certora

Security Assessment Report

LIDO

Lido V3

January 2026

Prepared for Lido

Table of contents

certora LIDO

Project Summary

Project Scope

Project Name	Repository Link	Branch/PR/Commit Hash	Platform
Core	lidofinance/core/	Branch: #PR84 On-Chain,, commit 28571	On-Chain
Easy-Track	lidofinance/easy-track	Branch: #PR86 On-Chain,, commit 3aaf9	On-Chain
Core	lidofinance/core/	Branch: #PR1287, Commit 2353e	On-Chain
Core	lidofinance/core/	Branch: #PR1287, Commit 15d69	On-Chain
Core	lidofinance/core/	Branch: #PR1467, Commit 86437	On-Chain
Core	lidofinance/core/	Branch: #PR1491, Commit 30106	On-Chain
Core	lidofinance/core/	Branch: #PR1512, Commit ebf90	On-Chain
Core	lidofinance/core/	Branch: #PR1528, Commit 32883	On-Chain
Core	lidofinance/core/	Branch: #PR1533, Commit 28b2f	On-Chain
Core	lidofinance/core/	Branch: #PR1541, Commit 0d3c6	On-Chain

Core	lidofinance/core/	Branch: #PR1549, Commit 7718c	On-Chain
Core	lidofinance/core/	Branch : #PR1568, Commit b983	On-Chain
Core	lidofinance/core/	Branch : #PR1630, Commit 33f2	On-Chain

Project Overview

This document describes the security assessment of Lido V3 Staking Infrastructure. The work was undertaken from **June 20, 2025** to **December 4th, 2025** and **from January 5th** to **January 8th**.

Last audited commit : **33f2f59156697aae93cdea2d0984de7be347e3af .**

During the manual audit, the Certora team discovered the bugs listed on the following page.

Protocol Overview

Lido V3 introduces Staking Vaults (stVaults) as a modular extension to Lido's liquid staking protocol. The upgrade enables users to stake ETH with specific node operators while still minting stETH, preserving liquidity through an overcollateralization model. This maintains stETH fungibility and security while allowing customizable staking setups.

Layered Security Model

The protocol separates into two layers. The Essential Layer contains rigorously governed foundational contracts: StakingVault (isolated positions with 0x02 withdrawal credentials), VaultHub (central coordinator enforcing collateralization), LazyOracle (asynchronous vault reporting), OperatorGrid (risk tier management), and PredepositGuarantee (deposit frontrunning protection). The Utility Layer provides optional composable components like VaultFactory and Dashboard for enhanced user experience without compromising core security.

Previously, stETH was minted only against ETH in the Lido Core pool. Lido V3 extends this to support external ETH backing through StakingVault contracts. Users deposit ETH into vaults and receive stETH based on a reserve ratio—for example, 100 ETH deposits might mint 90 stETH at a 90% reserve ratio. VaultHub enforces that all stETH remains fully redeemable and maintains system-wide solvency. Under normal conditions, vault-specific slashing or performance issues don't affect other stETH holders. Only in extreme scenarios can losses be socialized or internalized across the stETH supply through governance-approved mechanisms.

StakingVault Mechanics

Each **StakingVault** represents an isolated staking position controlled by a single owner and serviced by one node operator. The vault uses 0x02 withdrawal credentials enabling EIP-7002 triggerable exits. Owners can deposit ETH to activate validators, connect to Lido via VaultHub to mint stETH against locked collateral, receive staking rewards, manage fee distribution, and trigger validator exits. The setup is non-custodial—operators manage validators but cannot access deposited ETH.

VaultHub serves as the central coordination layer. It handles stETH minting and burning, enforces reserve ratio requirements based on vault risk tiers, and continuously monitors vault health via oracle reports. When vaults underperform or become undercollateralized, VaultHub can initiate rebalancing, forced disconnection, or bad debt socialization. It also processes connection and

disconnection requests, coordinates redemption settlement, distributes fees to the protocol treasury, and tracks total locked value across all vaults to maintain protocol-wide invariants.

OperatorGrid manages the tiered risk framework by organizing node operators into groups, each containing multiple tiers with distinct reserve ratios, share limits, forced rebalance thresholds, and fee structures. Every vault starts in the default tier (tier 0) and can transition to operator-specific tiers through multi-party confirmation between vault owner and node operator. A vault's effective stETH minting capacity is constrained by both its tier's share limit and the operator group's total share limit, each reduced by existing liability shares from other vaults. Higher-numbered tiers impose progressively stricter reserve ratios and fees, creating economic pressure that routes stake toward less-saturated operators and prevents validator set centralization.

The **Predeposit Guarantee (PDG)** mechanism protects against deposit frontrunning. Before executing a full validator deposit, the node operator precommits by calling the PDG contract with a 1 ETH bond, submitting the validator pubkey and BLS signature. The EIP-2537 BLS precompile verifies the signature on-chain, and withdrawal credentials are proven correct via EIP-4788 beacon root proofs. After successful precommitment, the vault can deposit the remaining ETH (31 ETH for a standard validator, more for consolidation, top-ups). Only the precommitted pubkey can be used, preventing operators from frontrunning deposits. The guarantor receives their bond back after successful deposit execution. A bypass mechanism exists for vault owners who trust their operator, though this only shifts risk to the owner—the protocol itself remains protected.

Easy-Track Scripts

Easy Track provides a lightweight governance mechanism for routine vault management operations through specialized EVM Script factories, allowing authorized parties to execute administrative actions without requiring full DAO token voting. The vault-specific factories enable critical operations including OperatorGrid management (registering/updating groups and tiers, setting vault fees and jail status) and VaultHub operations (forcing validator exits, socializing bad debt, setting liability targets) through a VaultsAdapter intermediary. These motions execute automatically unless the minimum objection threshold is exceeded, reducing governance friction while maintaining DAO oversight through an objection-based voting model.

Security Considerations

The team considered risks and attack vectors that could potentially compromise the integrity, availability, and financial security of the Lido V3 staking infrastructure, with particular focus on the stVaults feature and its associated vault management operations. The stVaults architecture introduces new attack surfaces related to vault collateralization, cross-vault interactions, and the coordination between VaultHub, StakingVault contracts, and the broader core Lido protocol.

The analysis encompasses both technical vulnerabilities in vault accounting and state transitions, and broader systemic risks arising from the interaction between isolated vault positions and the unified stETH token. Specific attention was given to the core integration of stvaults, reserve,health ratio enforcement mechanisms, forced rebalancing and exit flows, cross-vault invariant maintenance, and the Predeposit Guarantee system, as these represent critical attack surfaces with high potential impact on vault operations and user funds.

Specifically, the team looked for risks regarding errors and malicious actors that could lead to incorrect vault accounting, excessive stETH minting without corresponding collateral, denial of service on critical vault operations, or the possibility to bypass security mitigations and protocol safeguards. While auditing, the following types of attackers were considered: external attackers targeting vault contracts directly, malicious vault owners attempting to extract value or evade obligations, malicious stakers seeking to exploit vault mechanics, malicious node operators manipulating validator operations or deposits, compromised DAO governance attempting unauthorized parameter changes, and coordinated attacks involving multiple compromised vaults.

Audit Goals

The primary goal of this security audit is to comprehensively assess the security posture of the Lido V3 on-chain smart contracts, with particular emphasis on the stVaults feature and its associated vault management operations. The audit aims to identify potential vulnerabilities, attack vectors, and security weaknesses that could compromise the integrity, availability, and financial security of the stVaults ecosystem.

1. Enumerate the attack surfaces related to newly introduced stVaults contracts and their integration points with existing Lido infrastructure.
2. Find specific risks and attack vectors which could realistically lead to incorrect vault accounting, protocol insolvency, denial of services or loss of user funds.
3. Suggest limited and accurate fixes for such risks while maintaining the architectural integrity of the stVaults design.

Key Focus Areas

The audit's scope encompasses the stVaults smart contract implementation and integration with the Lido Core protocol. Specifically, the team looked into the following areas of interest:

- stVaults Accounting and State Management : Ensure vaults maintain required collateralization levels and cannot mint excessive stETH.
- Reserve Ratio Enforcement and Collateralization : Ensure vaults maintain required collateralization levels and cannot mint excessive stETH.
- Cross-Vault Invariants and Protocol Solvency : Verify system-wide invariants hold across all vaults and the core pool.
- Predeposit Guarantee and Deposit Security : Validate the deposit frontrunning protection mechanism functions correctly.
- Triggerable Exits and Forced Rebalancing : Verify forced exit mechanisms work correctly and securely.
- Vault Economic Security and Fee Mechanisms : Evaluate the security of vault fee calculations and reward distributions.
- Access Control and Permissions : Validate role-based access control and authorization boundaries.
- Upgradeability and Initialization : Ensure safe upgrade paths and initialization procedures.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	7	7	7
High	14	14	14
Medium	29	29	25
Low	21	21	13
Informational	13	13	11
Total	84	84	70

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Detailed Findings

ID	Title	Severity	Status
C-01	Redemption settlement underflow can brick vault	Criticial	Fixed
C-02	Missing vault ownership check in compensation withdrawal	Critical	Fixed
C-03	Frontrun vault compensation to create bad debt and steal deposits	Critical	Fixed
C-04	Locked value not updated when tier changes reserve requirements	Critical	Fixed
C-05	Fee payment underflow inflates vault totalValue	Critical	Fixed
C-06	Reentrancy in _activateAndTopUpValidator allows unlocking all guarantor balance	Critical	Fixed
C-07	Cross-Vault Predeposit Manipulation Enables Inflated Total Value and Bad Debt Creation	Critical	Fixed
H-01	Ghost collateral creation via inOutDelta cache timing	High	Fixed

H-02	Vault draining via liability manipulation through mint/rebalance loop	High	Fixed
H-03	Self-induced vault unhealthiness via unfavorable rebalance rounding	High	Fixed
H-04	Type confusion mixing shares and assets in shortage calculation	High	Fixed
H-05	Premature locked and tier reset during disconnect cancellation	High	Fixed
H-06	Gifting ETH delays rebalancing permanently	High	Fixed
H-07	Locked value doesn't track liability after debt repayment between reports	High	Fixed
H-08	Redemptions permanently stuck due to double rounding	High	Fixed
H-09	Excessive fees on vault reconnection in same refSlot	High	Fixed
H-10	Decreasing total vault value using predeposit	High	Fixed
H-11	Make ETH collateral stuck in pending validator	High	Fixed
H-12	Double counting by disconnecting	High	Fixed

H-13	Vault Value Manipulation via Predeposit State Transitions	High	Fixed
H-14	Node Operator Fees Bypassed via Quarantine Manipulation	High	Fixed
M-01	Division by zero DOS when internal shares reach zero	Medium	Fixed
M-02	Excess withdrawal fees lost when refundRecipient is zero address	Medium	Fixed
M-03	Wrong order of external shares update corrupts rebase events	Medium	Fixed
M-04	StETH contract can receive external shares breaking transfer block	Medium	Fixed
M-05	Duplicate redemption updates in settlement flow	Medium	Fixed
M-06	Rebase smoothing calculations use total shares instead of internal	Medium	Fixed
M-07	Smoothing ignores bad debt internalization in rate calculation	Medium	Acknowledged
M-08	Fee recipients partially absorb bad debt in same report	Medium	Acknowledged
M-09	Vault permanently stuck when redemptions equal max shares during slashing	Medium	Fixed

M-10	Bad debt remains when PDG compensation available	Medium	Acknowledged
M-11	Vault data update possible for disconnected vaults in LazyOracle	Medium	Fixed
M-12	Repetitive vault reports bypass quarantine mechanism	Medium	Fixed
M-13	Quarantine bypass via pre-ordering deposits and withdrawals	Medium	Fixed
M-14	Shortfall rebalance insufficient for vault health	Medium	Fixed
M-15	Withdrawal bypass via gifting frontrun	Medium	Fixed
M-16	Redemption bypass via core deposit restoring mint capability	Medium	Fixed
M-17	Bad debt socialization prevented by pending disconnect	Medium	Fixed
M-18	Locked value drift due to share rate changes	Medium	Fixed
M-19	Staking limits bypass via external ether rebalancing	Medium	Acknowledged
M-20	Redemptions can increase due to rounding in settlement	Medium	Fixed

M-21	Group rebalancing frontrun prevents bad debt socialization	Medium	Fixed
M-22	Vault DOS via gifting enables withdrawal manipulation	Medium	Fixed
M-23	Share limit zero bypass via tier change	Medium	Fixed
M-24	Node Operator Can Block Vault Disconnection by Refusing Fee Disbursement	Medium	Fixed
M-25	Missing check that the owner of the stakingVault is msg.sender when connecting vault	Medium	Fixed
M-26	addFeeExemption with max_int128 causes permanent disconnect DOS through underflow	Medium	Fixed
M-27	Bypass isApprovedToConnect check through ownership transfer chain	Medium	Fixed
M-28	Bypass fees by rebalancing	Medium	Fixed
M-29	Block disconnection by setting malicious fee recipient	Medium	Fixed
L-01	Manual pause flag conflict with automatic pause in deposit control	Low	Fixed
L-02	Zero external shares despite non-zero vault liabilities due to	Low	Fixed

	rounding		
L-03	Inconsistent behavior when totalUnsettledObligations equals MIN_BEACON_DEPOSIT	Low	Fixed
L-04	Missing connection check allows operations on disconnected vaults	Low	Fixed
L-05	Quarantine penalizes legitimate rewards up to sanity limit	Low	Acknowledged
L-06	Cannot set redemptions to zero when liability is zero	Low	Fixed
L-07	Permanently stuck ETH via payable depositToBeaconChain	Low	Fixed
L-08	Default tier restoration bypass via disconnect/reconnect	Low	Acknowledged
L-09	Reverse limiting order suboptimal in sanity checker	Low	Acknowledged
L-10	If shares rate is less than 1 anyone can transfer shares from anyone else	Low	Acknowledged
L-11	Revert in applyVaultReport because of underflow	Low	Acknowledged
L-12	Should check pause deposits for a vault that receives badDebt	Low	Fixed
L-13	proveAndDeposit should not be	Low	Fixed

	payable		
L-14	Impossible to disconnect vault with fees exceeding vault value	Low	Fixed
L-15	Permanently Locked Node Operator Balance Blocks Guarantor Updates	Low	Fixed
L-16	Fee Updates Blocked by Reserve Ratio Check and Pending Disconnect State	Low	Fixed
L-17	Node Operator Fees Lost When Vault is Force Disconnected by DAO	Low	Acknowledged
L-18	Default tier params not validated during initialization	Low	Fixed
L-19	Self-Induce vault Unhealthiness via Unfavorable Rebalance Rounding	Low	Acknowledged
L-20	Solidity 0.4.24 allocates more memory than needed	Low	Acknowledged
L-21	Predeposit Frontrun compensation bypasses InOutDelta tracking and triggers quarantine	Low	Acknowledged
I-01	Missing array length validation in batch predeposit validation	Informational	Fixed

I-02	Division by Zero in get_steth_by_shares Function	Informational	Fixed
I-03	Missing return value breaks compensation amount tracking	Informational	Fixed
I-04	Outdated comment about non-connected vault tier changes	Informational	Fixed
I-05	Zero bad debt write-off incorrectly allowed	Informational	Fixed
I-06	Multiple gas optimization opportunities	Informational	Fixed
I-07	Unnecessary vault check in redemption validation	Informational	Fixed
I-08	The value of fee shares may be too low	Informational	Acknowledged
I-09	Redundant condition - one contains the other	Informational	Fixed
I-10	Some functions should be external not public	Informational	Fixed
I-11	Cannot set manual pause deposit flag to false if resume not allowed	Informational	Fixed
I-12	Burning External Shares Reverts When Current Stake Limit Equals uint96 Maximum	Informational	Fixed
I-13	Missing BLS Signature Length	Informational	Acknowledged

	Validation in Predeposit Flow		

Critical Severity Issues

C-01 - Redemption Settlement Underflow Can Brick Vault

Severity: **Critical**	Impact: **High**	Likelihood: **Medium**
Files: VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

VaultHub._settleObligations() lacks validation that valueToRebalance doesn't exceed vault totalValue. When inOutDelta's absolute magnitude exceeds reported totalValue, the settlement process causes integer underflow, permanently bricking the vault.

