Security Review Report
NM-0445 Renzo

NETHERMIND
SECURITY

(April 14, 2025)

Contents

1 Executive Summary

This document presents the security review conducted by Nethermind Security for the core smart contracts of the Renzo Protocol. Renzo is a restaking solution that abstracts and manages AVS strategies for Liquid Restaking Tokens (LRTs), simplifying Ethereum restaking and making it accessible to a broader audience. By leveraging EigenLayer, Renzo enables users to restake their ETH without the complexities of node operation, AVS strategy selection, or direct smart contract interactions.

The security review covered all core smart contracts, with a primary focus on Renzo's integration with EigenLayer in preparation for the upcoming EigenLayer slashing update.

The audited code comprises 2774 lines of code written in the Solidity language. The security review included all of the core Renzo protocol smart contracts.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, and (c) creation of test cases. **Along this document, we report** nine points of attention, where two are classified as `High`, one is classified as `Medium`, three are classified as `Low`, and three are classified as `Informational` or `Best Practice` severity. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the test suite evaluation and automated tests. Section 9 concludes the document.

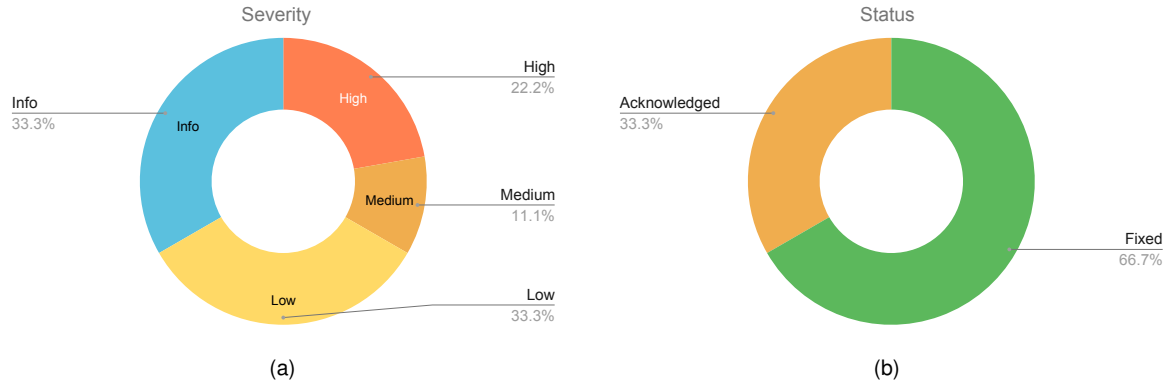

Fig. 1: Distribution of issues: Critical (0), **High** (2), **Medium** (1), **Low** (3), **Undetermined** (0), **Informational** (3), **Best Practices** (0). **Distribution of status: Fixed** (6), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	March 28, 2025
Final Report	April 14, 2025
Repository	Contracts
Start Commit	eaceb5f812c7ffb341c6e83b4bf28bc494380fa4
Final Commit	f4d9610da5c88f3f328f249002c04a4c01dd9578
Documentation	Docs
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/RestakeManagerStorage.sol	36	16	44.4%	15	67
2	contracts/IRestakeManager.sol	23	1	4.3%	7	31
3	contracts/RestakeManager.sol	445	174	39.1%	116	735
4	contracts/token/IEzEthToken.sol	6	1	16.7%	3	10
5	contracts/token/EzEthTokenStorage.sol	6	16	266.7%	3	25
6	contracts/token/EzEthToken.sol	37	17	45.9%	10	64
7	contracts/Oracle/RenzoOracle.sol	134	54	40.3%	39	227
8	contracts/Oracle/RenzoOracleStorage.sol	11	3	27.3%	4	18
9	contracts/Oracle/IRenzoOracle.sol	32	1	3.1%	5	38
10	contracts/Oracle/Lido/StETHShim.sol	43	16	37.2%	8	67
11	contracts/Oracle/Mantle/METHShim.sol	52	19	36.5%	11	82
12	contracts/Oracle/Mantle/METHShimStorage.sol	6	1	16.7%	2	9
13	contracts/Oracle/Mantle/IMethStaking.sol	5	1	20.0%	2	8
14	contracts/Oracle/Binance/WBETHShim.sol	52	19	36.5%	11	82
15	contracts/Oracle/Binance/WBETHShimStorage.sol	6	1	16.7%	2	9
16	contracts/Oracle/Binance/IStakedTokenV2.sol	4	2	50.0%	1	7
17	contracts/RateProvider/IRateProvider.sol	4	11	275.0%	4	19
18	contracts/RateProvider/BalancerRateProviderStorage.sol	7	3	42.9%	3	13
19	contracts/RateProvider/BalancerRateProvider.sol	25	9	36.0%	8	42
20	contracts/Permissions/IRoleManager.sol	22	31	140.9%	15	68
21	contracts/Permissions/RoleManagerStorage.sol	27	22	81.5%	14	63
22	contracts/Permissions/RoleManager.sol	64	39	60.9%	20	123
23	contracts/Delegation/OperatorDelegator.sol	568	234	41.2%	134	936
24	contracts/Delegation/OperatorDelegatorStorage.sol	57	36	63.2%	26	119
25	contracts/Delegation/IOperatorDelegator.sol	18	9	50.0%	9	36
26	contracts/Delegation/utils/OperatorDelegatorAdminLib.sol	185	38	20.5%	46	269
27	contracts/Delegation/utils/WETHUnwrapper.sol	22	6	27.3%	8	36
28	contracts/Deposits/DepositQueueStorage.sol	15	7	46.7%	7	29
29	contracts/Deposits/IDepositQueue.sol	9	1	11.1%	2	12
30	contracts/Deposits/DepositQueue.sol	174	71	40.8%	52	297
31	contracts/Errors/Errors.sol	71	71	100.0%	70	212
32	contracts/Withdraw/IWithdrawQueue.sol	9	13	144.4%	6	28
33	contracts/Withdraw/WithdrawQueue.sol	548	177	32.3%	116	841
34	contracts/Withdraw/WithdrawQueueStorage.sol	51	20	39.2%	22	93
	Total	**2774**	**1140**	**41.1%**	**801**	**4715**

3 Summary of Issues

	Finding	Severity	Update
1	Incorrect `queuedShares` scaling might lead to DoS and incorrect TVL calculation	High	Fixed
2	The slashed Beacon Chain ETH strategy shares are accounted for twice in TVL calculations	High	Fixed
3	The `emergencyTrackSlashedQueuedWithdrawalDelta(...)` function will revert for `beaconChainETHStrategy` due to missing `underlyingToken(...)` function	Medium	Fixed
4	Checkpoint system can be DoSed when `currentCheckpointTimestamp` loses synchronization	Low	Acknowledged
5	The `emergencyTrackSlashedQueuedWithdrawalDelta(...)` function can be used to artificially decrease the TVL to get better exchange rate on `ezETH`	Low	Fixed
6	Users might get a worse exchange rate on `ezETH` due to the TVL spike after queueing the withdrawal	Low	Fixed
7	The `emergencyTrackQueuedWithdrawals(...)` function doesn't verify the queued withdrawal ownership, potentially leading to inflated TVL	Info	Fixed
8	The `fillERC20withdrawBuffer(...)` function doesn't refund excess tokens sent	Info	Acknowledged
9	The `totalBeaconChainExitBalance` can be inflated if no checkpoint was finalized after the validator got slashed	Info	Acknowledged

4 System Overview

Renzo Protocol is a liquid restaking and collateral management solution built as an overlay to EigenLayer. It enables users to pool their staked tokens, deposit ERC20 collateral, and earn rewards while retaining proportional ownership through the issuance of ezETH. The protocol's architecture is modular and comprises several contracts that facilitate depositing, staking, withdrawing, reward forwarding, fee collection, and governance.

4.1 Actors

- **Users**: Users are the primary participants who interact with the protocol by depositing collateral tokens such as ERC20 tokens or native ETH to receive **ezETH** tokens representing their share in the pooled staking funds.
- **Admins**: Admins are entrusted with the governance and operational oversight of the protocol. Beyond setting key parameters, admins manage the staking lifecycle by deciding when to stake, unstake, or withdraw funds from **OperatorDelegator** contracts. Using a role-based access control system enforced by the **RoleManager**, they monitor system metrics such as the withdrawal buffer, deposit queue status, and validator performance to ensure that the system operates as intended.
- **OperatorDelegator contracts**: Stakers (in EigenLayer's terminology); these contracts aggregate user funds and execute staking operations on behalf of Renzo. These contracts manage deposits and commission staking activities by delegating these funds to EigenLayer Operators.
- **Operators**: These are independent nodes within the EigenLayer network. In Renzo, **OperatorDelegator** contracts delegate their pooled stake to these **Operators**. The performance of the Operators directly influences the protocol economics.
- **AVS (Actively Validated Services)**: **AVS** refers to the external services that rely on the deposited collateral as a security guarantee. In return for economic security, these services provide additional rewards that enhance the overall yield of the protocol.

