.2 `TOKEN_STAKING_NODE_OPERATOR_ROLE` Is Ultimately Not Used

CS-YNPROTO-001

The role `TOKEN_STAKING_NODE_OPERATOR_ROLE` is defined in `TokenStakingNodesManager` and used in the modifier `TokenStakingNode.onlyOperator`. However, this modifier is not used in the code.

## 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

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.

### 9.2 ynEigen Shares ERC 4626 Features

Even though `ynEigen` shares some features and interfaces with the ERC 4626 vault standard, the user needs to be aware that the implemented contracts are not ERC 4626 compatible vaults.