VaultHub.sol (_planRebalance around lines 1334-1336):

```python
uint256 cappedRedemptionsShares = Math256.min(_record.liabilityShares, redemptionShares);
sharesToRebalance = Math256.min(cappedRedemptionsShares, _getSharesByPooledEth(balance));

valueToRebalance = _getPooledEthBySharesRoundUp(sharesToRebalance);
unsettledRedemptions = _getPooledEthBySharesRoundUp(redemptionShares - sharesToRebalance);
// Missing check: valueToRebalance <= totalValue
```

Scenario:

1. Create vault and mint full stETH allocation against 32 ETH collateral
2. Set redemptions to maximum (entire liability)
3. Vault suffers slashing event creating bad debt
4. Oracle submits report with reduced totalValue
5. Attacker gifts ETH directly to vault contract
6. settleVaultObligations() called
7. valueToRebalance calculated based on gifted balance exceeds reported totalValue
8. Withdrawal at line 1412 executes for valueToRebalance amount
9. inOutDelta becomes negative with magnitude > totalValue

10. Line 1178: `uint256(int256(uint256(report.totalValue)) + _record.inOutDelta.value)` underflows
11. totalValue wraps to extremely large number, vault permanently bricked

The underflow prevents all future operations requiring valid totalValue calculations.

Recommendations : Cap redemptions settlement to vault's actual totalValue in _planRebalance to prevent processing more obligations than the vault can cover.

Lido's response: Fixed in PR #1350 by capping obligations settlement over totalValue.

Fix Review: Fixed. The settlement logic now limits valueToRebalance to not exceed totalValue, preventing underflow conditions when processing redemptions.

C-02 - Missing Vault Ownership Check in Compensation Withdrawal

Severity: **Critical**	Impact: **Medium**	Likelihood: **High**
Files: VaultHub.sol, PredepositGuarantee.sol	Status: Fixed	Violated Property: N/A

Description:

The VaultHub contract contained a critical access control flaw in the compensateDisprovenPredepositFromPDG() function that allowed any vault owner to steal predeposit compensation funds from any other vault. The vulnerability existed because the function validated that the caller owned *a* vault, but failed to verify that the validator pubkey being claimed actually belonged to that vault.

The attack flow exploited two layers of insufficient validation:

VaultHub.sol (line ~942-944):

```
None
function compensateDisprovenPredepositFromPDG( address _vault, bytes calldata _pubkey,
address _recipient ) external returns (uint256)
{
   _checkConnectionAndOwner(_vault);
   // Only checks caller owns THIS vault
   _predepositGuarantee().compensateDisprovenPredeposit(_pubkey, _recipient);
}
```

The `_checkConnectionAndOwner()` verified that `msg.sender` owned `_vault`, but crucially did not validate any relationship between `_vault` and `_pubkey`. An attacker could pass their own vault address while claiming compensation for a victim's validator pubkey.

PredepositGuarantee.sol (line 568):

```
None
if (msg.sender != stakingVault.owner()) revert NotStakingVaultOwner();
```

This check in PredepositGuarantee appeared protective but was ineffective because VaultHub is the canonical owner of all StakingVault contracts when connected. When VaultHub called this function, the check would always pass regardless of which vault initiated the original request.

Attack Scenario:

1. Attacker creates and connects their own malicious vault to VaultHub (permissionless)
2. Victim's vault has a disproven validator with 1 ETH locked in PredepositGuarantee
3. Attacker calls `VaultHub.compensateDisprovenPredepositFromPDG(attackerVault, victimPubkey, attackerAddress)`
4. Function passes `_checkConnectionAndOwner()` check (attacker owns attackerVault)
5. Compensation flows to attacker's chosen address instead of victim's vault
6. Attacker receives 1 ETH that should have been returned to victim's vault

Recommendations : Redirect the compensation back to the vault itself.

Lido's response: Fixed in #PR1336

Fix Review: Fixed. Lido acknowledged the critical severity and implemented the recommended permissionless approach via PR #1336 and #1337.

C-03 - Frontrunning Vault Compensation to Create Bad Debt

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: VaultHub.sol, PredepositGuarantee.sol	Status: Fixed	Violated Property: N/A

Description:

The VaultHub.compensateDisprovenPredepositFromPDG() function allowed vault owners to frontrun their own compensation claims after maxing out borrowing, enabling theft of predeposit guarantee funds while leaving behind bad debt. The attack exploited the arbitrary recipient parameter that permitted routing compensation away from the vault.

VaultHub.sol (line 944):

```python
function compensateDisprovenPredepositFromPDG(address _vault, bytes calldata _pubkey, address
_recipient ) external returns (uint256)
{
   _checkConnectionAndOwner(_vault);
   return _predepositGuarantee().compensateDisprovenPredeposit(_pubkey, _recipient);
}
```

Attack scenario:

1. Vault owner predeposits 1 ETH through PredepositGuarantee for a validator
2. Validator gets frontrun with incorrect withdrawal credentials
3. Owner mints maximum stETH shares against their vault's 32 ETH collateral
4. Owner immediately calls compensateDisprovenPredepositFromPDG() with an arbitrary recipient address under their control
5. 1 ETH compensation flows to the arbitrary recipient instead of repaying vault debt
6. Vault now has full liability shares minted but missing 1 ETH of actual collateral
7. This 1 ETH shortfall becomes bad debt that gets socialized to all Lido stakers

The issue enabled vault owners to extract compensation funds that should have restored vault health, deliberately creating undercollateralized positions where innocent stakers bear the loss.

Recommendations: Make compensation permissionless and enforce that funds route back to the affected vault. Compensation should automatically restore vault collateral.

Lido's response: Fixed in #PR1336

Fix Review: Fixed via PR #1336 making compensation permissionless and routing funds directly to vaults.

C-04 - Locked Value Not Updated on Tier Change

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: VaultHub.sol, OperatorGrid.sol	Status: Fixed	Violated Property: N/A

Description:

VaultHub.updateConnection() fails to recalculate the vault's locked value when OperatorGrid.changeTier() modifies reserve ratio parameters. This allows vault owners to withdraw funds that should remain locked, violating health requirements and enabling undercollateralization.

VaultHub.sol (lines 474-479):

```python
function updateConnection(address _vault, uint256 _shareLimit, uint256 _reserveRatioBP,
uint256 _forcedRebalanceThresholdBP, uint256 _infraFeeBP, uint256 _liquidityFeeBP, uint256
_reservationFeeBP ) external onlyRole(CHANGE_CONNECTION_ROLE)
{
   // ... validation checks ...
   connection.shareLimit = uint96(_shareLimit);
   connection.reserveRatioBP = uint16(_reserveRatioBP);

   // Updated
   connection.forcedRebalanceThresholdBP = uint16(_forcedRebalanceThresholdBP);
   // Updated
   connection.infraFeeBP = uint16(_infraFeeBP);
   connection.liquidityFeeBP = uint16(_liquidityFeeBP);
   connection.reservationFeeBP = uint16(_reservationFeeBP);

   // Missing: record.locked recalculation
}
```

Attack scenario:

1. Vault has totalValue=1000 ETH, liabilityShares worth 369 ETH, locked=451 ETH, reserveRatioBP=10%
2. Vault is healthy under current parameters
3. OperatorGrid.changeTier() increases reserveRatioBP to 63.1% and forcedRebalanceThresholdBP to 20%
4. locked value remains 451 ETH despite higher reserve requirements
5. Owner immediately calls withdraw() for 548 ETH (totalValue - locked = 549 ETH available)
6. Withdrawal succeeds because locked value wasn't updated
7. Vault now unhealthy with insufficient collateral for new reserve ratio

The invariant "vault's locked value covers liability + reserve ratio" breaks immediately after tier changes.

Recommendations : Recalculate locked value in updateConnection() using the same formula as applyVaultReport(): locked = max(liability * TOTAL_BASIS_POINTS / (TOTAL_BASIS_POINTS - reserveRatioBP), CONNECT_DEPOSIT).

Lido's response: Fixed in #PR1425

Fix Review: updateConnection() now recalculates locked value when reserve parameters change.

C-05 - Fee Payment Underflow Inflates Vault TotalValue

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

When vault fees exceed vault value, the fee settlement process lacks underflow protection, causing totalValue to wrap around to maximum uint256. This enables massive unbacked minting by artificially inflating the vault's reported collateral.

VaultHub.sol (lines 1360 and 1417):

```python
// Line 1360 - calculating value to transfer
valueToTransferToLido = Math256.min(_obligations.unsettledLidoFees, remainingBalance);

// Line 1417 - withdrawal without underflow check _withdraw(_vault, _record,
LIDO_LOCATOR.treasury(), valueToTransferToLido);

// Later in totalValue calculation - no check that fees don't exceed totalValue
// If totalValue < valueToTransferToLido, this underflows
uint256 calculatedValue = totalValue - feesTransferred; // UNDERFLOW
```

Attack scenario:

1. Vault accumulates fees exceeding its actual value through slashing
2. Owner initiates voluntary disconnect
3. Owner gifts ETH directly to vault contract (not through fund())
4. Gift bypasses inOutDelta tracking but adds to vault balance
5. settleLidoFees() called during disconnect
6. valueToTransferToLido calculated based on vault balance (including gift)
7. Withdrawal at line 1417 executes for amount exceeding totalValue
8. totalValue calculation: `totalValue - feesWithdrawn` underflows

9. totalValue wraps to 2^256-1 (maximum uint256)
10. Owner can now mint stETH shares against effectively infinite collateral

The inflated totalValue persists until next report, enabling extraction of protocol funds through unbacked minting.

Recommendations : Cap fee payments to actual vault totalValue before withdrawal. Add explicit check: feesToPay = Math256.min(feesToPay, totalValue) before any fee-related withdrawals.

Lido's response: Fixed in PR #1350. Fees settlement now caps transfer value to totalValue for connected vaults and to vault balance for disconnected vaults. Additional SafeCast operations added to totalValue calculations to prevent underflow.

Fix Review: Fixed. Fee settlement logic now enforces bounds checking to ensure fees cannot exceed available vault value, preventing underflow in totalValue calculations.

C-06 - Reentrancy in _activateAndTopUpValidator Drains Guarantor Balance

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: PredepositGuarantee.sol	Status: Fixed	Violated Property: N/A

Description:

PredepositGuarantee._activateAndTopUpValidator() contains a reentrancy vulnerability enabling attackers to drain all locked guarantor balance. The function unlocks guarantor balance through _proveWC() before validator.stage updates, allowing reentrant calls during the vault's depositFromStaged() callback.

PredepositGuarantee.sol :

```python
if (stage == ValidatorStage.PREDEPOSITED)
{
   _proveWC(_witnesses[i], vault, withdrawalCredentials, nodeOperator);
   // _proveWC decreases balance.locked immediately
   stage = ValidatorStage.PROVEN;
}

if (stage == ValidatorStage.PROVEN)
{
   validator.stage = ValidatorStage.ACTIVATED;
   // Stage update AFTER activation _activateAndTopUpValidator(...);
   // Calls vault.depositFromStaged() - external call before state update
}
```

The vulnerability chain:

1. _proveWC() decreases balance.locked before validator.stage changes
2. validator.stage remains PREDEPOSITED until after _activateAndTopUpValidator()

3. _activateAndTopUpValidator() calls _stakingVault.depositFromStaged()
4. Malicious vault reenters proveWCActivateAndTopUpValidators() during callback
5. Reentrancy succeeds because validator.stage still PREDEPOSITED
6. Each reentrant call executes _proveWC() again, decreasing balance.locked by 1 ETH
7. Continue until balance.locked reaches zero

Attack scenario:

1. Attacker creates multiple predeposits to legitimate vaults, increasing target NO's balance.locked to 100 ETH
2. Attacker creates malicious StakingVault with reentrant depositFromStaged()
3. Attacker makes valid predeposit to malicious vault
4. Attacker calls proveWCActivateAndTopUpValidators() on malicious vault
5. During _activateAndTopUpValidator(), malicious vault's callback reenters
6. Reentrancy succeeds 100 times because validator.stage unchanged
7. Each call decreases balance.locked by 1 ETH
8. After 100 reentrancies, balance.locked = 0
9. Node operator withdraws entire 100 ETH guarantee
10. Legitimate vaults never receive predeposit compensation

The same vulnerability exists in proveWCAndActivate() function.

Recommendations : Update validator.stage to ACTIVATED before external call to vault.depositFromStaged(). Follow checks-effects-interactions pattern: complete all state changes before external calls.

Lido's response: Fixed in #PR1467

Fix Review: Fixed. validator.stage now updates to ACTIVATED before calling _activateAndTopUpValidator(), preventing reentrancy by making the validator ineligible for repeated _proveWC() calls.

C-07 - Cross-Vault Predeposit Manipulation Enables Inflated Total Value and Bad Debt Creation

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: PredepositGuarantee.sol	Status: Fixed	Violated Property: N/A

Description:

An attacker can inflate a vault's total value by exploiting a mismatch between beacon chain validator withdrawal credentials and PDG validator status tracking. By predepositing the same validator to two different vaults' withdrawal credentials, the attacker can cause one vault to incorrectly count a predeposit intended for another vault, enabling borrowing against phantom collateral and creating bad debt.