4.2 Deposits

Collateral Deposits and ezETH Minting

Users can deposit either **native Ether** or whitelisted **ERC20** collateral tokens such as stETH or rETH. The main entry point for users is the **RestakeManager** contract.

Upon depositing ERC20 tokens, the **RestakeManager** contract will pick an **OperatorDelegator** contract to transfer the funds to before depositing these tokens into **EigenLayer**.

In the case of native Ether deposits, the funds will be sent to the **DepositQueue** contract, and they will be stored there until used for a validator deposit.

The **RenzoOracle** contract provides on-chain token valuations used in the calculation of the **TVL** (Total Value Locked). Based on the current TVL and the existing supply of **ezETH**, new tokens are minted and allocated to the depositor

4.3 Withdrawals

Withdrawals are executed through a two-step process. If users want to withdraw, they will interact with the **WithdrawQueue** contract. When a user queues a withdrawal, their **ezETH** tokens are moved to the **WithdrawQueue** contract. After a delay period, users can claim their withdrawals. The protocol will burn the corresponding amount of **ezETH** tokens from the **WithdrawQueue** contract upon completing the withdrawal.

In addition, the system incorporates a withdrawal buffer mechanism. The buffer holds a reserve of tokens for each supported collateral asset. If a user's withdrawal request does not exceed the available balance in the buffer, the funds are available at hand, meaning that the protocol does not need to withdraw tokens from EigenLayer. This design enhances liquidity and maximizes staking rewards for Users.

4.4 Admin responsibilities and protocol operation

Admins ensure the smooth operation of Renzo Protocol. They monitor key system activities and invoke permissioned functions to maintain system integrity. Key operational tasks include:

- **Handle withdrawals**: Admins monitor the withdrawal buffers, ensuring that sufficient funds are available in the buffer to facilitate user withdrawals.
- **Monitoring for slashing**: Given that Validators in EigenLayer secure the staked funds, any misbehavior may lead to slashing events. Admins continuously monitor validator performance to detect potential slashing or other behaviors that lower the operator's balance.
- **Staking from the DepositQueue**: The protocol aggregates native ETH deposits within the DepositQueue. Admins oversee this queue to ensure that once the deposit threshold is reached, the ETH is staked through the OperatorDelegator's staking functions.

By performing these tasks, admins help ensure that there is always sufficient liquidity in the protocol and that the system remains operational and secure under varying conditions.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [High] Incorrect `queuedShares` scaling might lead to DoS and incorrect TVL calculation

File(s): contracts/Delegation/OperatorDelegator.sol

Description: To correctly track the staker's share accounting in case of a slash, EigenLayer distinguishes between two share types: deposit and withdrawable shares. Deposit shares track the staker's original deposit in the protocol. In contrast, withdrawable shares specify what can be withdrawn from the protocol, factoring in the slashing that could have happened post-deposit. The conversion between share types is performed with the help of the deposit scaling factor (DSF) and slashing factor. Provided deposit shares D, the following equation can be used to calculate withdrawable shares W.

$$W = D * \frac{DSF}{1e18} * \frac{slashing\ factor}{1e18}$$

In EigenLayer's contracts, whenever the staker's delegation is increased, the deposit scaling factor for that staker is updated as well. Examples of such operations include: depositing tokens into a strategy, completing the queued withdrawal in the form of shares or delegating shares to an operator.

```
function _increaseDelegation(...) internal {
    // ...
    DepositScalingFactor storage dsf =
        _depositScalingFactor[staker][strategy];
    dsf.update(prevDepositShares, addedShares, slashingFactor);
    // ...
}
```

The slashing factor is a number that starts at the value of `1e18` and can only decrease down to the lower boundary value of `0` in case of a full slash. The DSF also starts at `1e18` for a fresh deposit but can exceed `1e18` in certain scenarios. A notable example includes the first deposit / delegation to an operator that was previously slashed. Since stakers shouldn't be affected by the slashes that happened prior to their initial delegation, their DSF will be incremented to offset the current slashing factor for that operator.

```
function update(
    DepositScalingFactor storage dsf,
    uint256 prevDepositShares,
    uint256 addedShares,
    uint256 slashingFactor
) internal {
    // @audit If this is the staker's first deposit or they are delegating to
    // an operator, the prevDepositShares will be zero.
    if (prevDepositShares == 0) {
        // @audit Stakers shouldn't be punished for any slashing that happened
        // prior to their first deposit. The inverse scaling factor is calculated
        // to offset prior slashing.
        dsf._scalingFactor = dsf.scalingFactor().divWad(slashingFactor);
        return;
    }
    // ...
}
```

For example if the current slashing factor for an operator is `0.9`, the staker that deposited `100` shares would be able to withdraw only `90` shares right after the deposit. To mitigate the effect of previous slashing, the inversed DSF is calculated for the staker: `1 * (1 / 0.9) = 1.11`. Using the formula, withdrawable shares would be computed as: `100 * 1.11 * 0.9 = 100`, which matches the initial deposit of `100` shares.

The problem in Renzo's `OperatorDelegator` contract is that in the withdrawal flow, the deposit share amounts are used interchangeably with deposit amounts scaled by the DSF, which is incorrect. Whenever an admin queues the withdrawal on the `OperatorDelegator` contract, the `queueWithdrawal(...)` function from the `OperatorDelegatorAdminLib` is executed to prepare the withdrawal params with **deposit** share amounts to withdraw. These deposit amounts are added to the `queuedShares` mapping for each token that is being withdrawn. The `queuedShares` mapping is used in the TVL calculation to track the 'in-flight' funds.

```
function queueWithdrawal(
    IERC20[] calldata tokens,
    uint256[] calldata tokenAmounts,
    // ...
) external returns (/* ... */) {
    // ...
    for (uint256 i; i < tokens.length;) {
        if (address(tokens[i]) == IS_NATIVE) {
            // ...
            // @audit For BeaconChainETH strategy token amount is 1:1 with deposit shares.
            queuedWithdrawalParams[0].depositShares[i] = tokenAmounts[i];
        } else {
            // ...
            // @audit For ERC20 strategies get the exchange ratio from the strategy.
            queuedWithdrawalParams[0].depositShares[i] =
            tokenStrategyMapping[tokens[i]].underlyingToSharesView(
                tokenAmounts[i]
            );
        }
        // ...
        queuedShares[address(tokens[i])] +=
            queuedWithdrawalParams[0].depositShares[i];
        // ...
    }
    // ...
    // @audit Create the withdrawal struct, queue it, and return its hash.
    bytes32 withdrawalRoot =
        delegationManager.queueWithdrawals(queuedWithdrawalParams)[0];
    // ...
}
```

The `queueWithdrawals(...)` function from the `DelegationManager` contract will forward the `queuedWithdrawalParams` to the `_removeSharesAndQueueWithdrawal(...)` function to construct the `Withdrawal` struct, queue it and return its hash as the `withdrawalRoot`. The thing to note is that the `_removeSharesAndQueueWithdrawal(...)` function will first scale the provided deposit share amounts by multiplying them with the staker's current DSF and then storing them in the `Withdrawal` struct.

```
function _removeSharesAndQueueWithdrawal(
    address staker,
    address operator,
    IStrategy[] memory strategies,
    uint256[] memory depositSharesToWithdraw, // @audit The provided deposit share amount.
    // ...
) internal returns (bytes32) {
    // ...
    for (uint256 i = 0; i < strategies.length; ++i) {
        // ...
        DepositScalingFactor memory dsf =
            _depositScalingFactor[staker][strategies[i]];
        // ...
        // @audit The shares queued for withdrawal are scaled by DSF. This value can be bigger than the provided deposit
        ↪  shares amount.
        scaledShares[i] =
            dsf.scaleForQueueWithdrawal(depositSharesToWithdraw[i]);
        // ...
    } // ...
    Withdrawal memory withdrawal = Withdrawal({
        staker: staker,
        delegatedTo: operator,
        withdrawer: staker,
        nonce: nonce,
        startBlock: uint32(block.number),
        strategies: strategies,
        // @audit-issue Withdrawal struct holds SCALED deposit shares,
        // while queuedShares mapping holds regular deposit shares.
        scaledShares: scaledShares
    });
    bytes32 withdrawalRoot = calculateWithdrawalRoot(withdrawal);
    // ...
    return withdrawalRoot;
}
```