Attack Flow:

1. Vault1 has 1 ETH, Vault2 has 32 ETH
2. Attacker predeposits Validator1 to beacon chain with Vault1's WC (externally, not through PDG)
3. Attacker calls PDG.predeposit() for same Validator1 but with Vault2's WC and stages 31 ETH
4. Oracle calculates Vault1's total value: sees Validator1 has Vault1's WC on beacon chain + is in PREDEPOSITED state → incorrectly adds 1 ETH
5. Vault1 can now borrow against inflated 2 ETH value (1 real + 1 phantom)
6. Attacker compensates Vault2 via proveInvalidValidatorWC()
7. Vault1's value drops to 1 ETH, creating bad debt on borrowed amount

The off-chain oracle queries beacon chain for validators matching a vault's WC and checks their PDG status, but doesn't verify that ValidatorStatus.stakingVault matches the vault being reported.

Recommendations : Modify the oracle calculation to verify that both the withdrawal credentials and the ValidatorStatus.stakingVault field match the vault being reported. The oracle should only count a validator towards a vault's total value if:

1. The validator's WC on beacon chain matches the vault's WC
2. The validator is in PREDEPOSITED/PROVEN/ACTIVATED stage
3. The ValidatorStatus.stakingVault field matches the vault being reported

Lido's response: Fixed.

Fix Review: Fixed in #PR1584.

High Severity Issues

H-01 - Ghost Collateral Creation via InOutDelta Timing Exploitation

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: VaultHub.sol, LazyOracle.sol	Status: Fixed	Violated Property: N/A

Description:

LazyOracle uses the current inOutDelta value instead of the cached refSlot value when validating vault reports. When a report created for refSlot1 arrives after refSlot2 starts, the oracle passes refSlot2's current inOutDelta rather than the cached refSlot1 value. An attacker connects a vault with 32 ETH at refSlot 99, withdraws 31 ETH at refSlot 100 (setting current inOutDelta to 1 ETH), while the oracle observes 32 ETH. When the report submits at refSlot 101, VaultHub calculates totalValue = 32 + (1 - 1) = 32 ETH despite only 1 ETH remaining.

```
Python
// LazyOracle.sol (around line 423)
function _handleSanityChecks(...) internal returns (uint256 totalValueWithoutQuarantine,
int256 inOutDeltaOnRefSlot)
{
   inOutDeltaOnRefSlot = record.inOutDelta.getValueForRefSlot(uint48(_reportRefSlot));
   // Issue: getValueForRefSlot returns refSlot2's value when report arrives late
   // Should return cached refSlot1 value when report was created
}
```

Recommendations : Implement a double RefSlot cache storing values for the two most recent refSlots to ensure correct resolution based on actual report context.

Lido's response: Fixed in PR #1216 implementing double inOutDelta cache that safely stores values for two most recent refSlots.

Fix Review: The fix adds DoubleRefSlotCache structure maintaining two cached values, allowing correct retrieval of inOutDelta for the actual refSlot when reports were created regardless of submission timing.

H-02 - Vault Draining via Liability Manipulation Through Mint/Rebalance Loop

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: VaultHub.sol, StakingVault.sol	Status: Fixed	Violated Property: N/A

Description:

An attacker can bypass locked funds limits by repeatedly calling mintShares() followed by rebalance(). Each iteration increases liabilityShares via minting, then rebalances to pay back shares which decreases liabilities. The decreased liabilities permit additional minting in the next cycle. This loop drains the vault's available ETH without paying fees or maintaining required slashing reserves.

Python

```
// VaultHub.sol
function mintShares(address _vault, address _recipient, uint256 _amountOfShares) external
whenResumed
{
  // Increases liabilityShares
   _increaseLiability(...);
   LIDO.mintExternalShares(_recipient, _amountOfShares);
}

function rebalance(address _vault, uint256 _shares) external whenResumed
{
   _rebalance(_vault, record, _shares);
}

function _rebalance(address _vault, VaultRecord storage _record, uint256 _shares) internal
{
   _decreaseLiability(_vault, _record, _shares); // Decreases liabilityShares
   _withdraw(_vault, _record, address(this), valueToRebalance);
   // Loop: decreased liabilities now allow more minting
}
```

Recommendations : Track cumulative rebalanced amounts within the same refSlot to enforce one-time settlement per report period, preventing liability cycling.

Lido's response: Fixed in PR #1354 implementing minimal reserve requirements that prevent draining vaults below required thresholds.

Fix Review: The fix introduces minimalReserve checks ensuring vaults maintain minimum ETH reserves (connection deposit + slashing reserve), blocking withdrawals that would breach this threshold.

H-03 - Self-Induced Vault Unhealthiness via Unfavorable Rebalance Rounding

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

Vault owners can intentionally degrade their health factor through small rebalances exploiting round-up behavior. When rebalancing 1 share worth 1.1 ETH, `_getPooledEthBySharesRoundUp()` withdraws 2 wei while decreasing liability by only 1 share. Repeating this operation degrades the vault's collateral-to-liability ratio, making a healthy vault unhealthy through systematic unfavorable rounding.

```python
// VaultHub.sol (around lines 1118-1123)
function _rebalance(address _vault, VaultRecord storage _record, uint256 _shares) internal
{
   uint256 valueToRebalance = _getPooledEthBySharesRoundUp(_shares);
   // Rounds up _decreaseLiability(_vault, _record, _shares);
   // Decreases shares exactly
   _withdraw(_vault, _record, address(this), valueToRebalance);
   // Withdraws rounded amount
   // Each call: withdraw 2 wei but decrease liability by 1 share
   // Repeated execution degrades vault health
}
```

Recommendations : Add health factor validation preventing healthy vaults from becoming unhealthy through rebalancing operations.

Lido's response: Fixed in #PR1287

Fix Review: Fixed.

H-04 - Type Confusion Mixing Shares and Assets in Shortage Calculation

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

The shortage calculation in _totalUnsettledObligations() mixes incompatible units by adding share-denominated values with asset-denominated values. The function calculates required = _totalUnsettledObligations(obligations) + _rebalanceShortfall(...), where obligations are in shares but are being compared/added with rebalance shortfall calculations that may use different units, leading to incorrect shortage determinations.

```python
// VaultHub.sol (around line 834)
uint256 required = _totalUnsettledObligations(obligations) + _rebalanceShortfall(connection,
record);
// Type confusion: mixing shares with assets in arithmetic
```

Recommendations : Convert redemptions calculations to use shares consistently throughout the shortage calculation logic.

Lido's response: Fixed in PR #1350 refactoring redemptions to calculate everything in shares rather than mixing asset and share values.

Fix Review: The fix standardizes all redemption-related calculations to use shares uniformly, eliminating the type confusion between shares and assets.

H-05 - Premature Locked and Tier Reset During Disconnect Cancellation

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: VaultHub.sol, OperatorGrid.sol	Status: Fixed	Violated Property: N/A

Description:

During _initiateDisconnect(), the vault's locked value and tier are immediately reset to zero before the disconnection completes. If slashing occurs after disconnection initiation but before finalization, the disconnect is canceled during the next report. However, the vault owner can withdraw the connection deposit before the report since locked value is already zero, bypassing slashing reserve requirements. Additionally, tier accounting becomes inconsistent after cancellation.

```python
// VaultHub.sol (around line 1018 in _initiateDisconnection)
_record.locked = 0;
// Unlock immediately // tier reset happens here as well
```

Attack flow:

- *1. voluntaryDisconnect() -> calls _initiateDisconnect, sets locked=0*
- *2. Validator gets slashed*
- *3. Owner withdraws connection deposit (allowed since locked=0)*
- *4. Report arrives, cancels disconnect due to slashing*
- *5. Slashing reserve not maintained, tier accounting broken*

Recommendations : Defer locked value and tier reset until the final report when disconnection is confirmed complete.

Lido's response: PR #1378 fix disconnection flow by deferring state resets until disconnect finalization.

Fix Review: The fix moves locked and tier reset logic to occur only after disconnect completion is confirmed in the final report, preventing premature withdrawals.

H-06 - Gifting ETH Delays Rebalancing Permanently

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: VaultHub.sol, StakingVault.sol	Status: Fixed	Violated Property: N/A

Description:

When an attacker gifts ETH directly to a vault (not through normal deposit flows), the vault's actual balance exceeds its recorded totalValue.
The _withdrawableValue() calculation returns type(uint256).max when balance exceeds reported value. During rebalance, _getSharesByPooledEth(vault.balance) calculates a shares amount based on inflated balance.
This causes valueToRebalance = _getPooledEthBySharesRoundUp(_shares) to exceed the vault's totalValue, triggering the check if (valueToRebalance > totalValue_) revert. The vault becomes permanently stuck unable to rebalance.

VaultHub.sol (around lines 1051-1054, 1154, 882)

```
Python
// function _rebalance(address _vault, VaultRecord storage _record, uint256 _shares) internal
{
    uint256 valueToRebalance = _getPooledEthBySharesRoundUp(_shares);
    // If valueToRebalance > totalValue due to gifted ETH inflating shares calculation
    if (valueToRebalance > totalValue_) revert RebalanceAmountExceedsTotalValue(totalValue_,
valueToRebalance);
}

function _maxRebalance(...) internal view returns (uint256) {
    return type(uint256).max; // Returns max when balance > totalValue
}
```

Recommendations : Cap rebalance calculations at the vault's recorded totalValue rather than allowing unlimited values based on gifted balance.

Lido's response: Fixed in #PR1287

Fix Review: Fixed.

H-07 - Locked Value Doesn't Track Liability After Debt Repayment Between Reports

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

When calculating required locked value during report application, the system uses `max(_record.liabilityShares, _reportLiabilityShares)` at line 1034. If debt is repaid between oracle observation and report submission (during the next refSlot), the vault locks against the lower current liability rather than the higher observed liability. The oracle observed high liability at refSlot N, but on-chain liability decreased due to repayment before the refSlot N report applies. This creates an undercollateralized position until the next report cycle.

```python
// VaultHub.sol (around line 1034)
uint256 liabilityShares_ = Math256.max(_record.liabilityShares, _reportLiabilityShares);
// Uses max of current and reported, but if debt repaid after oracle observation
// vault doesn't lock sufficient value for the originally observed liability
```

Recommendations : Track required locked value separately or ensure locked value accounts for maximum observed liability regardless of subsequent repayments.

Lido's response: PR #1378 fixed it by addressing disconnection-related issues including locked value tracking during state transitions.

Fix Review: The fix defers locked value resets and ensures proper tracking through disconnection flows, though full implementation details for this specific case remain in review.

H-08 - Redemptions Permanently Stuck Due to Double Rounding

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: VaultHub.sol, Lido.sol	Status: Fixed	Violated Property: N/A

Description:

Lines 1336-1337 perform two successive roundUp operations when calculating redemption shares: sharesToSettle = getSharesByPooledEthRoundUp(redemptionsToSettle) followed by redemptionsToSettle = getPooledEthBySharesRoundUp(...).
The double rounding inflates the required value by 1-2 wei. This causes the resume check at line 1401 to revert permanently. Combined with issue #1297 preventing new share minting when redemptions are high, vaults become permanently stuck unable to settle the final dust amount.

```python
// VaultHub.sol (around lines 1336-1337) : First roundUp
uint256 sharesToSettle = getSharesByPooledEthRoundUp(redemptionsToSettle);
// Second roundUp
valueToRebalance = _getPooledEthBySharesRoundUp(sharesToSettle);
// Double rounding creates 1-2 wei excess, permanently blocking settlement
```

Recommendations : Use `roundDown` on the final calculation or implement dust amount tolerance to handle inevitable 1 wei discrepancies.

Lido's response: Fixed in PR #1350

Fix Review: Fixed.

H-09 - Excessive Fees on Vault Reconnection in Same RefSlot

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: VaultHub.sol, OperatorGrid.sol	Status: Fixed	Violated Property: N/A

Description:

When reconnecting a disconnected vault within the same refSlot, line 1279 calculates fees using zero storage values but non-zero `cumulativeLidoFees` from the oracle. The fee delta calculation becomes `cumulativeLidoFeesFromOracle - 0 = entire cumulative fees`. The vault is incorrectly charged all historical fees again upon reconnection, as the system doesn't account for fees already paid before disconnection.

VaultHub.sol (line 1279)

```python
cumulativeLidoFees (from oracle) - settledLidoFees (reset to 0)
// On reconnection in same refSlot: uses non-zero cumulative but zero settled
// Results in charging all historical fees again
```

Recommendations : Reset oracle cumulative fees on disconnection or maintain last disconnection state to calculate correct fee delta on reconnection.

Lido's response: Fixed in #PR1377.

Fix Review: Fixed.

H-10 - Decreasing Total Vault Value Using Predeposit

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: StakingVault.sol, PredepositGuarantee.sol, VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

Multi-step attack manipulates vault totalValue through predeposit state transitions:

(1) Connect vault with 32 ETH
(2) Predeposit validator1 with 1 ETH leaving 31 staged
(3) Disconnect vault
(4) Prove validator1 but don't activate (PROVEN state)
(5) Zero the stage
(6) Deposit 1 ETH and reconnect
(7) Predeposit validator2
(8) Activate validator1 increasing totalValue to 33 ETH
(9) Borrow against inflated 33 ETH collateral
(10) Prove validator2 which decreases totalValue back to 32 ETH.

The borrowed shares remain backed by reduced collateral, creating stuck debt.

Recommendations : Add ACTIVATED state requirement and only allow topup for validators not in PROVEN state to prevent manipulation via incomplete state transitions.

Lido's response: Fixed in #PR1467 by having an explicit activation flow for PDG.

Fix Review: Fixed.

H-11 - Make ETH Collateral Stuck in Pending Validator

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: StakingVault.sol, PredepositGuarantee.sol, VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

Complex attack path traps collateral through validator state manipulation:

- (1) Connect vault with 32 ETH
- (2) Predeposit 1 ETH
- (3) Disconnect
- (4) Change depositor
- (5) Prove but don't activate validator entering PROVEN state
- (6) Zero the stage
- (7) Reconnect vault
- (8) Mint shares creating debt
- (9) Topup existing validators with 30 ETH.

Result: vault totalValue becomes 0 but debt exists. After bad debt internalization, disconnecting and topping up to activate permanently traps the collateral in a pending validator that can't be recovered.

Recommendations : Add ACTIVATED state tracking and restrict topup operations to exclude validators in PROVEN state.

Lido's response: Fixed in #PR1467 by having explicit activation flow for PDG.

Fix Review: Fixed.

H-12 - Double Counting by Disconnecting

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: StakingVault.sol, PredepositGuarantee.sol, VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

When validators are activated through paths not involving PredepositGuarantee (PDG), the pending predeposit amount gets double-counted in accounting. The validator's 1 ETH predeposit is counted both as part of the vault's pending amount and again when activated outside the PDG flow. This creates a 1 ETH accounting discrepancy per affected validator, inflating the vault's apparent totalValue beyond actual collateral.

Recommendations : Ensure proper accounting reconciliation when validators activate through non-PDG paths, deducting pending amounts from totalValue calculations.

Lido's response: Fixed in #PR1512 by improving activation flow for PDG.

Fix Review: Fixed.

H-13 - Vault Value Manipulation via Predeposit State Transitions

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: StakingVault.sol, PredepositGuarantee.sol, VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

When validators are activated through paths not involving PredepositGuarantee (PDG), the pending predeposit amount gets double-counted in accounting. The validator's 1 ETH predeposit is counted both as part of the vault's pending amount and again when activated outside the PDG flow. This creates a 1 ETH accounting discrepancy per affected validator, inflating the vault's apparent totalValue beyond actual collateral.

VaultHub.sol (line 1279)

```python
// PredepositGuarantee.sol
function proveWCActivateAndTopUpValidators(
    ValidatorWitness[] calldata _witnesses,
    uint256[] calldata _amounts
) external whenResumed {
    // ...
    if (stage == ValidatorStage.PROVEN) {
        validator.stage = ValidatorStage.ACTIVATED;
        _activateAndTopUpValidator(..., _amounts[i], ...);
        // _amounts[i] can be 0, activating with only predeposited 1 ETH
        // Validator counts toward totalValue despite incomplete funding
    }
}
```

Attack Path:

1. Connect vault (32 ETH)
2. Predeposit validator1: 1 ETH predeposited, 31 ETH staged (totalValue = 32 ETH)
3. Disconnect vault

4. Prove validator1 without activation (PROVEN state)
5. Zero staged balance as depositor
6. Deposit 1 ETH, reconnect vault (totalValue = 32 ETH)
7. Predeposit validator2: 31 ETH staged
8. Activate validator1 with 0 top-up (totalValue inflates to 33 ETH)
9. Borrow against 33 ETH collateral
10. Prove validator2 (totalValue drops to 32 ETH)
11. Borrowed shares now backed by reduced collateral

Recommendations : Introduce ACTIVATED validator state and restrict top-up operations to exclude validators in PROVEN state. Prevent activation without sufficient funding to reach the 32 ETH threshold.

Lido's response: Fixed in #PR1482 by creating a better activation flow for PDG.

Fix Review: The fix adds ACTIVATED state tracking and enforces that validators can only be topped up after full activation. This prevents the state manipulation where partially funded validators temporarily inflate totalValue, closing the gap that allowed borrowing against phantom collateral.

H-14 - Node Operator Fees Bypassed via Quarantine Manipulation

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: NodeOperatorFee.sol,LazyOracle.sol, Off-Chain	Status: Fixed	Violated Property: N/A

Description:

A vault owner could manipulate the quarantine mechanism to avoid paying node operator fees on vault growth. By gifting funds to increase total value, the excess would enter quarantine. Since node operator fees were calculated only on the reported totalValue (excluding quarantined amounts), the owner could disconnect and leave without paying fees on the quarantined growth. The owner could repeatedly exploit this by cycling funds in and out of quarantine.

VaultHub.sol (line 1279)

```python
function _calculateFee() internal view returns (uint256 fee, int256 growth, uint256
abnormallyHighFeeThreshold) {
   VaultHub.Report memory report = latestReport();
   // quarantine value was NOT included
   growth = int256(uint256(report.totalValue)) - report.inOutDelta;
   int256 unsettledGrowth = growth - settledGrowth;
   if (unsettledGrowth > 0) {
      fee = (uint256(unsettledGrowth) * feeRate) / TOTAL_BASIS_POINTS; }
}
```

Recommendations : Include quarantined value when calculating node operator fees to ensure all vault growth is subject to fees regardless of quarantine status.

Lido's response: Fixed in #PR1541. Charge node operator fees over quarantined value also (like Lido fees).

Fix Review: The fix has been properly implemented in the _calculateFee() function. Quarantined value is now retrieved from LazyOracle and added to the reported totalValue before calculating growth and fees. This ensures node operator fees are charged on the complete vault value including any amounts held in quarantine, effectively closing the fee bypass vulnerability.

Medium Severity Issues

M-01 - Division by zero DOS when internal shares reach zero

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: Lido.sol, VaultHub.sol, WithdrawalQueue.sol	Status: Fixed	Violated Property: N/A

Description:

External shares can be withdrawn via the withdrawal queue, enabling an attack where all internal shares are withdrawn. Since `internalShares = totalShares - externalShares`, withdrawing all internal shares causes `internalShares = 0`. This triggers division by zero across all share-to-ETH conversion calculations, causing protocol-wide DOS.

In Lido.sol

```python
return (externalShares * _internalEther) / internalShares;
```

The vulnerability affects `getPooledEthByShares()` and related functions. Attackers can mint external shares through stVaults, then request withdrawal of all internal shares. Even blocking external mint count won't prevent this - any external shares beyond the initial 0xdead mint enables the attack.

Recommendations: Implement a simulatedShareRate in oracle reports to handle edge cases. The withdrawal queue must prevent internal shares from reaching zero, or the protocol must maintain the last valid ratio for recovery.

Lido's response: Fixed in PR #1404. The fix introduces a simulatedShareRate parameter in oracle reports to handle scenarios where internal shares approach zero, preventing division by zero while maintaining accurate share rate calculations.

Fix Review: Fixed. Oracle reports now include simulatedShareRate as a safety mechanism. The withdrawal queue uses this rate when processing withdrawals, ensuring internal shares never reach zero during finalization operations.

M-02 - Excess withdrawal fees lost when refundRecipient is zero address

Severity: **Medium**	Impact: **Medium**	Likelihood: **Low**
Files: WithdrawalQueue.sol	Status: Fixed	Violated Property: N/A

Description:

When calling `VaultHub.triggerValidatorWithdrawals(_refundRecipient = address(0))` with excess `msg.value`, StakingVault defaults the refund recipient to `msg.sender` (VaultHub) instead of the original user. The root cause is that StakingVault doesn't know the original user's address - it only sees VaultHub as the caller. User overpayments accumulate permanently in the VaultHub contract with no recovery mechanism.

Recommendations: In VaultHub, check if `_refundRecipient == address(0)` and set it to `msg.sender` before forwarding the call to StakingVault. This ensures excess fees are refunded to the actual user initiating the transaction.

Lido's response: Fixed in PR#1403. VaultHub now handles the zero address case by substituting msg.sender as the refund recipient before forwarding calls to StakingVault.

Fix Review: Fixed. VaultHub properly sets the refund recipient to the transaction sender when zero address is provided, ensuring users receive their excess withdrawal fee refunds.

M-03 - Wrong order of external shares update corrupts rebase events

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: Lido.sol, VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

In `_burnExternalShares()`, external shares are decreased BEFORE calculating pre/post rebase token amounts. This causes internal shares to inflate artificially during the calculation, corrupting emitted rebase events.

Lido.sol

```python
_setExternalShares(externalShares - _amountOfShares);
```

Example: totalShares=2000, externalShares=1500, internalShares=500, totalAssets=3000, ratio 1:6.

Burning 1000 external shares first decreases externalShares to 500, making internalShares appear as 1500. This changes the ratio from 1:6 to 1:2. Emitted events show preBalance=2000, postBalance=6000 (incorrect values).

Recommendations: Calculate token balances before modifying externalShares storage, then emit events with the correct pre/post balance values. Follow the proper sequence: capture state, modify state, emit events.

Lido's response: Fixed in PR #1403.

Fix Review: Fixed.

M-04 - StETH contract can receive external shares breaking transfer block

Severity: **Medium**	Impact: **Medium**	Likelihood: **Low**
Files: StETH.sol, Lido.sol	Status: Fixed	Violated Property: N/A

Description:

Direct mintExternalShares(StETH_ADDRESS) bypasses transfer checks. While StETH._transfer() blocks transfers to self and Lido blocks minting to StETH via regular paths, the new external mint path in Lido.mintExternalShares() lacks recipient validation.

Lido.sol

```python
function mintExternalShares(address _recipient, uint256 _amountOfShares) external
{
   require(_amountOfShares != 0, "MINT_ZERO_AMOUNT_OF_SHARES");
   _auth(_vaultHub()); _whenNotStopped();
   _mintShares(_recipient, _amountOfShares);
```

Shares sent to the StETH contract become permanently locked, breaking accounting assumptions since the contract cannot interact with itself.

Recommendations: Add `require(_recipient != STETH)` check in external mint validation to prevent minting shares to the StETH contract address.

Lido's response: Fixed in PR #1403. Recipient validation added to prevent minting external shares to the StETH contract.

Fix Review: Fixed. mintExternalShares() now includes a check preventing the StETH contract from being the recipient, maintaining the transfer block invariant.

M-05 - Duplicate redemption updates in settlement flow

Severity: **Medium**	Impact: **Low**	Likelihood: **Medium**
Files: VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

In the `_settleObligations()` execution path, redemptions are updated twice: once in `_decreaseRedemptions()` and again in `_updateRedemptions()`. This double update can cause incorrect redemption values in the second update, leading to accounting inconsistencies.

Recommendations: Consolidate to a single redemption update point in the settlement flow. Remove the duplicate call to ensure redemptions are only modified once per settlement operation.

Lido's response: Fixed in PR #1350. The redemptions refactoring consolidates all redemption updates to a single call point, eliminating the duplicate update pattern.

Fix Review: Fixed. Settlement flow now performs a single redemption update, removing the possibility of double-counting or incorrect second updates.

M-06 - Rebase smoothing calculations use total shares instead of internal

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: Lido.sol, Accounting.sol	Status: Fixed	Violated Property: N/A

Description:

The positive rebase limiter in Accounting._handlePositiveTokenRebase() uses totalShares and totalPooledEther instead of internalShares and internalEther. Using total values makes the actual increase/decrease appear smaller than reality, allowing the protocol to burn more shares than the smoothing limit should permit. This causes the rate to increase faster than intended by the positive rebase cap.

Accounting.sol

```python
_pre.totalPooledEther,
_pre.totalShares,
```

Recommendations : Change calculations to use `_getInternalPooledEther()` and `internalShares` for accurate smoothing enforcement. This ensures the limiter operates on the correct share subset.

Lido's response: Fixed in PR #1404. Smoothing calculations now use internal shares and internal ether values.

Fix Review: Fixed. Rebase smoothing now correctly calculates limits based on internal shares and internal ether, ensuring the positive rebase cap functions as designed.

M-07 - Smoothing ignores bad debt internalization in rate calculation

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: Lido.sol, Accounting.sol	Status: Acknowledged	Violated Property: N/A

Description:

Positive rebase smoothing doesn't account for `badDebtToInternalize`, preventing the protocol from reaching the target positive rebase rate even with excess yield. Bad debt must be internalized first (reducing the rate), only then can smoothing apply. This delays reaching the intended rebase rate, harming user rewards. Users cannot benefit from excess rewards during bad debt internalization periods.

Recommendations: Factor internalized bad debt into the smoothing calculation, or apply smoothing after bad debt settlement. This ensures the rebase limiter accounts for all rate-affecting operations.

Lido's response: Acknowledged.

Fix Review: Acknowledged.

M-08 - Fee recipients partially absorb bad debt in same report

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: VaultHub.sol, OperatorGrid.sol	Status: Acknowledged	Violated Property: N/A

Description:

Fees are calculated before bad debt internalization occurs. The share rate used for fee calculation excludes the internalized bad debt impact, meaning fee recipients' shares are valued higher than they should be post-internalization. Effectively, fees subsidize bad debt losses. The comment at line 263 in Accounting.sol implies fees shouldn't bear bad debt costs, but the implementation contradicts this intention.

Accounting.sol

```python
(update.sharesToMintAsFees, update.feeDistribution) = _calculateProtocolFees(...)
```

Recommendations: Calculate fees after bad debt is internalized to use the correct post-internalization share rate. This ensures fee recipients don't inadvertently absorb bad debt costs.

Lido's response: Acknowledged.

Fix Review: Acknowledged by Lido's team.

M-09 - Vault permanently stuck when redemptions equal max shares during slashing

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: VaultHub.sol, StakingVault.sol	Status: Fixed	Violated Property: N/A

Description:

When redemptions are set to maximum (entire liabilityShares converted to ETH value) and Lido core suffers slashing (share rate drops), the vault owner cannot fully settle redemptions. As shares are burned during settlement, the redemption ETH value remains constant but the share value decreases. This causes the vault to run out of liability shares before redemptions reach zero, leaving the vault permanently stuck - unable to disconnect or clear its obligations.

Recommendations: Cap redemptions in ETH to prevent this scenario, or allow partial settlement with zero liability. Ensure redemption mechanics account for share rate changes during the settlement period.

Lido's response: Fixed in PR #1350. Redemption mechanics now handle share rate changes properly, preventing vaults from becoming stuck due to slashing events.

Fix Review: Fixed. Vaults can now settle redemptions even when share rates decline, with proper handling of edge cases where liability reaches zero before full settlement.

M-10 - Bad debt remains when PDG compensation available

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: VaultHub.sol, PredepositGuarantee.sol	Status: Acknowledged.	Violated Property: N/A

Description:

When a predeposit frontrun causes vault bad debt, the protocol allows `internalizeBadDebt()` to execute even though the PredepositGuarantee (PDG) has funds available to compensate. The protocol should force PDG compensation first before internalizing debt to Lido stakeholders. Bad debt gets socialized to innocent parties when funds exist to make the vault whole.

Recommendations: Block `internalizeBadDebt()` when PDG has pending compensation available for the vault. Force the compensation claim first to ensure available funds are used before socializing debt across the protocol.

Lido's response: The internalization is a manual process which starts with a governance decision and the compensation availability will be checked before on the stage when forced rebalance is applied as it's not an automatic process but requires governance decision.

Fix Review: Acknowledged.

M-11 - Vault data update possible for disconnected vaults in LazyOracle

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: LazyOracle.sol, VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

`LazyOracle.updateVaultData()` lacks a connection check, allowing updates to be processed for disconnected vaults. This corrupts accounting by modifying state that should be frozen post-disconnection, potentially leading to ghost liabilities or rewards attributed to inactive vaults.

LazyOracle.sol

```python
function updateVaultData( address _vault, uint256 _totalValue, uint256 _cumulativeLidoFees,
uint256 _liabilityShares, uint256 _maxLiabilityShares, uint256 _slashingReserve, bytes32[]
calldata _proof ) external
{
   // Missing connection check
   bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(...))));
   if (!MerkleProof.verify(_proof, _storage().vaultsDataTreeRoot, leaf)) revert
InvalidProof();

   _vaultHub().applyVaultReport(...);
}
```

Recommendations: Add a connection validation to prevent updates on disconnected vaults.

Lido's response: Fixed in PR #1403. Connection validation added to prevent updates on disconnected vaults.

Fix Review: Fixed. LazyOracle now verifies vault connection status before processing any vault data updates, preventing state modifications on disconnected vaults.

M-12 - Repetitive vault reports bypass quarantine mechanism

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: LazyOracle.sol, VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

`_applyVaultReport()` is missing a check to verify whether a report has already been applied for a given timestamp/refSlot. This allows the same report to be submitted multiple times through `LazyOracle.updateVaultData()`, which breaks the quarantine mechanism, allows replaying stale data, and disrupts accounting.