The fact that the `Withdrawal` struct holds scaled deposit shares becomes problematic once the `completeQueuedWithdrawal(...)` function is called to finalize the withdrawal. The function will first call the `DelegationManager` to complete the withdrawal and withdraw the shares as tokens. Since the shares are no longer queued, the `_reduceQueuedShares(...)` will be called to decrement the `queuedShares` mapping initially incremented in the `queueWithdrawal(...)` function.

```
function completeQueuedWithdrawal(
    IDelegationManager.Withdrawal calldata withdrawal,
    IERC20[] calldata tokens
) external nonReentrant onlyNativeEthRestakeAdmin {
    // ...
    // @audit Withdraw shares as tokens.
    try delegationManager.completeQueuedWithdrawal(
        withdrawal, tokens, true
    ) {} catch {
        delegationManager.completeQueuedWithdrawal(
            withdrawal, tokens, 0, true
        );
    }
    // @audit Since the shares are no longer queued, decrease
    // the queuedShares mapping.
    _reduceQueuedShares(withdrawal, tokens);
    // ...
}
```

If there was no slash, the `queuedShares` mapping should be decremented by the amount it was incremented with. This, however, might not be the case since scaled shares can be larger than the deposit amount if the staker had the DSF greater than `1e18` at the time of queuing the withdrawal.

```
function _reduceQueuedShares(
    IDelegationManager.Withdrawal calldata withdrawal,
    IERC20[] memory tokens
) internal {
    for (uint256 i; i < tokens.length;) {
        // ...
        // @audit-issue If 100 deposit shares were queued for withdrawal, the queuedShares
        // mapping holds the value of 100. The scaled shares, however can be >100 if the staker's
        // DSF is greater than 1e18. The following line will either revert leading to DoS
        // or will lead to incorrect TVL calculation if too many shares are subtracted.
        queuedShares[address(tokens[i])] -= withdrawal.scaledShares[i];
        // ...
    }
}
```

If there are no more withdrawals queued, the subtraction of too many shares will cause the `_reduceQueuedShares(...)` function to revert, leading to a DoS of the withdrawal completion flow until the `queuedShares` mapping is filled again with new withdrawals to make the subtraction succeed.

Even if there are enough shares in the `queuedShares` mapping, subtracting the scaled shares will influence the TVL calculation, making the TVL lower than it should be.

The problem is also present in the `emergencyTrackQueuedWithdrawal(...)` function.

Recommendation(s): The main purpose of the `queuedShares` mapping is to track the 'in-flight' tokens in the TVL calculation. Once the withdrawal is completed via the `completeQueuedWithdrawal(...)` function, the value in the mapping should be decremented accordingly. Consider subtracting the unscaled deposit amount of shares from the `queuedShares`.

Status: Fixed

Update from the client: Updated the code to properly reduce `initialWithdrawableShares` (tracked in `queuedWithdrawalTokenInfo` mapping) from the queuedShares while completing the withdrawal through `completeQueuedWithdrawals`. Code Ref.

6.2 [High] The slashed Beacon Chain ETH strategy shares are accounted for twice in TVL calculations

File(s): contracts/Delegation/OperatorDelegator.sol

Description: Whenever an AVS slashes an operator, the `DelegationManager` contract will call the `increaseBurnableShares(...)` function on an appropriate share manager contract. The `StrategyManager` contract will be called for ERC20 strategies and `EigenPodManager` contract for Beacon Chain ETH strategy. Both contracts track the amount of shares to burn. The `StrategyManager` contract allows anyone to call the `burnShares(...)` function to transfer the slashed ERC20 tokens to the burn address. This however, is not the case for the `EigenPodManager` contract. The slashed native tokens stay in the `EigenPod` contract. The `burnableETHShares` state variable is not utilized by EigenLayer contracts in their current version. It might change post Pectra when beacon chain withdrawals are allowed to be triggered from the execution layer.

```
function increaseBurnableShares(
    IStrategy,
    uint256 addedSharesToBurn
) external onlyDelegationManager nonReentrant {
    // @audit The burnableETHShares variable is not utilized by the
    // current version of EigenLayer contracts, except for the 'write'
    // and event emission below.
    burnableETHShares += addedSharesToBurn;
    emit BurnableETHSharesIncreased(addedSharesToBurn);
}
```

The fact that native tokens are not transferred out of the contract when burned, makes it so that the tokens slashed by an AVS, are included in the `podBalanceGwei` when starting the checkpoint.

```
function _startCheckpoint(...) internal {
    // ...
    // @audit The slashed native tokens are not "burned", they are locked
    // in the EigenPod contract. The address(this).balance includes these funds.
    uint64 podBalanceGwei = uint64(address(this).balance / GWEI_TO_WEI)
        - restakedExecutionLayerGwei;
    // ...
    Checkpoint memory checkpoint = Checkpoint({
        // ...
        podBalanceGwei: podBalanceGwei,
        // ...
    });
    // ...
}
```

Subsequently the slashed native tokens are included in the `restakedExecutionLayerGwei` when the checkpoint is finalized.

```
// @audit This function is called when finalizing the checkpoint.
function _updateCheckpoint(
    Checkpoint memory checkpoint
) internal {
    // ...
    // @audit Increment restakedExecutionLayerGwei by pod balance gwei that was
    // set in the _startCheckpoint(...) function.
    restakedExecutionLayerGwei += checkpoint.podBalanceGwei;
    // ...
}
```

In Renzo's TVL calculation the `restakedExecutionLayerGwei` is used to exclude any unclaimed rewards sitting in the `EigenPod` contract from the TVL calculation until they are claimed and processed by the `OperatorDelegator` contract. The value is queried by calling the `withdrawableRestakedExecutionLayerGwei(...)` function from the `EigenPod` contract.

```
function withdrawableRestakedExecutionLayerGwei() external view returns (uint64)
{
    // @audit Renzo's TVL calculation utilizes this function. It does not account
    // for the `burnableETHShares` locked in the EigenPod.
    return restakedExecutionLayerGwei;
}
```

The problem is that the `_getPartialWithdrawalsPodDelta(...)` function that calls the `withdrawableRestakedExecutionLayerGwei(...)` does not accommodate for the fact that the value returned from the call includes the AVS slashed tokens that are locked in the EigenPod contract.

```
function _getPartialWithdrawalsPodDelta() internal view
    returns (uint256 podDelta)
{
    podDelta = (
        // @audit Since the restaked execution layer gwei can be inflated by locked
        // native Ether, the condition will more likely evaluate to TRUE.
        uint256(eigenPod.withdrawableRestakedExecutionLayerGwei())
            * GWEI_TO_WEI > totalBeaconChainExitBalance
    )
        // @audit If condition is TRUE, the returned pod delta is non-zero.
        ? uint256(eigenPod.withdrawableRestakedExecutionLayerGwei())
            * GWEI_TO_WEI - totalBeaconChainExitBalance
        : 0;
}
```

As a result the non-zero pod delta will be returned even in situations when there are no rewards and no full withdrawal proceeds in the `EigenPod` contract. The pod delta is subtracted from the withdrawable shares of a particular operator. It is incorrect since withdrawable shares were already adjusted by any slashing imposed by an AVS.

```
// @audit This function is called for every operator delegator during TVL calculation.
function getStakedETHBalance() external view returns (uint256) {
    // ...
    // @audit The withdrawable shares are already adjusted by slashing.
    try delegationManager.getWithdrawableShares(address(this), strategies)
    returns (uint256[] memory withdrawableShares, uint256[] memory) {
        uint256 collateralBalance = withdrawableShares[0];
        // ...
        // @audit-issue The shares are lowered by the pod delta which includes slashing.
        // Slashing is included twice now. TVL will be lower OR subtraction will underflow
        // and revert the TVL calculation.
        collateralBalance -= _getPartialWithdrawalsPodDelta();
        return collateralBalance;
    } catch {
        // ...
    }
}
```

Over time the TVL discrepancy will only increase if the validator incurs slashes on the Beacon Chain ETH strategy. If the withdrawable shares are sufficient to cover the inflated pod delta, the TVL will be lower than it should be. However, if there are not enough withdrawable shares, the TVL calculation will revert due to underflow in the `getStakedETHBalance(...)` function causing a DoS in core flows such as deposits and withdrawals.

Recommendation(s): Since the `EigenPod` contract cannot directly differentiate between rewards and slashed tokens, and AVS slashing is rare, consider adding an admin-controlled mechanism to adjust the TVL if a slash occurs in the Beacon Chain ETH strategy. The adjustment value can be derived from the `OperatorSharesSlashed` events emitted by the `slashOperatorShares(...)` function from the `DelegationManager` contract whenever an Operator utilized by Renzo is slashed.

This solution could serve as a temporary measure until EigenLayer introduces native token burns.