Recommendations: Add a check if (currentTimestamp >= _timestamp) revert ReportTooOld(); and update $.report.timestamp = _timestamp; to prevent duplicate submissions.

Lido's response: Fixed in PR #1404. Report timestamp tracking prevents duplicate submissions for the same refSlot.

Fix Review: Fixed. VaultHub now maintains report timestamps per vault and rejects submissions that attempt to replay already-applied reports.

M-13 - Quarantine bypass via pre-ordering deposits and withdrawals

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: LazyOracle.sol, VaultHub.sol, StakingVault.sol	Status: Fixed	Violated Property: N/A

Description:

If a vault gifts high-value ETH triggering quarantine, then immediately withdraws the gifted amount after quarantine becomes active, the quarantine persists but the delta is zeroed on-chain. When the next gift occurs, it doesn't trigger `_processTotalValue()` to release the quarantine, allowing unlimited deposits without quarantine using the pre-established delta.

Recommendations: Ensure quarantine state properly resets when on-chain totalValue returns to baseline. Track actual value changes rather than just deltas to prevent quarantine bypass through withdrawal manipulation.

Lido's response: Quarantine mechanism now properly handles withdrawal sequences and prevents bypass through pre-ordering.

Fix Review: Fixed. Quarantine processing now correctly handles value decreases and prevents the bypass by tracking actual totalValue changes rather than just relying on deltas.

M-14 - Shortfall rebalance insufficient for vault health

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

`VaultHub.rebalanceShortfall()` may leave a vault unhealthy due to rounding, especially when `reserveRatioBP == forcedRebalanceThresholdBP`.

Example: Vault has 1000 ETH totalValue, liability of 689 shares (801 ETH at current rate), RR=20%, and shortfall calculates as 5 shares. After rebalancing 5 shares (6 ETH worth), vault totalValue becomes 994 ETH, liability becomes 796 ETH, but threshold remains at 795 ETH. The vault is still unhealthy and requires multiple rebalance calls to achieve health.

Recommendations: Calculate the exact shortfall including a rounding buffer to ensure a single rebalance call is sufficient to restore vault health. Add `Math.ceilDiv()` for share-to-ETH conversions in shortfall calculations.

Lido's response: Fixed

Fix Review: Fixed in PR1549.

M-15 - Withdrawal bypass via gifting frontrun

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: StakingVault.sol, VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

The check at lines 1476-1479 validates `totalValue >= lockedPlusUnsettled`, but a vault owner can gift/fund the vault, call withdraw, then receive the gifted amount back.

By frontrunning `forceValidatorExit()` with vault funding to make it appear healthy, the owner can withdraw, then unfund post-settlement. This allows indefinite avoidance of redemption payments by timing funding with withdrawals.

Recommendations: Check against actual vault balance rather than reported totalValue, or lock withdrawals when unsettled obligations exist. Implement a time-lock on withdrawals following deposits to prevent atomicity.

Lido's response: Fixed in PR #1350. Withdrawal validation now properly accounts for actual vault balance and prevents manipulation through temporary funding.

Fix Review: Fixed. Vault withdrawal checks now use balance-based validation that cannot be manipulated through short-term gifting, and withdrawal restrictions apply when obligations are pending.

M-16 - Redemption bypass via core deposit restoring mint capability

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: VaultHub.sol, Lido.sol	Status: Fixed	Violated Property: N/A

Description:

Redemptions force vault repayment when the internal:external share ratio is off. However, this can be bypassed by:

(1) Depositing ETH to Lido core, temporarily fixing the ratio

(2) Minting new liability and repaying to zero redemptions

(3) Withdrawing through the queue.

The new deposit relieves pressure, but the attacker adds equal withdrawal requests maintaining net pressure. Redemptions are zeroed so repayment can't be forced despite unchanged net pressure.

Recommendations: Consider net withdrawal pressure (pending queue requests) rather than just the current ratio when setting redemptions. Track both deposits and pending withdrawals to calculate actual system pressure.

Lido's response: Fixed in PR #1350. Redemption mechanism now accounts for net withdrawal pressure including pending queue requests.

Fix Review: Fixed.

M-17 - Bad debt socialization prevented by pending disconnect

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: VaultHub.sol,	Status: Fixed	Violated Property: N/A

Description:

Before `_initiateDisconnect()` sets `pending=true`, both liability is verified as zero and locked becomes zero.

However, `socializeBadDebt()` checks `!pending && locked != 0`, preventing socialization to pending vaults even when appropriate. A vault owner can disconnect before bad debt is socialized, avoiding their share of protocol losses.

Recommendations: Allow bad debt socialization to pending vaults, or check for pending disconnect before allowing disconnect initiation. Ensure bad debt is handled before state transitions.

Lido's response: Fixed in PR #1350. Bad debt socialization logic now properly handles pending disconnect states and ensures debt is socialized before allowing disconnection.

Fix Review: Fixed. Protocol now socializes bad debt before marking vaults as pending disconnect, preventing owners from escaping their proportional share of losses.

M-18 - Locked value drift due to share rate changes

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: VaultHub.sol, Lido.sol	Status: Fixed	Violated Property: N/A

Description:

Vault locked value is calculated once, but the Lido internal share : ETH ratio constantly changes.

Between reports, locked value may not cover `liabilityShares * ratio + reserveRatio` due to ratio drift. This creates temporary undercollateralization between reports, where vault health depends on when the report is submitted.

Recommendations: Recalculate locked value on-demand using the current ratio, or acknowledge this as an accepted risk mitigated by report frequency. Implement continuous locked value adjustments based on real-time share rates.

Lido's response: Locked value calculations now account for share rate changes through more frequent updates and dynamic adjustments.

Fix Review: Fixed.

M-19 - Staking limits bypass via external ether rebalancing

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: Lido.sol, VaultHub.sol	Status: Acknowledged	Violated Property: N/A

Description:

`Lido.rebalanceExternalEtherToInternal()` completely ignores staking limit mechanisms. This allows unlimited deposits via external rebalancing, circumventing STAKING_CONTROL and rate limiting. This breaks a fundamental protocol safeguard against excessive deposits.

Lido.sol

```python
function rebalanceExternalEtherToInternal(uint256 _amountOfShares) external
{
    _auth(_vaultHub()); _whenNotStopped();
    // Missing staking limit checks
    uint256 etherAmount = getPooledEthByShares(_amountOfShares);
    _burnShares(msg.sender, _amountOfShares);
    _setExternalShares(getExternalShares() - _amountOfShares);
    _submit(address(0));
}
```

Recommendations: Apply staking limit checks in the rebalance function, or exempt only amounts up to the current available limit. Ensure rebalancing operations respect global deposit limits.

Lido's response: Rebalancing bypasses the limits, but minting and burning external shares now takes the limits into account. Rebalancing does not affect the total issuance of a token (just internalizes shares) so it is safe from Lido perspective. It may also affect a simulatedShareRate check in OracleReportSanityChecker, but the parameter can be modified to avoid any interruption.

Fix Review: Acceptable on Certora side as rebalancing just internalizes shares.

M-20 - Redemptions can increase due to rounding in settlement

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

In VaultHub's rebalancing mechanism, share-to-ETH conversion rounding causes redemption values to unintentionally increase during settlement operations. The issue occurs in the rebalancing calculation flow where shares are converted to ETH and back.

VaultHub.sol (lines 1324-1338):

```python
function _planRebalance( address _vault, VaultRecord storage _record, VaultObligations
storage _obligations ) internal view returns (uint256 valueToRebalance, uint256
sharesToRebalance)
{
   uint256 redemptionShares = _getSharesByPooledEth(_obligations.redemptions);
   uint256 maxRedemptionsValue = _getPooledEthBySharesRoundUp(redemptionShares);
   if (maxRedemptionsValue < _obligations.redemptions) redemptionShares += 1;

   uint256 cappedRedemptionsShares =
      Math256.min(_record.liabilityShares, redemptionShares);
   sharesToRebalance = Math256.min(cappedRedemptionsShares, _getSharesByPooledEth(balance));
   valueToRebalance = _getPooledEthBySharesRoundUp(sharesToRebalance);

}
```

The vulnerability chain:

1. `redemptionShares` calculated by converting ETH to shares (rounds down)
2. `maxRedemptionsValue` converts back to ETH using `_getPooledEthBySharesRoundUp` (rounds up)
3. If `maxRedemptionsValue < _obligations.redemptions`, an extra share is added
4. This process can cause small incremental increases in redemption values

5. Affected functions: `applyVaultReport()`, `resumeBeaconChainDeposits()`, and `settleVaultObligations()`

The rounding errors accumulate over multiple reports, gradually inflating redemption obligations beyond what should be settable only via `setVaultRedemptions()`.

Recommendations: Use consistent rounding direction (round down for both conversions) or explicitly cap redemption increases to prevent inflation outside of `setVaultRedemptions()`.

Lido's response: Fixed in PR #1350. The fix refactored the redemption calculation logic to use consistent rounding and prevent unintended increases.

Fix Review: The refactoring ensures redemptions can only increase through explicit setVaultRedemptions() calls by eliminating the rounding inconsistency in the conversion chain. The new implementation maintains redemption invariants across all settlement operations.

M-21 - Group rebalancing frontrun prevents bad debt socialization

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: VaultHub.sol, OperatorGrid.sol	Status: Fixed	Violated Property: N/A

Description:

Vault owners can frontrun `socializeBadDebt()` transactions by moving their vault to a different group with insufficient capacity, effectively preventing bad debt assignment to their vault.

VaultHub.sol (line 568):

```python
function socializeBadDebt( address _vault, address[] calldata _acceptorVaults ) external
{
   // Bad debt socialization attempts to distribute losses
   // across multiple vaults in the same group
   // ...
   _shareLimit: _getSharesByPooledEth(recordAcceptor.locked)

}
```

Attack scenario:

1. Protocol initiates bad debt socialization for an unhealthy vault
2. Attacker monitors mempool and detects incoming socializeBadDebt() call
3. Attacker frontruns by calling updateConnection() via OperatorGrid
4. Vault moved to a group that has no available share limit capacity
5. socializeBadDebt() transaction reverts due to capacity constraints
6. Attacker's vault avoids taking its share of protocol losses

This allows malicious actors to tactically reorganize group membership to dodge bad debt obligations, undermining the protocol's loss socialization mechanism.

Recommendations: Lock group membership changes during bad debt socialization periods, or reserve emergency capacity in all groups to ensure socialization can proceed regardless of group arrangements.

Lido's response: Fixed in PR #1395 (Bad Debt refactoring). The fix restructures the bad debt socialization mechanism to prevent frontrunning through group changes.

Fix Review: Fixed.

M-22 - Vault DOS via gifting enables withdrawal manipulation

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: VaultHub.sol, StakingVault.sol	Status: Fixed	Violated Property: N/A

Description:

Withdrawing more than the vault's actual value through gifts during slashing causes LazyOracle to revert on the next report, creating a denial-of-service condition.

LazyOracle.sol (line 311):

```python
function _handleSanityChecks( address _vault, uint256 _reportedTotalValue, // ... ) internal
{
   // Reverts if totalValue calculation underflows
   // when withdrawals exceed actual vault balance
}
```

Attack scenario:

1. Attacker gifts ETH directly to vault (increasing balance)
2. Vault experiences slashing event
3. Owner withdraws amount based on inflated balance (gift + actual)
4. Actual vault value now less than withdrawn amount
5. Next oracle report attempts to calculate totalValue
6. Calculation underflows at line 311, causing revert
7. Vault stuck in DOS state until owner manually funds the difference

The vulnerability allows attackers to grief vaults by gifting small amounts of ETH before slashing events, then the vault owner unknowingly overwithdraw, bricking future reports until additional capital is injected.

Recommendations: Limit withdrawals to min(requested, vaultBalance) or handle underflows gracefully by detecting and preventing overdraft conditions.

Lido's response: Fixed in #PR1451 : Lido team implemented withdrawal limits that consider actual vault balance rather than inflated totalValue calculations that include gifts.

Fix Review: Withdrawals are now properly bounded by the actual balance available in the vault, preventing the underflow scenario that caused report DOS. The fix ensures gifts cannot be used to manipulate withdrawal limits.

M-23 - Share limit zero bypass via tier change

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: VaultHub.sol, OperatorGrid.sol	Status: Fixed	Violated Property: N/A

Description:

Vault owners can bypass a zero share limit restriction by changing tiers, which resets the limit to the new tier's default value, circumventing owner-imposed restrictions.

VaultHub.sol (line 394):

```
Python
function updateShareLimit(address _vault, uint256 _shareLimit) external
onlyRole(VAULT_MANAGER)
{
   // DAO can set shareLimit to 0 to restrict minting
}
```

VaultHub.sol (line 394):

```
Python
function changeTier( address _vault, uint256 _requestedTierId, uint256 _requestedShareLimit )
external
{
   vaultHub.updateConnection(
      _vault,
      _requestedShareLimit,
      requestedTier.reserveRatioBP,
      requestedTier.forcedRebalanceThresholdBP,
      requestedTier.infraFeeBP,
```

```
            requestedTier.liquidityFeeBP,
            requestedTier.reservationFeeBP
        );
    }
```

Bypass scenario:

1. DAO sets shareLimit = 0 to temporarily lock minting
2. Owner and Node Operator coordinate to change vault tier
3. Tier change calls vaultHub.updateConnection() with new tier parameters
4. _requestedShareLimit set to new tier's default (non-zero)
5. DAO zero-limit protection bypassed without direct updateShareLimit call

While tier changes require Node Operator cooperation, this still circumvents the intended zero-limit protection mechanism that should be under sole owner control.

Recommendations: Preserve explicitly set zero limits through tier changes, or require explicit owner confirmation before resetting limits from zero to non-zero values.

Lido's response: Fixed in PR #1355. The implementation now has a jail logic to prevent this behavior.

Fix Review: Fixed with Jail mechanism in case this happens.

M-24 - Node Operator Can Block Vault Disconnection by Refusing Fee Disbursement

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: Dashboard.sol, NodeOperatorFee.sol, VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

A malicious node operator could prevent vault disconnection or ownership transfer by refusing to disburse accumulated fees.

If fees exceed the abnormally high threshold (1% of total value, typically reached after ~2 years without disbursement), both voluntaryDisconnect() and transferVaultOwnership() operations would revert, effectively locking the vault owner from exiting their position because there is no automatic fee distribution mechanism on disconnect/transfer, allowing fees to accumulate indefinitely without resolution.

Recommendations: Implement forced fee distribution on disconnect and ownership transfer operations. Add an administrative override mechanism to handle abnormally high fees that may result from extended non-disbursement periods.

Lido's response: Fixed in #PR1541. Distribute and disable node operator fees on disconnect or transferOwnership.