Status: Fixed

Update from the client: Added an admin function `emergencyTrackAVSEthSlashedAmount` to track `beaconChainEthAvsSlashingAmount` in `OperatorDelegator` along with updated calculation in `_getPartialWithdrawalsPodDelta`. Code Ref.

6.3 [Medium] The `emergencyTrackSlashedQueuedWithdrawalDelta(...)` function will revert for `beaconChainETHStrategy` due to missing `underlyingToken(...)` function

File(s): contracts/Delegation/utils/OperatorDelegatorAdminLib.sol

Description: With the upcoming update to EigenLayer, shares in the withdrawal queue remain slashable for `MIN_WITHDRAWAL_DELAY_BLOCKS` (approximately 17.5 days in blocks). During this delay period, any slashing will decrease the amount of withdrawable shares in EigenLayer. Slashing of the shares in the withdrawal queue takes effect immediately in EigenLayer and is reflected in the withdrawable share balance.

On Renzo's side, the `emergencyTrackSlashedQueuedWithdrawalDelta(...)` is used to track the potential slashes to the shares that are currently in the EigenLayer withdrawal queue.

```
// @audit Permisionless function to track queued shares that were slashed.
function emergencyTrackSlashedQueuedWithdrawalDelta(
    bytes32[] calldata withdrawalRoots
) external {
    OperatorDelegatorAdminLib.trackSlashedQueuedWithdrawalDelta(
        withdrawalRoots,
        queuedWithdrawal,
        queuedWithdrawalTokensSlashedDelta,
        totalTokenQueuedSharesSlashedDelta,
        delegationManager
    );
}
```

This function is permissionless and can be called by anyone to calculate the slashing delta between the deposit and withdrawable shares. Any difference between these share types is considered a slashing delta and will be deducted from the TVL.

To calculate the slashing delta, the `trackSlashedQueuedWithdrawalDelta` function in the `OperatorDelegatorAdminLib` library will iterate through the withdrawal roots, retrieving the queued withdrawal struct from EigenLayer:

```
function trackSlashedQueuedWithdrawalDelta(
    bytes32[] calldata withdrawalRoots,
    // ...
) external {
    for (uint256 i = 0; i < withdrawalRoots.length;) {
        // ...
        for (uint256 j = 0; j < withdrawal.scaledShares.length;) {
            // @audit-issue The BeaconChainETH strategy does not implement the
            // underlyingToken(...) function. The call will revert.
            address underlyingToken =
                address(withdrawal.strategies[j].underlyingToken());
           // ...
        }
        // ...
    }
}
```

The issue is that this function assumes each EigenLayer strategy implements the `underlyingToken()` function. However, it doesn't account for the fact that `beaconChainETHStrategy` doesn't implement this function. In EigenLayer, the `beaconChainETHStrategy` is a special virtual strategy that points to an empty address (`0xbeaC0eeEeeeeEEeEeEEEEeeEEeEeeeEeeEEBEaC0`) as defined in the EigenPodManagerStorage contract.

When the `emergencyTrackSlashedQueuedWithdrawalDelta` function is called for a withdrawal that includes the `beaconChainETHStrategy`, the transaction will revert when trying to call `underlyingToken(...)` on the virtual strategy address, making it impossible to track slashing for withdrawals that include `beaconChainETHStrategy`.

Recommendation(s): Consider modifying the logic inside the `trackSlashedQueuedWithdrawalDelta` function to handle the special case of `beaconChainETHStrategy`.

Status: Fixed

Update from the client: Implemented `_getUnderlyingFromStrategy` . Fixed in el-slashing branch.

6.4 [Low] Checkpoint system can be DoSed when `currentCheckpointTimestamp` loses synchronization

File(s): `contracts/Delegation/OperatorDelegator.sol`

Description: EigenLayer's checkpoint system allows `EigenPod` owners to prove the current balances of their validators on the Beacon Chain and receive corresponding EigenLayer shares or tokens from withdrawal and reward proceeds. The process starts with a call to `startCheckpoint(...)` and ends when all checkpoint proofs are verified by calling `verifyCheckpointProofs(...)`. EigenPod also includes an emergency checkpointing mechanism via `verifyStaleBalance(...)` that allows anyone to start a checkpoint if a validator has been slashed on the Beacon Chain.

Renzo protocol manages the checkpointing process for EigenLayer through two primary functions on the `OperatorDelegator` contract: `startCheckpoint(...)` and `verifyCheckpointProofs(...)`. This process is crucial for tracking validator balances, claiming rewards, and handling validator exits. The protocol's admin calls `startCheckpoint(...)` to initiate a checkpoint, which sets the `currentCheckpointTimestamp` to track the active checkpoint:

```
function startCheckpoint(...) external onlyNativeEthRestakeAdmin {
    if (currentCheckpointTimestamp != 0) revert CheckpointAlreadyActive();
    // ...
    eigenPod.startCheckpoint(true);
    // @audit The current checkpoint time is current block.timestamp
    currentCheckpointTimestamp = eigenPod.currentCheckpointTimestamp();
    // ...
}
```

Once the checkpoint proofs are verified, `verifyCheckpointProofs(...)` function updates the `lastCheckpointTimestamp` and clears the `currentCheckpointTimestamp`:

```
function verifyCheckpointProofs(...) external onlyNativeEthRestakeAdmin {
    // ...
    if (eigenPod.lastCheckpointTimestamp() == currentCheckpointTimestamp) {
        // ...
        // @audit Set the current as last finalized.
        lastCheckpointTimestamp = currentCheckpointTimestamp;
        // @audit Clear the current so that checkpoint can be started again.
        delete currentCheckpointTimestamp;
    }
}
```

The code above checks if the checkpoint has been completed by comparing the `eigenPod.lastCheckpointTimestamp()` with the `currentCheckpointTimestamp`. Since in the `EigenPod` contract the `lastCheckpointTimestamp` is updated only after checkpoint proofs are verified, this check works correctly. The checkpoint state stays synchronized between the protocol and the EigenPod.

However, in a specific scenario the `eigenPod.lastCheckpointTimestamp()` can go out of sync with the `currentCheckpointTimestamp`. In such a scenario, verifying the checkpoint via Renzo contracts won't be possible. Consider the following sequence of events:

1. The protocol admin starts a checkpoint via `startCheckpoint()`, setting `currentCheckpointTimestamp` to the current block timestamp. For the sake of example, lets assume the current block timestamp is 1. After a `startCheckpoint(...)` function is called in Renzo, this will be the protocol state:

```
        currentCheckpointTimestamp = 1
        lastCheckpointTimestamp = 0
        eigenPod.currentCheckpointTimestamp() = 1
        eigenPod.lastCheckpointTimestamp() = 0
```

2. Instead of using the protocol's `verifyCheckpointProofs(...)` function, a user directly calls `verifyCheckpointProofs(...)` on the EigenPod contract, completing the checkpoint. This updates the EigenPod's state but not the protocol's state:

```
        currentCheckpointTimestamp = 1
        lastCheckpointTimestamp = 0
        eigenPod.currentCheckpointTimestamp() = 0
        eigenPod.lastCheckpointTimestamp() = 1
```

3. A validator gets slashed on the Beacon Chain;
4. Anyone can then call `verifyStaleBalance(...)` directly on the EigenPod, which automatically starts a new checkpoint in the EigenPod. Lets assume the new timestamp is 3:

```
        currentCheckpointTimestamp = 1
        lastCheckpointTimestamp = 0
        eigenPod.currentCheckpointTimestamp() = 3
        eigenPod.lastCheckpointTimestamp() = 1
```

5. The user completes this new checkpoint by calling `verifyCheckpointProofs(...)` directly on the EigenPod:

```
        currentCheckpointTimestamp = 1
        lastCheckpointTimestamp = 0
        eigenPod.currentCheckpointTimestamp() = 0
        eigenPod.lastCheckpointTimestamp() = 3
```

Now, the protocol and EigenPod states are completely out of sync. When the protocol admin calls `verifyCheckpointProofs(...)`, the condition `eigenPod.lastCheckpointTimestamp() == currentCheckpointTimestamp` will be false (`3 != 1`), so the protocol's `currentCheckpointTimestamp` remains stuck at 1.

Since `currentCheckpointTimestamp` is non-zero, the admin cannot start a new checkpoint due to the check in `startCheckpoint(...)`:

```
function startCheckpoint() external onlyNativeEthRestakeAdmin {
    if (currentCheckpointTimestamp != 0) revert CheckpointAlreadyActive();
    // ...
}
```

The protocol does have an emergency function `emergencyTrackMissedCheckpoint(...)` that can update the `lastCheckpointTimestamp`, but it doesn't reset the `currentCheckpointTimestamp`.

This creates a permanent denial-of-service condition where the protocol cannot start new checkpoints or verify existing ones, preventing it from:

- Minting new shares for accrued validator rewards;
- Withdrawing ETH from validators that have exited the Beacon Chain;

If the checkpoints are tracked off-chain in an automatic manner by Renzo's backend, the described scenario is less likely to happen during regular protocol operations. If the attacker, however, turned out to be one of the Validator Operators utilized by Renzo, the timing of calls to `verifyCheckpointProofs(...)` and `verifyStaleBalance(...)` is less strict since the Operator can commit the slashable offence at any point in time and call `verifyStaleBalance(...)` right after.

Note that if the checkpoint is started on the Renzo side, it is marked as recorded. If, after that, the checkpoint is verified outside of Renzo and an additional checkpoint is missed, the `totalBeaconChainExitBalance` will not be updated correctly. If Renzo's admin attempts to call the `emergencyTrackMissedCheckpoint(...)` function with both missed checkpoints, the call will revert since the first one is marked as recorded. The `totalBeaconChainExitBalance` can only be updated for the second one.

Recommendation(s): Consider adding an additional emergency setter function for the `currentCheckpointTimestamp` to ensure that the protocol is still operational in a rare scenario where more than one checkpoint is missed.

Status: Acknowledged

Update from the client: For now considering the rarity of the issue. The offchain watcher will prioritize to sync the checkpoint through `OperatorDelegator::verifyCheckpointProofs` as step 4 and 5 will not be executed atomically which provides a time window during which admin can prioritize and sync the checkpoints on `OperatorDelegator`.

6.5 [Low] The `emergencyTrackSlashedQueuedWithdrawalDelta(...)` function can be used to artificially decrease the TVL to get better exchange rate on `ezETH`

File(s): `contracts/Delegation/utils/OperatorDelegatorAdminLib.sol`

Description: The `emergencyTrackSlashedQueuedWithdrawalDelta(...)` function from the `OperatorDelegator` contract can be called by anyone to signal that the shares queued for withdrawal have been slashed while waiting in the queue. The function utilizes the `OperatorDelegatorAdminLib`, where the `trackSlashedQueuedWithdrawalDelta(...)` function implements all the functionality.

The function first queries the currently withdrawable shares for a particular withdrawal root (the root must be present in the withdrawal queue). It then compares the currently withdrawable value with the number of shares queued for deposit when the `queueWithdrawal(...)` function was called. If there is a difference between the two, it means that there was a slash, and the slashing delta must be subtracted from the TVL calculation.

The problem with the code handling this logic is that the value of shares queued for deposit and stored in the `Withdrawal` struct is scaled by the deposit scaling factor. In a situation where the staker delegated to an operator that was previously slashed, the scaled shares will be higher than the deposit amount provided by the staker to offset the prior slash. In such scenario even when there was no slash in between the deposit and withdrawal, anyone can call the `emergencyTrackSlashedQueuedWithdrawalDelta(...)` function to signal a slash since the `slashingDelta` will be positive.

```
function trackSlashedQueuedWithdrawalDelta(
    bytes32[] calldata withdrawalRoots,
    // ...
) external {
    for (uint256 i = 0; i < withdrawalRoots.length;) {
        // ...
        (
            IDelegationManager.Withdrawal memory withdrawal,
            // @audit currentShares are WITHDRAWABLE shares
            uint256[] memory currentShares
        ) = delegationManager.getQueuedWithdrawal(
            withdrawalRoots[i]
        );

        for (uint256 j = 0; j < withdrawal.scaledShares.length;) {
            // ...
            // @audit Since scaled shares are not affected by the slashing factor,
            // They will be higher than currently withdrawable shares even if there was
            // no new slash.
            uint256 slashingDelta = withdrawal.scaledShares[j]
                > currentShares[j]
                ? withdrawal.scaledShares[j] - currentShares[j]
                : 0;
            // @audit Adjust the slashed amount. This will lower the TVL calculation.
            totalTokenQueuedSharesSlashedDelta[underlyingToken] +=
                slashingDelta;
            // ...
        }
        // ...
    }
}
```

The `totalTokenQueuedSharesSlashedDelta` mapping is updated with the latest slashing delta. Since this value is used to decrease the value of 'in-flight' shares during the TVL calculation, the TVL will be lower than it should be. It allows the user to call this function before depositing to get a more favorable exchange rate when minting `ezETH` tokens.

Recommendation(s): Since the slashing delta informs about the difference between withdrawable shares at two points in time, consider adjusting the scaled shares amount by the slashing factor at the time of the deposit to make the computation correct.

Status: Fixed

Update from the client: Updated the code to calculate slashing delta in the denomination of `withdrawableShares` instead of `scaledShares`. Code Ref.

6.6 [Low] Users might get a worse exchange rate on ezETH due to the TVL spike after AVS are the withdrawal

File(s): contracts/Delegation/OperatorDelegator.sol

Description: The `queueWithdrawals(...)` function in the `OperatorDelegator` contract allows the protocol admin to start a withdrawal from the specified token strategies in EigenLayer. In the function arguments, the Renzo admin will provide the strategies to withdraw from and the amount of tokens to be withdrawn from EigenLayer.

The EigenLayer will decrease the `OperatorDelegator`'s withdrawable shares as soon as the withdrawal is queued. Since Renzo's TVL calculation before queuing the withdrawal was based on the amount of withdrawable shares, this difference must be accounted for.

Renzo tracks the shares in EigenLayer's withdrawal queue to include the in-flight tokens in the TVL calculation. This is done via the `queuedShares` storage variable incremented in the `queueWithdrawals(...)` function in the `OperatorDelegator` contract by the `depositShares` amount. Thanks to this mechanism, whenever the TVL is decreased by the amount of the withdrawable shares when the withdrawal is queued, it is incremented right after by the deposit share amount expected to be received. This ensures that the TVL remains constant.

```
function queueWithdrawal(
    IERC20[] calldata tokens,
    uint256[] calldata tokenAmounts,
    // ...
) external returns (/*...*/) {
    // ...
    for (uint256 i; i < tokens.length;) {
        if (address(tokens[i]) == IS_NATIVE) {
            // ...
            queuedWithdrawalParams[0].depositShares[i] = tokenAmounts[i];
        } else {
            // ...
            queuedWithdrawalParams[0].depositShares[i] =
            tokenStrategyMapping[tokens[i]].underlyingToSharesView(
                tokenAmounts[i]
            );
        }
        // ...
        // @audit-issue In case of a 50% slash on 100 deposit shares, only 50
        // shares are withdrawable and accounted in the TVL before the withdrawal.
        // The queuedShares mapping gets incremented by 100, which inflates the TVL by 50 shares.
        queuedShares[address(tokens[i])] +=
            queuedWithdrawalParams[0].depositShares[i];
        // ...
    }
    // ...
}
```

Tracking the in-flight tokens via the `queuedShares` mapping works whenever the deposit shares are equivalent to the withdrawable shares. This, however, might not be the case if the operator was previously slashed. Consider the following scenario:

1. Renzo's TVL is 100 at the beginning (100 deposit shares delegated to a single operator);
2. The operator gets slashed 50% by an AVS (or Beacon Chain; it does not matter for the example);
3. Renzo's TVL drops to 50 since the `getWithdrawableShares(...)` from EigenLayer represents the updated slashed amount. The deposit shares are still 100 since they represent the nominal deposit value;
4. Since EigenLayer calculates the withdrawable shares on-demand with the latest slashing factor, to make the full withdrawal from EigenLayer, Renzo's admin must provide 100 deposit shares as an input amount. This amount will be adjusted and Renzo will be credited with 50 shares once withdrawal is completed;
5. Once the admin queues the full withdrawal specifying 100 deposit shares as input amount, all available shares in EigenLayer (50 shares) are decreased to 0;
6. Renzo increases the `queuedShares` mapping by 100 deposit shares to account for the in-flight tokens. The TVL rises to 100;
7. After the withdrawal period, once the `completeQueuedWithdrawal(...)` function is called and withdrawal is complete, the TVL drops to 50 since only 50 shares were withdrawn from EigenLayer;

Since the TVL is inflated right after queueing the withdrawal, any user that deposits funds into the protocol at that time will get a worse exchange rate on `ezETH` tokens. The problem is also present in the `emergencyTrackQueuedWithdrawal(...)` function.

Recommendation(s): Starting from the slashing update, EigenLayer distinguishes between the deposit and withdrawable shares. The same distinction should be implemented in Renzo's smart contracts. Consider separating the deposit amount of shares used as input to the function from the amount of withdrawable shares utilized during the TVL calculation. The fix for this issue could pair with the fix for [High] Incorrect `queuedShares` scaling might lead to DoS and incorrect TVL calculation issue.