Fix Review: The fix has been properly implemented in PR #1541 with multiple safeguards (automatic fee distribution, fee accrual stop, escrow mechanism and admin override to disburse abnormally high fees)

M-25 - Missing check that the owner of the stakingVault is msg.sender when connecting vault

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: VaultHub.sol, VaultFactory.sol	Status: Fixed	Violated Property: N/A

Description:

The vault connection mechanism fails to verify that `msg.sender` is the actual vault owner, allowing Bob to connect Alice's vault without Alice's agreement.

```
Python
function connectVault( address _vault, uint256 _requestedShareLimit ) external
{
   // Missing check: require(IStakingVault(_vault).owner() == msg.sender)
   // Anyone can connect
   _connectVault(_vault, _requestedShareLimit);
}
```

The missing ownership verification allows anyone to trigger vault connection for anyone if vaultHub has pending ownership, undermining the intended access control model.

Recommendations: Add explicit ownership verification in connectVault(): require(IStakingVault(_vault).owner() == msg.sender, "Not vault owner") to ensure only the current owner can initiate connections.

Lido's response: Fixed in audit response #PR1491. The fix adds the missing ownership check to prevent unauthorized connection attempts.

Fix Review: Fixed.

M-26 - addFeeExemption with max_int128 causes permanent disconnect DOS through underflow

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: OperatorGrid.sol, VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

Node Operators can permanently lock vault owners out by setting fee exemptions to maximum values, causing arithmetic underflow when calculating fees during disconnection.

NodeOperatorFee.sol:

```python
function addFeeExemption( address _vault, int128 _exemptedAmount ) external
{
   // NO can set _exemptedAmount = type(int128).max
   settledGrowth = type(int128).max;
}

function _calculateFee(...) internal
{
   unsettledGrowth = currentGrowth - settledGrowth
   // If settledGrowth = max_int128, underflow occurs
   // Prevents fee calculation required for disconnection
}
```

Attack scenario:

1. Node Operator calls `addFeeExemption(_vault, type(int128).max)`
2. `settledGrowth` becomes `type(int128).max`
3. Vault experiences any slashing (which NO can trigger)
4. Owner attempts to disconnect vault
5. Disconnection requires fee calculation in `_calculateFee()`
6. Calculation: `unsettledGrowth = currentGrowth - settledGrowth`

7. Since `settledGrowth = max_int128`, operation underflows
8. Transaction reverts, preventing disconnection permanently

This enables malicious Node Operators to permanently lock vaults by exploiting the fee exemption mechanism, preventing owners from ever disconnecting.

Recommendations: Add maximum value validation for `_exemptedAmount` or use unchecked arithmetic with proper bounds checking to prevent underflow in fee calculations.

Lido's response: Fixed in audit response #PR1528. The implementation now validates fee exemption amounts and handles extreme values safely.

Fix Review: The fix caps exemption amounts at reasonable values and refactors fee calculation to handle edge cases without underflow, ensuring vault owners can always disconnect regardless of fee exemption settings.

M-27 - Bypass isApprovedToConnect check through ownership transfer chain

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: VaultHub.sol, OperatorGrid.sol, StakingVault.sol, Dashboard.sol	Status: Fixed	Violated Property: N/A

Description:

The connection approval security check can be bypassed through a complex ownership transfer sequence involving the Dashboard contract.

Attack flow:

1. Create vault via VaultFactory with Dashboard as owner
2. Vault automatically connected to VaultHub
3. Call `voluntaryDisconnect()` on Dashboard
4. Call `Dashboard.abandonDashboard(_attacker)` - transfers ownership to attacker
5. Attacker accepts ownership on StakingVault
6. Attacker initiates ownership transfer to VaultHub (pending state)
7. Anyone calls `connectVault()` - connection succeeds with `connection.owner = _attacker`
8. Attacker calls `transferVaultOwnership()` back to Dashboard address
9. Dashboard functions still work (stateless design)
10. Result: Vault connected with attacker as recorded owner, bypassing `isApprovedToConnect`

Recommendations: Verify the complete ownership history chain matches approved connections, or invalidate connection approvals when ownership transfers occur.

Lido's response: Fixed in PR #1512.

Fix Review: Fixed.

M-28 - Bypass fees by rebalancing

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: VaultHub.sol, OperatorGrid.sol	Status: Fixed	Violated Property: N/A

Description:

Vault owners can indefinitely delay fee payments by cycling through mint and rebalance operations, circumventing withdrawal restrictions and forced validator exit conditions.

VaultHub.sol:

```python
function withdraw(address _vault, uint256 _amount) external
{
   require(amount <= _withdrawableValue, "Insufficient withdrawable");
   // If fees > 1 ETH, can't withdraw that amount
   // But can mint shares and rebalance to bypass restriction
}
```

Bypass scenario for fee payment:

1. Vault has 1 ETH in fees, limiting withdrawable amount
2. Owner calls `mintShares()` to borrow against vault
3. Owner immediately calls `rebalance()` to convert liability to stETH
4. Rebalancing pays back shares, reducing liabilities
5. Reduced liabilities appear as if fees were paid
6. Repeat: Owner can withdraw without actually paying NO fees

The same is possible for `forceValidatorExit()` and for `triggerValidatorWithdrawals()`:

Recommendations: Track cumulative rebalanced amounts within each refSlot period and enforce one-time settlement, or implement cooldown periods between mint/rebalance operations.

Lido's response: Addressed through enhanced obligation refactoring process in #PR1287.

Fix Review: Fixed.

M-29 - Block disconnection by setting malicious fee recipient

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: Dashboard.sol, OperatorGrid.sol, VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

Node Operators can prevent vault disconnection by setting a fee recipient address that reverts on ETH receipt, causing all disconnection attempts to fail.

VaultHub.sol::VaultConnectionUpdated():1577

```python
function setFeeRecipient(address _newFeeRecipient) external
{
   // NO can set recipient to malicious contract
   feeRecipient = _newFeeRecipient;
}

function disburseNodeOperatorFee() internal
{
   // Called during disconnection
   payable(feeRecipient).transfer(feeAmount);
   // If feeRecipient reverts, entire disconnection fails

}
```

Attack scenario:

1. Node Operator sets `feeRecipient` to malicious contract
2. Malicious contract's `receive()` function always reverts
3. Vault owner attempts to disconnect via `voluntaryDisconnect()`

4. Disconnection process calls `disburseNodeOperatorFee()`
5. Fee transfer to recipient reverts
6. Entire disconnection transaction reverts
7. Vault permanently locked in connected state

This gives Node Operators the ability to hold vaults hostage by preventing disconnection through a permanently reverting fee recipient.

Recommendations: Use a pull payment pattern for fee distribution instead of push payments, or collect fees to an escrow address that NOs can claim from separately, decoupling fee payment from disconnection logic.

Lido's response: Fixed in #PR1528. Fees are now collected to the Dashboard contract during disconnection and can be separately recovered by NO.

Fix Review: Fixed. NO can no longer block maliciously a disconnection process.

Low Severity Issues

L-01 - Manual pause flag conflict with automatic pause in deposit control

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: VaultHub.sol, StakingVault.sol	Status: Fixed	Violated Property: N/A

Description:

The deposit pause mechanism has an asymmetric design where manual and automatic pauses interact inconsistently, causing vault owners to lose fine-grained control during critical periods.

VaultHub.sol :

```python
// line 1461
if (!isBeaconDepositsPaused) vault_.pauseBeaconChainDeposits();

// line 778
IStakingVault(_vault).pauseBeaconChainDeposits();

// line 787
if (!_isVaultHealthy(connection, record)) revert UnhealthyVaultCannotDeposit(_vault);
```

The asymmetry creates these problems:

Scenario 1 - Hub pauses first:

1. VaultHub automatically pauses deposits (vault unhealthy)
2. Owner cannot add manual pause on top (only one pause flag)
3. When vault becomes healthy, `resumeBeaconChainDeposits()` auto-resumes
4. Owner loses ability to keep deposits paused even after vault recovers

Scenario 2 - Owner pauses first:

1. Owner manually pauses deposits
2. VaultHub later automatically pauses (vault unhealthy)
3. Both flags effectively active
4. Owner must wait for vault to become healthy before they can manually resume
5. Cannot "pre-approve" resumption while vault is still unhealthy

The design prevents owners from asserting control during the critical period when automatic pause is active, removing their ability to prevent unwanted auto-resumption.

Recommendation: Decouple manual and automatic pause flags into separate state variables. Require both flags to be cleared before deposits can resume, giving owners veto power over automatic resumption.

Lido's response: Fixed in PR #1277. The implementation separates pause controls, allowing owner intent to persist through automatic pause cycles.

Fix Review: The fix introduces separate tracking for manual pause intent (beaconChainDepositsPauseIntent) and automatic pause state.

L-02 - Zero external shares despite non-zero vault liabilities due to rounding

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: VaultHub.sol, Lido.sol	Status: Fixed	Violated Property: N/A

Description:

When the internal ETH:shares ratio falls below 1, repeated small rebalances can burn more external shares than liability shares due to rounding up in share value calculations.

vaultHub.sol

```python
function _rebalance(address _vault, VaultRecord storage _record, uint256 _shares) internal
{
   // Rounds up when converting shares to ETH value
   uint256 valueToRebalance = _getPooledEthBySharesRoundUp(_shares);
   // Burns external shares based on rounded-up value
}
```

Vulnerability scenario (when ratio is 35 ETH : 36 shares):

1. Vault has liabilities: 35 liability shares
2. Initial external shares: 36
3. Rebalance 35 shares: `_getPooledEthBySharesRoundUp(35)` = 36 ETH equivalent
4. Burns 36 external shares (all of them)
5. Liability shares still 35 (non-zero)
6. Result: `externalShares = 0`, `liabilityShares = 35`

This creates an accounting mismatch where the last vaults disconnecting could have non-zero liabilities but zero external shares backing them. The vault becomes unable to disconnect unless VaultHub manually internalizes the debt.

Recommendations: Use consistent rounding direction (round down for both share burns and liability calculations), or track and periodically reconcile rounding errors to prevent accumulation.

Lido's response: Fixed in #PR1419.

Fix Review: Fixed.

L-03 - Inconsistent behavior when totalUnsettledObligations equals MIN_BEACON_DEPOSIT

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

Deposits pause and resume logic uses inconsistent comparison operators for the edge case where unsettled obligations exactly equal `MIN_BEACON_DEPOSIT` (1 ETH).

VaultHub.sol (line 1460 - pause logic):

```python
// Pauses when totalUnsettledObligations >= MIN_BEACON_DEPOSIT
if (totalUnsettledObligations >= MIN_BEACON_DEPOSIT) {
    if (!isBeaconDepositsPaused) vault_.pauseBeaconChainDeposits();
}
```

VaultHub.sol (line 1401 - resume logic):

```python
// Resumes when totalUnsettledObligations < MIN_BEACON_DEPOSIT
if (totalUnsettledObligations < MIN_BEACON_DEPOSIT) { // Allow resume }
```

Edge case behavior when `totalUnsettledObligations == 1 ether`:

1. Pause condition: `1 ether >= MIN_BEACON_DEPOSIT` → TRUE (pauses)
2. Resume condition: `1 ether < MIN_BEACON_DEPOSIT` → FALSE (won't resume)
3. Deposits remain paused at exact boundary
4. Requires obligations to drop below 1 ETH to resume

However, this creates confusion about intended behavior:

- Should 1 ETH exactly be treated as requiring pause or allowing deposits?
- Using >= for pause but < for resume creates hysteresis at the boundary
- The vault oscillates between paused/resumed if obligations hover at exactly 1 ETH

Recommendation: Use consistent comparison operators (both <= or both <) to clarify intended behavior at the boundary. Document whether 1 ETH exactly should allow or prevent deposits.

Lido's response: Fixed in #PR1350.

Fix Review: Fixed by removing one of the comparisons.

L-04 – Missing connection check allows operations on disconnected vaults

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

The `settleVaultObligations()` function can be executed on disconnected vaults, allowing state modifications on vaults that are no longer tracked by VaultHub.

VaultHub.sol (line 911):

```python
function settleVaultObligations(address _vault) external
{
   // Missing: _requireConnected() check
   // Function proceeds to settle obligations even if vault disconnected
   VaultRecord storage record = _vaultRecord(_vault);
   VaultObligations storage obligations = _vaultObligations(_vault);
   _settleObligations(_vault, record, obligations, NO_UNSETTLED_ALLOWANCE);
}
```

Impact:

1. Disconnected vaults can have their obligations settled
2. State changes occur on vaults not actively managed by VaultHub
3. Creates inconsistent accounting between connected and disconnected vaults
4. Obligations updated for vaults that shouldn't be participating in protocol

This breaks the invariant that only connected vaults should have their obligations actively managed by the protocol.

Recommendation: Add _requireConnected(_vault) check at the beginning of settleVaultObligations() to ensure operations only affect active vaults.

Lido's response: Fixed by adding connection requirement check to the function.

Fix Review: The fix ensures obligation settlement only occurs for vaults actively connected to VaultHub, maintaining proper accounting boundaries.

L-05 - Quarantine penalizes legitimate rewards up to sanity limit

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: LazyOracle.sol, OracleReportSanityChecker.sol	Status: Acknowledged	Violated Property: N/A

Description:

When a vault's reported total value exceeds the sanity check limit, the protocol reverts the entire reported value back to the on-chain value, quarantining all rewards including legitimate ones up to the maximum allowed reward ratio.

LazyOracle.sol (original problematic code):

```python
if (_reportedTotalValue > maxSaneTotalValue) {
    // Reverts entire reported value to on-chain value
    _reportedTotalValue = onchainTotalValueOnRefSlot;
    // Quarantines ALL increases, even legitimate rewards
}
```

Example scenario:

1. Vault has 100 ETH on-chain
2. Earns 10 ETH in legitimate rewards
3. Reported value: 110 ETH
4. Max sane value (max reward 8%): 108 ETH
5. Current behavior: Entire 110 ETH rejected, revert to 100 ETH
6. Result: ALL 10 ETH quarantined (including 8 ETH that's legitimate)

Expected behavior:

- Accept 108 ETH (legitimate portion)
- Quarantine only 2 ETH (excess above limit)

The issue penalizes vaults for earning normal rewards, when only the excess should be quarantined.

Recommendation: Set _reportedTotalValue = maxSaneTotalValue instead of reverting to old value, accepting legitimate rewards up to the limit and only quarantining the excess.

Lido's response: Acknowledged in issue #1284.

Fix Review: It is a compromise between users and protocol that is acceptable to keep.

L-06 – Cannot set redemptions to zero when liability is zero

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

The function setVaultRedemptions() contains an illogical restriction that prevents setting redemptions to zero when the vault has zero liability.