Status: Fixed

Update from the client: Updated the code to increment `queuedShares` with the `withdrawableShares` amount instead of depositShares. While `queueWithdrawal` with `depositShares`. Code Ref.

6.7 [Info] The `emergencyTrackQueuedWithdrawals(...)` function doesn't verify the queued withdrawal ownership, potentially leading to inflated TVL

File(s): `contracts/Delegation/OperatorDelegator.sol`

Description: The `emergencyTrackQueuedWithdrawals(...)` function is a permissioned function and can only be called by the `EmergencyWithdrawTrackingAdmin`. This function is designed to track pending queued withdrawal shares if an Operator forcefully undelegates the `OperatorDelegator`.

The function utilizes the `trackQueuedWithdrawals(...)` function from the `OperatorDelegatorAdminLib`, which updates the `queuedShares` mapping used in TVL calculations:

```
// @audit OperatorDelegatorAdminLib.sol
function trackQueuedWithdrawals(
    IDelegationManager.Withdrawal[] calldata withdrawals,
    // ...
) external {
    // ...
    for (uint256 i = 0; i < withdrawals.length; ) {
        // ...
        for (uint256 j = 0; j < withdrawals[i].scaledShares.length; ) {
            // @audit-issue Increases queuedShares mapping without
            // verifying if withdrawal.staker is this contract.
            queuedShares[address(tokens[j])] += withdrawals[i].scaledShares[j];
            // ...
        }
        // ...
    }
}
```

However, the function lacks validation to ensure that the tracked withdrawal belongs to the `OperatorDelegator` contract. While the function verifies that the withdrawal is pending in EigenLayer, it doesn't check the `staker` field of each withdrawal to confirm the ownership.

Suppose an admin makes an error and calls this function with a queued withdrawal belonging to another EigenLayer user. In that case, it'll incorrectly increase the `queuedShares` mapping and consequently inflate the reported TVL despite the fact that this withdrawal does not belong to the `OperatorDelegator` contract.

Recommendation(s): Consider implementing a check that verifies each withdrawal's `staker` field matches the address of the `OperatorDelegator` contract before adding the corresponding shares to the `queuedShares` mapping.

Status: Fixed

Update from the client: Added a check to verify the staker should be OperatorDelegator at - Code Ref.

6.8 [Info] The `fillERC20withdrawBuffer(...)` function doesn't refund excess tokens sent

File(s): `contracts/Deposits/DepositQueue.sol`

Description: The `fillERC20withdrawBuffer(...)` function from the `DepositQueue` contract is used to refill the ERC20 withdrawal buffer. It is utilized when new deposits enter the protocol via `RestakeManager`'s `deposit(...)` function or during the completion of queued withdrawals in the `OperatorDelegator` contract.

In each case, if the token amount about to be transferred to the buffer is greater than the buffer's token deficit, the `amount` will be reduced before the transfer so that no extra tokens are sent to refill the buffer. However, the `fillERC20withdrawBuffer(...)` function in the `DepositQueue` contract is permissionless and can be called directly. If the admin (or a user) attempts to fill the buffer outside of the usual flow and sends an `amount` that is greater than the withdrawal queue deficit, the extra tokens will remain in the `WithdrawQueue` contract. They will be treated as a donation to be used for user withdrawals.

```
function fillERC20WithdrawBuffer(
    address _asset,
    uint256 _amount
) external nonReentrant onlyDepositQueue {
    // ...
    // @audit The provided _amount can be greater than queueFilled.
    queueFilled = _checkAndFillWithdrawQueue(_asset, _amount);
    // @audit The full _amount is sent to the contract regardless of queueFilled.
    IERC20(_asset).safeTransferFrom(msg.sender, address(this), _amount);

    emit ERC20BufferFilled(_asset, _amount - queueFilled);
}
```

Recommendation(s): If overfilling the buffer is a desired scenario, consider adding a similar warning comment for the users to the one present in the NatSpec of the `forwardFullWithdrawalETH(...)` function, stating that this function should not be used to send tokens to the protocol.

Otherwise, consider changing the logic so that only the necessary `queueFilled` amount is transferred to the `WithdrawalQueue` contract. The rest of the funds could stay in the `DepositQueue` and generate a higher yield for the users.

Status: Acknowledged

Update from the client: Overfilling the buffer is a desired scenario. Added the warning at - Code Ref.

6.9 [Info] The `totalBeaconChainExitBalance` can be inflated if no checkpoint was finalized after the validator got slashed

File(s): `contracts/Delegation/OperatorDelegator.sol`

Description: Under normal circumstances, the `totalBeaconChainExitBalance` variable in the `OperatorDelegator` contract is designed to track ETH that has exited the Beacon Chain and is expected to be received by the contract.

Whenever a checkpoint is finalized on the EigenLayer side, the `_verifyCheckpointProof(...)` function will try to determine whether the effective balance of a particular validator has dropped to zero on the Beacon Chain. If it has, then it means that the validator is fully withdrawn. The function compares the previously recorded validator's balance with the current one (zero balance) and marks the difference as the validator's exit balance. Renzo uses this variable to determine how many native tokens received by the protocol contracts come from Beacon Chain withdrawals and how many from rewards. The funds are routed differently based on the origin.

```
function _verifyCheckpointProof(...) {
    // ...
    // @audit Previously recorded validator's beacon chain balance.
    prevBalanceGwei = validatorInfo.restakedBalanceGwei;

    // @audit New effective balance. It can be 0 if the validator is fully withdrawn.
    uint64 newBalanceGwei = BeaconChainProofs.verifyValidatorBalance({
        balanceContainerRoot: balanceContainerRoot,
        validatorIndex: validatorIndex,
        proof: proof
    });

    if (newBalanceGwei != prevBalanceGwei) {
        // @audit The difference between the two means that:
        // - there could be rewards (positive delta)
        // - there could be a slash (negative delta)
        // - there could be a full withdrawal (new balance is 0)
        balanceDeltaGwei = int64(newBalanceGwei) - int64(prevBalanceGwei);
        // ...
    }
    // ...
    if (newBalanceGwei == 0) {
        activeValidatorCount--;
        validatorInfo.status = VALIDATOR_STATUS.WITHDRAWN;
        // @audit The previously recorded balance is used as exit balance
        // for a fully withdrawn validator.
        exitedBalanceGwei = uint64(-balanceDeltaGwei);
        // ...
    }
    // ...
}
```

While Renzo's approach of depending on the `exitedBalanceGwei` to track the withdrawals is correct, it is under the assumption that the checkpoints are performed often enough. Since the exit balance is based on the previously stored balance, the resulting exit balance will be incorrect if that value is not up to date. Consider the following scenario of a validator that was slashed:

- The slashed validator is automatically initiated for exit in 36 days;
- The validator can suffer an initial slashing penalty of up to `1 ETH`;
- If the checkpoint is started and finalized right after the initial penalty, the validator's effective balance can be recorded as `31 ETH` (initial stake of `32` minus `1 ETH` penalty);
- Halfway through the withdrawal period (after 18 days), the validator may incur a correlation penalty (e.g., an additional 50% of its effective balance, reducing it to `15.5 ETH`) due to coordinated misbehavior.
- **If no checkpoint is performed**, the stored balance remains `31 ETH`. Upon exit, the `exitedBalanceGwei` is set to `31 ETH` despite the balance being `15.5 ETH`;
- This inflates the `totalBeaconChainExitBalance` and misclassifies slashed funds as exit proceeds. Future rewards will decrease the inflated exit balance, reducing protocol fees (which are only earned on rewards);

Additionally, inactivity leaks during this period could further reduce the effective balance, exacerbating the discrepancy if checkpoints are skipped.

Recommendation(s): Consider monitoring all validators initiated for exit. This includes ejected validators due to inactivity leaks, voluntary exits, and slashed validators. If their effective balance changes before they are marked as withdrawn, consider starting and finalizing the checkpoint to ensure correct rewards accounting.

Status: Acknowledged

Update from the client: The protocol constantly keeps on running and proving checkpoint periodically. On top of that, watchers will be implemented to monitor for any slashing events and report it to the protocol to price in any slashing event. Also, EigenLayer has permissionless functionality for anyone to report a validator slashing through `verifyStaleBalance`. Through which anyone can report the validator slashing and start a checkpoint on OperatorDelegator Owned EigenPod which can be finalized in a permissionless manner.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about the Renzo protocol documentation

The documentation for the Renzo protocol is comprehensive and well-structured, providing detailed information on its architecture and functionality. The official website includes thorough explanations of the protocol's features and use cases, allowing readers to understand the core mechanics of Renzo, particularly its integration with EigenLayer for Ethereum restaking.

Additionally, the code itself is well-commented with extensive NatSpec annotations, making the codebase highly readable and facilitating a smooth review process. The Renzo team has made sure to provide clear, technical insights that helped the Nethermind Security team in understanding the project's inner workings and core logic.

The Renzo team was also proactive in making themselves available for sync calls to address any questions or concerns raised during the review. These discussions ensured that the security team had all the necessary context to perform a thorough analysis.

One area where additional documentation or comments would be beneficial is around handling edge cases, particularly when dealing with stETH price via secondary market rates. Without extra context or detailed explanations in this area, it can be challenging to fully understand the logic behind the handling of these cases, which may lead to confusion when evaluating the behavior of the protocol in certain conditions.

8 Test Suite Evaluation

The Renzo protocol's test suite is robust and clearly outlines the contract specifications. The test suite catches any change to the business logic. To assess the test coverage for these files, the Nethermind Security team applied a technique called mutation testing to uncover the untested paths. This technique introduces slight modifications to the code called "mutations" or "mutants." An example of a mutation is, for example, changing the operator in an expression `(a + b)` to `(a - b)`, or removing a `require(a > b)` statement entirely from the code. With these changes, the code no longer follows the expected business logic of the application, and the test suite should reflect that by failing.

Evaluation of the test suite with mutation testing consists of two phases:

a. Generating the modified version of each contract, called "mutants."

b. Inserting the mutant into the original codebase and running the test suite.

Only one modification can be tested at a time. If the contract has ten mutations, the test suite must run ten times (once for every mutation). If any of the tests fail, it means that the test suite caught the change in the code. Whenever that happens, the particular mutant is considered "slain" or "killed" and is removed from the mutant's set. If that does not occur, a new test case can be added to cover the code branch to "kill" the mutant.

The following table outlines the mutation analysis results performed on Renzo's core smart contracts at two key points: the initial audit commit (start of the engagement) and the final commit (after fixes were implemented in response to the audit findings). Each row corresponds to a contract tested using mutation testing. For both the initial and final commits, the table shows the number of mutants generated, how many were "slain" (i.e., caught by the test suite), and the corresponding mutation score. The final column highlights the delta in mutation score, indicating improvements or regressions in test suite effectiveness. A higher mutation score reflects stronger test coverage and better bug-detection capabilities.

Contract	Start (Slain / Total)	Start Score	Final (Slain / Total)	Final Score	Score Delta
`OperatorDelegatorAdminLib.sol`	95 / 103	92.23%	136 / 181	75.14%	-17.09%
`METHShim.sol`	4 / 4	100.00%	2 / 4	50.00%	-50.00%
`OperatorDelegator.sol`	261 / 327	79.82%	226 / 286	79.02%	-0.80%
`PaymentSplitter.sol`	85 / 98	86.73%	86 / 98	87.76%	+1.03%
`EzEthToken.sol`	6 / 11	54.55%	8 / 11	72.73%	+18.18%
`DepositQueue.sol`	127 / 133	95.49%	118 / 133	88.72%	-6.77%
`RoleManager.sol`	3 / 5	60.00%	4 / 5	80.00%	+20.00%
`WithdrawQueue.sol`	334 / 426	78.40%	335 / 426	78.64%	+0.24%
`WBETHShim.sol`	4 / 4	100.00%	2 / 4	50.00%	-50.00%
`RenzoOracle.sol`	150 / 167	89.82%	145 / 167	86.83%	-2.99%
`BalancerRateProvider.sol`	25 / 25	100.00%	23 / 25	92.00%	-8.00%
`RestakeManager.sol`	275 / 278	98.92%	260 / 278	93.53%	-5.39%
Total	1362 / 1574	86.14%	1345 / 1618	83.13%	-3.01%

The following is a list of code modifications not caught by the existing test suite. Each point highlights a scenario that could benefit from higher test coverage to enhance the protocol's overall security and resilience in the next code change iterations as the project evolves.

OperatorDelegator.sol

- In the `stakeEth(...)` function, the call to `eigenPodManager.stake(...)` can be removed with no effect on the test suite. Consider asserting the state changes caused by a call to the `EigenPodManager` contract.
- In the `startCheckpoint(...)` function, the call to `eigenPod.startCheckpoint(...)` can be removed with no effect on the test suite. Consider adding assertions around state changes that the `startCheckpoint(...)` causes.
- In the `claimRewards(...)` function, the call to `rewardsCoordinator.processClaim(...)` can be removed with no effect on the test suite. Consider increasing the coverage around the state changes caused by the call to `claimRewards(...)`.
- No test case covers the scenario where the claim token is `WETH` in the `claimRewards(...)` function. The condition in the `if` statement can be hardcoded to `false` with no effect on the test suite.
- The `catch` branch of the `getStakedETHBalance(...)` function, when `podOwnerShares` are negative, is undertested. The subtraction of `uint256(podOwnerShares)` can be replaced with addition, multiplication, or division with no effect on the test suite.
- Similarly, the `catch` branch of the `getStakedETHBalance(...)` function, when `podOwnerShares` are NOT negative, is undertested. The addition can be changed to subtraction. Consider adding a test case to validate the return value of `getStakedETHBalance(...)` based on different values in `queuedShares`, `stakedButNotVerifiedETH`, and `podOwnerShares`.
- There are no test cases for `InvalidZeroInput` errors in the `initialize(...)` function. The calls to `_checkZeroAddress(...)` can be removed with no effect on the test suite. This applies to the parameters: `_roleManager`, `_strategyManager`, `_restakeManager`, `_delegationManager`, and `_eigenPodManager`.

- In the `stakeEth(...)` function, the addition of `msg.value` in the condition `validatorCurrentStakedButNotVerifiedEth + msg.value > MAX_STAKE_BUT_NOT_VERIFIED_AMOUNT` can be changed to subtraction or multiplication with no effect on the test suite. Consider adding assertions on the state changes inside and outside the `if` statement.
- In the `startCheckpoint(...)`, `verifyCheckpointProofs(...)`, and `claimRewards(...)` functions, the call to `_recordGas(...)` can be removed with no effect on the test suite.
- In the `verifyCheckpointProofs(...)` function, all test cases cover the scenario where the condition `eigenPod.lastCheckpointTimestamp() == currentCheckpointTimestamp` is `true`. Consider validating that no state changes happen when the condition is `false`.
- In the `_recordGas(...)` function, the multiplication in the `gasSpent` calculation can be updated with addition, subtraction, or division with no effect on the test suite. Consider adding more test cases around expected gas spending.
- No test case asserts revert with `TransferFailed` in the `_refundGas(...)` function.
- No test case asserts revert with `InvalidZeroInput` in the `setDelegateAddress(...)` function.
- No test case asserts revert with `InvalidZeroInput` in the `_reduceQueuedShares(...)` function.
- In the `_reduceQueuedShares(...)` function, the second if statement `queuedWithdrawalTokensSlashedDelta[withdrawalRoot][address(tokens[i])] > 0` is not tested at all. The condition can be hard-coded to `true` or `false` with no effect on the test suite. The logic inside the `if` block is also untested.
- The first `if` statement (condition and logic) in the `_fillBufferAndReDeposit(...)` function can be arbitrarily changed with no effect on the test suite. Consider adding test cases around the `bufferToFill` adjustments.
- The call to `restakeManager.depositQueue().fillERC20withdrawBuffer(...)` in the `_fillBufferAndReDeposit(...)` function can be removed with no effect on the test suite.
- The `_processETH(...)` function is only tested when `totalBeaconChainExitBalance > 0` (the first `if` is always hit).
- No test case asserts the differences caused by an early return in the `_processETH(...)` function. The logic executed after the first `if` statement (gas refund and call to deposit queue) is not tested.
- The `receive(...)` function is not tested when the message sender is `WETH` (no state changes caused by `_processETH(...)`).
- No test case asserts revert with `MismatchedArrayLengths` in the `completeQueuedWithdrawals(...)` function.
- No test case asserts revert with `NotEigenLayerRewardsAdmin` error.

RestakeManager.sol

- The `stakeEthInOperatorDelegator(...)` function is only tested in scenarios where the provided operator delegator is found. Consider adding a test case that triggers the `NotFound` error.
- The `getCollateralTokenIndex(...)` check can be removed from the `setTokenTvlLimit(...)` function with no effect on the test suite. Consider adding a test case that validates the revert scenario when the token is not in the list.