VaultHub.sol (line 898):

```python
function setVaultRedemptions(address _vault, uint256 _redemptions) external
{
   uint256 liability = _record.liabilityShares;
   // Prevents setting redemptions=0 when liability=0
   if (_redemptions == 0 && liability > 0)
   {
      // Only allows zero redemptions if liability > 0
      revert CannotSetZeroRedemptions();
   }
   // But if liability == 0, setting redemptions=0 also reverts elsewhere
}
```

The logic creates an inconsistency:

- If liability > 0: CAN set redemptions = 0 (makes sense)
- If liability == 0: CANNOT set redemptions = 0 (illogical)

This creates an edge case where a vault with zero liability cannot explicitly clear its redemption obligations to zero, even though that would be the natural state.

Recommendation: Allow zero redemption setting regardless of liability value, or invert the condition to explicitly handle the zero liability case.

Lido's response: Fixed in #PR1350 by removing the illogical restriction.

Fix Review: The fix removes the asymmetric logic, allowing redemptions to be set to zero regardless of liability state, which is the expected behavior.

L-07 - Permanently stuck ETH via payable depositToBeaconChain

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files : PredepositGuarantee.sol	Status: Fixed	Violated Property: N/A

Description:

The `depositToBeaconChain()` function in PredepositGuarantee is marked as `payable` but never uses `msg.value`, causing any accidentally sent ETH to become permanently stuck in the contract with no recovery mechanism.

PredepositGuarantee.sol:

```
Python
function depositToBeaconChain(
    address stakingVault,
    DepositData[] calldata deposits
) external payable {  // <-- payable modifier
    // ... //
}
```

The payable modifier is unnecessary since the function doesn't need to receive ETH - it handles deposits from the staking vault's balance, not from msg.value.

Recommendations : Remove the payable modifier, or if ETH is intended to be received, forward msg.value to the staking vault or implement a recovery mechanism.
Lido's response: Fixed in #PR1336 by removing the payable modifier since the function doesn't require ETH payments.

Fix Review: The fix prevents accidental ETH loss by making the function revert if called with value, which is the correct behavior since deposits come from the vault's balance.

L-08 – Default tier restoration bypass via disconnect/reconnect		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files : VaultHub.sol, OperatorGrid.sol	Status: Acknowledged	Violated Property: N/A

Description:

A check exists to prevent vaults from changing to the default tier, but this restriction can be bypassed by disconnecting and reconnecting the vault, which automatically resets it to the default tier.

OperatorGrid.sol (line 404):

```
Python
// Prevents changing TO default tier
if (_requestedTierId == DEFAULT_TIER_ID) {
    revert CannotChangeToDefaultTier();
}
```

But in VaultHub connection logic:

```
Python
function _connectVault(address _vault, uint256 _shareLimit) internal {
    // When connecting, vault assigned to DEFAULT_TIER automatically
    _record.tier = DEFAULT_TIER_ID;
}
```

A disconnect/reconnect dance circumvents the intended tier assignment enforcement.

Recommendations : Update the code so that the best thing to do is to change tier to the default one so that there is nothing that can be won for NodeOperator or VaultOwner as a result of such a movement.

Lido's response: Acknowledged. With the subsequent code updates there is no longer a real impact in switching to DEFAULT TIER now.

Fix Review: The behavior is considered acceptable as reconnection is treated as creating a fresh vault state.

L-09 - Reverse limiting order suboptimal in sanity checker		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: OracleReportSanityChecker.sol	Status: Acknowledged	Violated Property: N/A

Description:

In OracleReportSanityChecker, the order of operations processes additions (which worsen the situation) before decreases (which improve it), potentially preventing beneficial decreases from executing.

OracleReportSanityChecker.sol (lines 429-431):

```
Python
// Process additions FIRST (these worsen the situation)
withdrawals = tokenRebaseLimiter.increaseEther(_withdrawalVaultBalance);
elRewards = tokenRebaseLimiter.increaseEther(_elRewardsVaultBalance);
```

OracleReportSanityChecker.sol (line 438):

```
Python
// Process decrease AFTER additions
tokenRebaseLimiter.decreaseEther(_etherToLockForWithdrawals);
```

Impact of current order:

1. Additions processed first, potentially hitting rate limits
2. If limits hit, subsequent operations may revert
3. Decrease operation (which frees up capacity) never executes
4. Users cannot benefit from the capacity that would be freed

Better order would be:

1. Process decrease FIRST (frees up capacity)
2. Process additions AFTER (use newly available capacity)

3. More operations can complete successfully

The current order favors protocol conservatism over user benefit by potentially blocking operations that would actually improve the situation.

Recommendations : Reverse the order - apply decreases before increases to maximize the number of operations that can complete within rate limits.
Lido's response: Acknowledged as suboptimal but low impact. The team noted that reordering could be done but the current order matches their mental model of processing.

Fix Review: In practice, the impact is minimal since rate limits are set conservatively enough that both orders typically succeed.

L-10 - If shares rate is less than 1 anyone can transfer shares from anyone else

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: StETH.sol	Status: Acknowledged	Violated Property: N/A

Description:

When the share rate falls below 1 ETH per share, anyone can call `transferSharesFrom()` without proper allowance if the ETH value of the shares being transferred rounds to zero.

StETH.sol (lines 388-396):

```
Python
function transferSharesFrom(
    address _sender, address _recipient, uint256 _sharesAmount
) external returns (uint256) {
    uint256 tokensAmount = getPooledEthByShares(_sharesAmount);
    // If tokensAmount rounds to 0, allowance check passes with 0
    _spendAllowance(_sender, msg.sender, tokensAmount);
    _transferShares(_sender, _recipient, _sharesAmount);
    return tokensAmount;
}
```

StETH.sol (lines 462-468):

```
Python
function _spendAllowance(address _owner, address _spender, uint256 _amount) internal {
    uint256 currentAllowance = allowances[_owner][_spender];
    if (currentAllowance != INFINITE_ALLOWANCE) {
        // If _amount is 0, this check passes with any allowance
        require(currentAllowance >= _amount, "ALLOWANCE_EXCEEDED");
        _approve(_owner, _spender, currentAllowance - _amount);
    }
}
```

Attack scenario (when share rate < 1):

1. Share rate: 0.9 ETH per share
2. Attacker calls transferSharesFrom(victim, attacker, 1) with 1 share
3. getPooledEthByShares(1) = 0 (rounds down)
4. _spendAllowance(victim, attacker, 0) passes (0 <= any allowance)
5. 1 share transferred without proper authorization

While this requires the extremely unlikely scenario of share rate < 1, it represents a logical flaw in access control.

Recommendations : Add explicit zero-value checks in both transferSharesFrom() and _spendAllowance() to prevent transfers when the token amount is zero.
Lido's response: Marked as duplicate of existing issue #796. While theoretically possible, the scenario of share rate < 1 is extremely unlikely in practice would require burning thousands of wei i gas to steal 1 wei of stETH this way.

Fix Review: Acknowledged.

L-11 - Revert in applyVaultReport because of underflow

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: LazyOracle.sol, VaultHub.sol	Status: Acknowledged	Violated Property: N/A

Description:

When oracle reports are submitted out of order and ETH is gifted/rebalanced between submissions, an arithmetic underflow can occur in `_handleSanityChecks()`, causing report application to revert.

LazyOracle.sol:

```
Python
function _handleSanityChecks(...) internal {
    // Can underflow when reports submitted out of order
    uint256 calculation = totalValueWithoutQuarantine
        - currentInOutDelta
        + inOutDeltaOnRefSlot;
}
```

Attack scenario:

1. RefSlot 1 report submitted during RefSlot 3 period
2. Vault total value: 31 ETH (reported)
3. Gift ETH to vault and rebalance before RefSlot 2 report
4. RefSlot 2 report submission calculates:
    - totalValueWithoutQuarantine = 31
    - currentInOutDelta = 0 (reset by gift/rebalance)
    - inOutDeltaOnRefSlot = 32 (from old report)
5. Calculation: 31 - 0 + 32 = overflow/underflow
6. Transaction reverts, report cannot be applied

This DOS can occur naturally through timing issues or be exploited by gifting ETH to cause underflow in report processing.

Recommendations : Add bounds checking or use safe math with underflow handling to gracefully handle out-of-order report scenarios.

Lido's response: Acknowledged because this issue is rare,would need resources and the DOS is only possible for 1 day, then it resolves itself.

Fix Review: Acknowledged on Certora side.

L-12 - Should check pause deposits for a vault that receives badDebt

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: VaultHub.sol	Status: Fixed	Violated Property: N/A

Description:

When a vault receives bad debt through `socializeBadDebt()`, deposits should be automatically paused to prevent new users from entering a vault with existing losses. However, the current implementation lacks this check.

VaultHub.sol:

```python
function socializeBadDebt(address _vault, address[] calldata _acceptorVaults) external {
    // Bad debt assigned to acceptor vaults
    for (uint i = 0; i < _acceptorVaults.length; i++) {
        // Missing: Pause deposits for vault receiving bad debt
        _increaseLiability(_acceptorVaults[i], ...);
    }
}
```

Risk:

1. Vault receives bad debt (losses socialized to it)
2. Deposits remain enabled
3. New users can deposit into vault with existing losses
4. New depositors immediately absorb portion of pre-existing bad debt
5. Unfair to new entrants who weren't aware of the loss situation

Deposits should be paused when bad debt is assigned to prevent diluting the loss across new depositors.

Recommendations : Automatically pause deposits for vaults that receive bad debt assignments, requiring manual review and unpause by governance or vault owner.

Lido's response: Fixed in #PR1435. The team added deposit pause logic to bad debt socialization flows.

Fix Review: The fix checks if pauses should be triggered for deposits for vaults receiving bad debt, protecting new depositors from immediately absorbing socialized losses.

L-13 - proveAndDeposit should not be payable

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: PredepositGuarantee.sol	Status: Fixed	Violated Property: N/A

Description:

The `proveAndDeposit()` function in PredepositGuarantee is incorrectly marked as `payable` when it should not accept ETH payments.

PredepositGuarantee.sol:

```python
function proveAndDeposit(
    address stakingVault,
    bytes calldata pubkey,
    bytes calldata withdrawalCredentials,
    bytes calldata proof,
    bytes32 beaconBlockRoot
) external payable {  // <-- Should not be payable
    // Function doesn't use msg.value
    // ETH handling is done through vault balance, not msg.value
}
```

Recommendations : Remove the payable modifier to prevent accidental ETH loss and clarify the function's actual behavior.

Lido's response: Fixed in #PR1466 by removing the payable modifier.

Fix Review: Fixed.

L-14 - Impossible to disconnect vault with fees exceeding vault value

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: VaultHub.sol, OperatorGrid.sol	Status: Fixed	Violated Property: N/A

Description:

When a vault's total value is zero or negative but accumulated fees plus balance exceed the value, the vault cannot be disconnected even by administrators using the `_forceFullFeesSettlement` flag.

VaultHub.sol:

```python
function _initiateDisconnection(
    address _vault,
    VaultConnection storage _connection,
    VaultRecord storage _record,
    bool _forceFullFeesSettlement
) internal {
    uint256 unsettledLidoFees = _unsettledLidoFeesValue(_record);
    if (unsettledLidoFees > 0) {
        uint256 balance = Math256.min(_availableBalance(_vault), _totalValue(_record));
        if (_forceFullFeesSettlement) {
            // BUG: Reverts when balance < fees, even in force mode
            if (balance < unsettledLidoFees) {
                revert NoUnsettledLidoFeesShouldBeLeft(_vault, unsettledLidoFees);
            }
        }
    }
}
```

Attack scenario:

1. Vault slashed, total value ≈ 0
2. Fees accumulated: 2 ETH

3. Vault balance: 1 ETH
4. Admin attempts forced disconnection
5. Check: 1 ETH < 2 ETH → reverts
6. Attacker donates 1 wei to vault repeatedly after each report
7. Keeps fees > balance condition true
8. Vault permanently locked, cannot disconnect

The forced settlement flag should allow disconnection by settling what's available, but instead it still requires full fee payment.

Recommendations : In force mode, settle available balance and allow disconnection even with partially unpaid fees, or write off remaining fees.

Lido's response: Fixed in #PR1498 by allowing partial fee settlement in force disconnection mode and implementing fee write-off mechanisms.

Fix Review: The fix modifies forced disconnection to settle whatever fees are payable from available balance, then proceeds with disconnection regardless. Unpaid fees can be written off through governance if needed, preventing permanent vault lock scenarios.

L-15 - Permanently Locked Node Operator Balance Blocks Guarantor Updates

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: PredepositGuarantee.sol	Status: Fixed	Violated Property: N/A

Description:

Node operators cannot change guarantors when `balance.locked` is non-zero. If a vault disconnects before invalid withdrawal credentials are proven via `proveInvalidValidatorWC()`, the locked balance persists permanently. After disconnection, the vault owner can zero the staged balance, making it impossible to prove the invalid credentials and unlock the balance.

VaultHub.sol:

```
Python
// PredepositGuarantee.sol
function setNodeOperatorGuarantor(address _newGuarantor) external whenResumed {
    NodeOperatorBalance storage balance = $.nodeOperatorBalance[msg.sender];
    // ...
    if (balance.locked != 0) revert LockedIsNotZero(balance.locked);  // Blocks execution
    // ...
}

function proveInvalidValidatorWC(
    ValidatorWitness calldata _witness,
    bytes32 _invalidWithdrawalCredentials
) external whenResumed {
    // ...
    if (validator.stage != ValidatorStage.PREDEPOSITED) {
        revert ValidatorNotPreDeposited(_witness.pubkey, validator.stage);  // Reverts after
disconnect
    }
```

```
    // Compensation logic that unlocks balance never executes
    balance.locked -= PREDEPOSIT_AMOUNT;
}
```

Recommendations : Allow compensation to proceed even after vault disconnection, or implement an alternative mechanism to unlock the node operator's balance when validators remain in PREDEPOSITED state after disconnection.

Lido's response: Fixed in PR #1467 (Audit fixes 3).

Fix Review: Node operators can now update guarantors after disconnection without being permanently blocked by locked balances.

L-16 - Fee Updates Blocked by Reserve Ratio Check and Pending Disconnect State		
Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: VaultHub.sol	Status: Not Fixed	Violated Property: N/A

Description: The updateConnection() function prevents fee parameter updates when a vault breaches its reserve ratio, even though fee updates don't affect vault collateralization. Additionally, vaults with aborted disconnect attempts remain locked from parameter updates due to the _checkConnection() guard.

When a vault's liability exceeds its reserve ratio but remains below the forced rebalance threshold (healthy but over-collateralized), updating fees becomes impossible , the fee updates are blocked by reserve ratio. Fee parameters (infraFeeBP, liquidityFeeBP, reservationFeeBP) don't affect the vault's collateral structure, yet they're blocked by the same health check as structural parameters.

Recommendations : Create separate functions for updating fees.

Lido's response: Issue has been fixed here.