OperatorDelegatorAdminLib.sol

- In the `verifyWithdrawalCredentials(...)` function, the `if` statement can be hardcoded to `false` with no effect on the test suite. Consider adding test cases that differentiate whether `validatorStakedButNotVerifiedEth` is zero or non-zero.
- In the `verifyWithdrawalCredentials(...)` function, the `++i` can be removed with no effect on the test suite. Consider adding a test case where multiple validators are verified at once.
- No test case exists for a revert with the `MismatchedArrayLengths` error in the `queueWithdrawals(...)` function.
- In the `trackMissedCheckpoint(...)` function, the condition `missedCheckpoints[i] > latestCheckpoint` can be hardcoded to `true` with no effect on the test suite.
- In the `trackMissedCheckpoint(...)` function, the `++i` can be removed with no effect on the test suite. Consider adding a test case where multiple missed checkpoints are processed.
- In the `trackSlashedQueuedWithdrawalDelta(...)` function, the `++j` can be removed with no effect on the test suite. Consider adding a test case where multiple tokens are processed in a single queued withdrawal.
- In the `trackQueuedWithdrawals(...)` function, the expression `queuedWithdrawal[withdrawalRoot] = true` can be removed with no effect on the test suite.

WithdrawQueue.sol

- No test case exists for a revert with `NotRestakeManager` when `onlyRestakeManager(...)` is called. Consider adding more test cases related to access control.
- In `initialize(...)`, the value of `_coolDownPeriod` can be hardcoded to any value with no effect on the test suite.
- No test cases in `setWhiteListed(...)` validate setting accounts to `false`. Consider adding test cases where different accounts are added and removed from the whitelist in a single call to the `setWhiteListed(...)` function.

- No test cases exist for whitelisting multiple accounts with `setWhiteListed(...)`. The `++i` can be removed with no effect on the test suite.
- No test case exists for setting multiple `stETH` depositors using the `setStETHDepositors(...)` function. The `++i` can be removed with no effect on the test suite.
- No test case validates the values emitted by the `EthBufferFilled` event in the `fillEthWithdrawBuffer(...)` function.
- No test case validates the values emitted by the `ERC20BufferFilled` event in the `fillERC20WithdrawBuffer(...)` function.
- In the `claim(...)` function, the `EarlyClaim` revert condition can be changed from `timestamp - createdAt` to multiplication or division without affecting the test suite. Consider adding more test cases at different claim timestamps.
- In the `claimETH(...)` function, the condition `claimAmountToRedeem < _withdrawRequest.amountToRedeem` is met in all test cases. Consider adding test cases where this condition is `false` and the amount to redeem is not lower at claim time.
- No test case exists for the `TransferFailed` revert in the `claimETH(...)` function.
- In the `claimERC20(...)` function, the condition for reverting with the `QueuedWithdrawalNotFilled` error is undertested. The arguments in the condition can be swapped with no effect on the test suite.
- No test case validates the ordering of `withdrawRequests` after calling the `claimERC20(...)` function. Consider verifying whether the swap and pop operations work correctly.
- The `_checkAvailableETHCollateralValue(...)` function can be removed from the `withdrawETH(...)` function with no effect on the test suite. Consider validating that withdrawals revert if there is insufficient collateral.
- In the `withdrawETH(...)` function, the expression `queued = true` can be removed entirely with no effect on the test suite. Consider verifying that the `WithdrawRequestCreated` event emits the correct `queued` value.
- Consider adding test cases around the withdrawal request ID emitted by the `WithdrawRequestCreated` event in the `withdrawETH(...)` function.
- No test case ensures that the `withdrawRequestNonce` is incremented as a result of calling the `withdrawERC20(...)` function.
- In the `withdrawERC20(...)` function, the condition `amountToRedeemAtSecondaryRate > 0` is met in all test cases. Consider adding test cases where this condition is `false`.
- In the `_checkAndClaimAtSecondaryRate(...)` function, the expression `stETHDepositorsWithdrawalAmount[_withdrawHash] = 0` can be removed or set to an arbitrary number with no effect on the test suite.
- In all test cases, the condition `queueDeficit > 0` in the `_checkAndFillWithdrawQueue(...)` function is always `true`. Consider adding a test case where no state changes occur.
- In the `availableCollateralAmount(...)` function, the condition `_asset != IS_NATIVE` can be hardcoded to `true` or `false` without affecting the test suite. Consider adding assertions to validate the differences between the `if` and `else` branches.
- In the `availableCollateralAmount(...)` function, the `collateralIndex` of ETH in the `else` branch can be hardcoded to any value instead of being dynamically computed as `_operatorDelegatorTokenTVLs[0].length - 1`. Consider adding test cases with a larger number of tokens so that the collateral index value matters.
- In the `availableCollateralAmount(...)` function, `totalCollateralAmount` can be arbitrarily changed, suggesting that this code branch is not tested. Consider adding test cases targeting the `else` branch of the `availableCollateralAmount(...)` function. The same applies to the `if` branch.
- In the `availableCollateralAmount(...)` function, `totalCollateralAmount` can be left unincremented by `assetTVL` with no effect on the test suite. Consider adding assertions around the expected TVL for mixed token types.
- In the `availableCollateralAmount(...)` function, the deduction of `queueDeficit` can be removed with no effect on the test suite.
- Similar cases are not tested in the `_checkAvailableETHCollateralValue(...)` function. The `collateralIndex`, `totalCollateralValue`, and `queueDeficitValue` can be changed arbitrarily with no effect on the test suite. The same applies to the `_checkAvailableERC20CollateralValue(...)` function.
- In `_checkAvailableETHCollateralValue(...)`, consider adding a test case for the `else` branch (revert with `NotEnoughCollateralValue`).
- In the `_checkAvailableERC20CollateralValue(...)` function, the first call to `renzoOracle.lookupTokenValue(...)` can be removed with no effect on the test suite. The test suite does not assert that the input `_amount` is overridden with the looked-up token value.

DepositQueue.sol

- No test case exists for a revert with the `NotERC20RewardsAdmin` error.

RenzoOracle.sol

- The `lookupTokenSecondaryValue(...)` function lacks a test case for an early return when `address(stETHSecondaryOracle) == address(0)`.
- The `lookupTokenSecondaryValue(...)` function lacks a test case for reverting with the `OraclePriceExpired` error. The same applies to the `lookupTokenSecondaryAmountFromValue(...)` function.

- The `lookupTokenSecondaryValue(...)` function lacks a test case for reverting with the `InvalidOraclePrice` error. The same applies to the `lookupTokenSecondaryAmountFromValue(...)` function.
- The `lookupTokenSecondaryAmountFromValue(...)` function lacks a test case for an early return when `address(stETHSecondaryOracle) == address(0)`.
- The `lookupTokenValue(...)` function lacks a test case for reverting with the `OracleNotFound` error.

EzEthToken.sol

- No test case validates the revert with the `NotEzETHMinterBurner` error from the `onlyMinterBurner` modifier.
- No test case validates the revert with the `NotTokenAdmin` error from the `onlyTokenAdmin` modifier.
- No test case validates the revert with the `InvalidZeroInput` error from the `initialize(...)` function.

PaymentSplitter.sol

- No test case ensures that the list of recipients remains correct after calling `removeRecipient(...)`. The removed recipient can be swapped with a different recipient without affecting the test suite.
- The `addToRecipientAmountOwed(...)` function lacks a test case for the `NotFound` error.
- The `addToRecipientAmountOwed(...)` function should be tested with more than one recipient in the list. The `i++` statement can be removed without affecting the test suite.
- The `subtractFromRecipientAmountOwed(...)` function should be tested with more than one recipient in the list. The `i++` statement can be removed without affecting the test suite.
- The `subtractFromRecipientAmountOwed(...)` function lacks a test case for the `NotFound` error.
- The early return condition `amountLeftToPay == 0` in the `receive(...)` function is not tested.
- The `continue` case in the `receive(...)` function is not tested. The `amountToPay == 0` condition is also untested.
- There is no test case for a failure scenario when calling `recipients[i].call` in the `receive(...)` function.
- There is no test case for a scenario where the condition `amountLeftToPay > DUST_AMOUNT` evaluates to false in the `receive(...)` function.
- There is no test case for a failure scenario when calling `fallbackPaymentAddress.call` in the `receive(...)` function.

Remarks about the Renzo Protocol Test Suite

The test suite for the Renzo Protocol is robust in terms of unit testing for core functionality. Changes in business logic are effectively caught, ensuring that key components behave as expected under various conditions.

The weakest aspect of the test suite lies in the mocking of some of EigenLayer's interactions. Many of the issues identified in this report could have been detected with an end-to-end test suite running on a forked network using the deployed version of EigenLayer. Relying solely on mocks limits the ability to catch integration issues that only surface in a real execution environment.

While testing slashing behaviors originating from the beacon chain is inherently difficult, additional coverage around AVS slashing should be considered, as it is more feasible to test. Strengthening this area would provide better assurances around the integration.

Once EigenLayer releases its final version intended for mainnet deployment, it is highly recommended to implement comprehensive end-to-end tests for all core protocol flows. Areas where issues (slashing and scaling of shares) have been identified should be prioritized.

8.1 Automated Tools

8.1.1 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.