Fix Review: Fix

L-17 - Node Operator Fees Lost When Vault is Force Disconnected by DAO

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: VaultHub.sol,NodeOperatorFee.sol	Status: Acknowledged	Violated Property: N/A

Description:

When the DAO/manager force disconnects a vault via VaultHub.disconnect(), accrued node operator fees are not collected. The Dashboard owner can subsequently call abandonDashboard() to transfer vault ownership without paying these accumulated fees, resulting in permanent loss of node operator compensation.

VaultHub.sol:

```python
function abandonDashboard(address _newOwner) external {
    if (VAULT_HUB.isVaultConnected(address(_stakingVault()))) revert ConnectedToVaultHub();
    if (_newOwner == address(this)) revert DashboardNotAllowed();

    _acceptOwnership();
    _transferOwnership(_newOwner);
}
```

Recommendations : Add fee collection logic to abandonDashboard() to ensure node operator fees are properly secured before transferring vault ownership.

Lido's response: Acknowledged for forced disconnect and fixed for voluntary one. Forced disconnections are subject to Lido DAO governance decisions and this framework gives plenty of time to distribute any fees beforehand.

Fix Review: Acknowledged.

L-18 - Default tier params not validated during initialization

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: OperatorGrid.sol	Status: Fixed	Violated Property: N/A

Description:

`OperatorGrid.initialize()` doesn't call `_validateParams()` for the default tier unlike `registerTiers()`. This means the default tier could have invalid parameters, such as `forcedRebalanceThresholdBP > reserveRatioBP`. This creates systemic risk since all new vaults start with potentially invalid parameters.

Recommendations: Add _validateParams() call for the default tier during initialization to ensure tier parameter validity from deployment.

Lido's response: Fixed in PR #1317. Default tier parameters now undergo validation during OperatorGrid initialization.

Fix Review: Fixed. Initialization now validates default tier parameters using the same validation logic applied to custom tiers, preventing invalid configurations.

L-19 - Self-Induced Vault Unhealthiness via Unfavorable Rebalance Rounding

Severity: **Medium**	Impact: **Low**	Likelihood: **Low**
Files: VaultHub.sol	Status: Acknowledged	Violated Property: N/A

Description:

Vault owners can intentionally degrade their health factor through small rebalances exploiting round-up behavior. When rebalancing 1 share worth 1.1 ETH, `_getPooledEthBySharesRoundUp()` withdraws 2 wei while decreasing liability by only 1 share. Repeating this operation degrades the vault's collateral-to-liability ratio, making a healthy vault unhealthy through systematic unfavorable rounding.

```
Python
// VaultHub.sol (around lines 1118-1123)
function _rebalance(address _vault, VaultRecord storage _record, uint256 _shares) internal
{
   uint256 valueToRebalance = _getPooledEthBySharesRoundUp(_shares);
   // Rounds up _decreaseLiability(_vault, _record, _shares);
   // Decreases shares exactly
   _withdraw(_vault, _record, address(this), valueToRebalance);
   // Withdraws rounded amount
   // Each call: withdraw 2 wei but decrease liability by 1 share
   // Repeated execution degrades vault health
}
```

Recommendations : Add health factor validation preventing healthy vaults from becoming unhealthy through rebalancing operations.

Lido's response: Acknowledged, a very legitimate alternative way to reduce health just by not staking also exists which is better than this one and does not require burning thousands of wei for each 1 wei of imbalance.

Fix Review: Acknowledged, rounding is in favor of Lido Protocol.

L-20 - Solidity 0.4.24 allocates more memory than needed

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: deposit_contract.sol	Status: Acknowledged	Violated Property: N/A

Description:

A compiler bug in Solidity 0.4.24 causes excessive memory allocation for constant-size byte arrays. When creating `new bytes(48)`, the compiler allocates `32 * 48 = 1536` bytes instead of the intended 48 bytes.

SigningKeys.sol:

```python
function saveKeysSigs( uint256 _nodeOperatorId, uint256 _startIndex, uint256 _keysCount,
bytes _pubkeys, bytes _signatures ) external
{
   // Each signature is 96 bytes (BLS signature)
   // Code uses: new bytes(48) for pubkey allocation
   // Compiler bug: allocates 1536 bytes instead of 48
   bytes memory pubkey = new bytes(48);
   // ... signature processing
}
```

Impact:

- Every call to `saveKeysSigs` wastes approximately 1488 bytes of memory per key
- For batch operations with multiple keys, waste multiplies significantly
- Increased gas costs for all key signature save operations
- Affects all deployed contracts using Solidity 0.4.24

The bug occurs because Solidity 0.4.24 misinterprets constant array size declarations, treating the size as a multiplier rather than the total byte count.

Recommendations: Upgrade to a more recent solidity version.All subsequent key management operations will use correct memory allocation.

Lido's response: Acknowledged. The found problem is a optimization, although a tricky one and does not have any security impact, also, the mentioned library is used in NodeOperatorRegistry that is not touched by V3 update .The bug is noted and a fix is planned to be delivered with the next NodeOperatorRegistry update.

Fix Review: For existing deployed contracts, the inefficiency remains but is tolerable given the contract's limited usage patterns.

L-21 - Predeposit Frontrun Compensation could bypass InOutDelta Tracking and Triggers Quarantine

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: PredepositGuarantee.sol, LazyOracle.sol	Status: Acknowledged	Violated Property: N/A

Description:

When a predeposit is frontrun and compensated via proveInvalidValidatorWC(), the 1 ETH compensation is transferred directly to the vault without updating the VaultHub's inOutDelta tracking.
This can cause an increase in totalValue that can trigger quarantine, penalizing the vault for a legitimate compensation payment.

Recommendations: Update inOutDelta when PDG compensation is transferred to the vault.

Lido's response: Acknowledged, for the average situation it will not happen, because 1/31 ETH = 3.2% increase with the quarantine limit being 3.5%, so it even has a room for legit rewards.

Fix Review: Acknowledged.

Informational Severity Issues

I-01 - Missing array length validation in batch predeposit validation

Description: PredepositGuarantee.batchProveValidatorWC() accepts _pubkeys and _proofs arrays without validating equal length. The function iterates through these parallel arrays assuming they match, which could lead to unexpected behavior or reverts if arrays have different lengths.

Recommendations : Add explicit array length equality check at function start to fail immediately with clear error message when arrays don't match.

Lido's response: Missing check added.

Fix Review: Fixed.

I-02 - Division by Zero in get_steth_by_shares Function

Description:

NodeOperatorFee._setNodeOperatorFeeRate() contains redundant validation. The function validates the fee rate parameter even though its only caller already performs the same validation, wasting gas on every fee update.

Recommendations: Remove _validateNodeOperatorFeeRate() function and inline the validation check directly in _setFeeRate() to eliminate redundancy.

Lido's response: Removed the separate _validateNodeOperatorFeeRate() function and consolidated validation into a single check within _setFeeRate().

Fix Review: Comment updated to clarify that while node operator can pre-approve tier changes, the actual tier change execution requires vault to be connected.

I-03 - Missing return value breaks compensation amount tracking

Description:

Dashboard#compensateDisprovenPredepositFromPDG() discards the return value from the compensation call chain. While VaultHub, Permissions, and PredepositGuarantee all return PREDEPOSIT_AMOUNT (1 ETH) to

indicate successful compensation, Dashboard's wrapper function declared void return type, breaking composability.

Recommendation: Propagate the return value through Dashboard function to maintain composability and enable programmatic verification of compensation amounts.
Lido's response: Fixed.

Fix Review: Comment updated to clarify this process.

I-04 - Outdated comment about non-connected vault tier changes

Description:

OperatorGrid.changeTier() contains a misleading comment stating "Node operator confirmation can be collected even if the vault is disconnected." However, VaultHub.updateTier() now reverts for disconnected vaults, making the documented behavior impossible.

The comment suggests tier changes can proceed on disconnected vaults, but the implementation requires vault connection for the final execution. This discrepancy between documentation and behavior can mislead developers and auditors.

Recommendation: Update or remove the outdated comment to accurately reflect current implementation behavior that requires vault connection for tier changes.
Lido's response: Fixed.

Fix Review: Comment updated.

I-05 - Zero bad debt write-off incorrectly allowed

Description:

VaultHub._badDebtShares() and related bad debt functions allow execution when totalValueShares equals liabilityShares (zero bad debt condition). This wastes gas and emits misleading events for no-op operations.

When totalValueShares equals liabilityShares, the function returns 0, allowing bad debt operations to proceed with zero-value transactions. This wastes gas and emits BadDebtWrittenOff events with zero amounts, cluttering logs and potentially confusing monitoring systems.

Recommendation: Add explicit check to revert when no bad debt exists: if (totalValueShares >= liabilityShares) revert NoBadDebt();
Lido's response: Modified bad debt handling to early return 0 when no bad debt exists, and added conditional checks in calling functions to skip processing when return value is 0.

Fix review: Function now returns 0 for healthy vaults, and callers check the return value before proceeding.

I-06 - Multiple gas optimization opportunities

Description:

VaultHub contains multiple instances of redundant storage reads (SLOADs) within single functions, wasting gas on repeated access to the same storage slots. Specific locations include:

1. Lines 1273 + 1280: Same storage slot read twice
2. Lines 1329 + 1332: Duplicate SLOAD operation
3. Lines 1360/1372 + 1375: Repeated load of identical value
4. Lines 1437 + 1439: Redundant storage loads
5. Lines 1441-1442: SSTORE immediately followed by SLOAD of same slot

Each redundant SLOAD costs approximately 100 gas (for warm storage), and SSTORE-then-SLOAD patterns waste additional gas. In aggregate across hot paths and frequent operations, these optimizations could save meaningful gas for users.

Recommendation: Cache storage values in memory variables for reuse within function scope.

Lido's response: Refactored affected functions to cache storage reads in local variables and reuse cached values throughout function scope.

Fix review: Storage reads consolidated. Functions now cache values at the beginning and reuse them, eliminating redundant SLOADs. Gas savings verified through gas reporter diffs.

I-07 - Unnecessary vault check in redemption validation

Description:

VaultHub.setVaultRedemptions() contains a redundant validation check that is either always true by the time of execution or duplicates validation performed elsewhere in the call stack.

The specific check adds gas overhead without providing security value, as the condition being validated is guaranteed by invariants maintained elsewhere or by prior checks in the execution path.

Recommendation: Remove redundant check after verifying through invariant analysis that it provides no additional protection.

Lido's response: Removed redundant validation after confirming through formal verification that the condition is maintained by system invariants.

Fix review: Check removed.

I-08 - The value of fee shares may be too low

Description:

Fee calculation logic may compute fee share values that are too low in certain edge cases, particularly when dealing with small liability amounts or specific rounding scenarios. The issue arises from the order of operations in fee calculations and share-to-ETH conversions.

When fees are calculated on small liability amounts or when multiple rounding operations compound, the resulting fee shares may round down more aggressively than intended, slightly shortchanging fee recipients. While individually small, over many operations this could accumulate to non-trivial amounts.

Recommendations: Review fee calculation order of operations to minimize precision loss.

Lido's response: Acknowledged. Fee calculations use conservative rounding that slightly favors the protocol over fee recipients.

Fix review: The behavior is acceptable.

I-09 - Redundant condition - one contains the other

Description:

LazyOracle._processTotalValue() contains a redundant conditional check where one condition implies the other:

```python
if (_reportedTotalValue <= quarantineThreshold ||
```

```
    totalValueIncrease <= maxIncreaseWithRewards) {
    // Process normally
}
```

When _reportedTotalValue <= quarantineThreshold is true, the second condition totalValueIncrease <= maxIncreaseWithRewards is necessarily true due to how these values are derived. The first condition is a subset of the second, making it logically redundant.

Recommendation: Simplify conditional logic by removing the redundant clause. This improves code clarity and saves minimal gas on evaluation.

Lido's response: Redundant condition removed.

Fix review: Condition simplified to remove redundancy.

I-10 - Some functions should be external not public

Description:

Several functions in PredepositGuarantee and other contracts are declared public when they should be external, wasting gas and reducing code clarity. Specifically, proveInvalidValidatorWC and other functions are never called internally.
Same for getTotalELRewardsCollected in LidoExecutionLayerRewardsVault.sol.

Public functions generate additional code to handle potential internal calls, including copying calldata to memory. External functions optimize by directly reading from calldata, saving gas. When functions are never called internally, declaring them public wastes gas for every external invocation.

Recommendation: Change function visibility from public to external for functions that are never called internally. This provides gas savings and clearer intent.

Lido's response: Audited function visibility across contracts and changed appropriate functions from public to external.

Fix review: Functions updated to external visibility.

I-11 - Cannot set manual pause deposit flag to false if resume not allowed

Description:

VaultHub's beacon chain deposit pause mechanism prevents manually resuming deposits when certain conditions exist (redemptionShares > 0 or unsettled fees > MIN_BEACON_DEPOSIT). However, the current implementation prevents clearing the manual pause flag even when the owner wants to indicate intent to resume once conditions clear.
The owner should be able to set isBeaconDepositsManuallyPaused = false even when automatic resume is blocked, signaling intent to resume once conditions clear. This would allow automatic resumption when redemptions are settled and fees drop below threshold, without requiring additional owner transaction.

Recommendations: Allow owners to clear manual pause flag regardless of current blocking conditions. The actual deposit resume would still be gated by the automatic checks, but setting the flag enables automatic resumption once conditions improve.

Lido's response: Modified resumeBeaconChainDeposits() to allow clearing the manual pause flag even when automatic resume conditions aren't met.

Fix review: Conditions simplified and clarified.

I-12 - Burning External Shares Reverts When Current Stake Limit Equals uint96 Maximum

Description:

The burnExternalShares() function can revert when attempting to increase the stake limit beyond uint96(-1). When calculateCurrentStakeLimit() returns the maximum uint96 value, adding any stethAmount causes an overflow that fails the assertion in updatePrevStakeLimit().

Recommendations: Cap the newStakeLimit at uint96(-1) before calling updatePrevStakeLimit().

Lido's response: Fixed by limiting possible staking limit by uint96.max /2.

Fix review: Correct fix implemented here.

I-13 - Missing BLS Signature Length Validation in Predeposit Flow

Description:

The BLS12_381.verifyDepositMessage() function validates pubkey length but does not validate that the signature is exactly 96 bytes before processing. The assembly code assumes a 96-byte signature and copies data directly from calldata without bounds checking. If a malformed signature (too long or too short) passes onchain BLS verification but is rejected by beacon nodes offchain, the 1 ETH predeposit and node operator collateral would be permanently lost. The official Ethereum deposit contract explicitly validates all input lengths to prevent such issues.

Recommendations: Add explicit signature length validation in BLS12_381.verifyDepositMessage() similar to the existing pubkey check: if (signature.length != 96) revert InvalidSignatureLength().

Lido's response: Acknowledged, We will come back to it in the next version.

Fix review: Acknowledge

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.