certora

Security Assessment Report

LIDO

Lido V3

January 2026

Prepared for Lido

Table of contents

Project Summary

Project Scope

Project Name	Repository Link	Branch/PR/Commit Hash	Platform
Core	lidofinance/core/	Branch : #PR1730, Commit cfdc	On-Chain

Project Overview

This document describes the security assessment of Lido V3 Staking Infrastructure. The work was undertaken from **March 18th** to **March 24th**.

Last audited commit : 702ef9da1bc067a996614de1e3c6a53b797fa4c2.

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.

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	0	0	0
High	0	0	0
Medium	0	0	0
Low	1	1	1
Informational	2	2	1
Total	3	3	2

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
L-01	Front-Running Forced Validator Exit via Deposit and Rebalance Griefing	Low	Fixed
I-01	Forced Validator Exit Griefing Persists for Jailed Vaults with Pre-Minted stETh	Informational	Acknowledged
I-02	Unsafe Downcast of Oracle-Reported Slashing Reserve to uint128	Informational	Fixed

Low Severity Issues

L-01 – Front-Running Deposit Bypasses Partial Withdrawal Prohibition for Unhealthy Vaults

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

Description:

Following the fix that blocks all partial withdrawals when a vault has an obligations shortfall , a vault owner can bypass this prohibition by front-running with an ETH deposit that temporarily eliminates the shortfall. This re-enables the consensus layer withdrawal queue clogging attack. The partial withdrawal check at triggerValidatorWithdrawals() relies on _obligationsShortfallValue() returning a non-zero value:

Python

```
if (hasPartialWithdrawals) {
           if (_operatorGrid().isVaultInJail(_vault)) revert
PartialValidatorWithdrawalNotAllowed();
           _requireFreshReport(_vault, record);
           /// @dev NB: Disallow partial withdrawals when the vault has obligations shortfall
in order to prevent the
           ///      vault owner from clogging the consensus layer withdrawal queue by
front-running and delaying the
           ///      forceful validator exits required for rebalancing the vault.
           uint256 obligationsShortfallAmount = _obligationsShortfallValue(_vault,
connection, record);
           if (obligationsShortfallAmount > 0) {
               revert PartialValidatorWithdrawalNotAllowed();
           }
       }
```

The shortfall computation depends on _availableBalance(), which is simply address(vault).balance - stagedBalance. Since StakingVault has an unrestricted receive() function, anyone can increase the vault's available balance by sending ETH directly.
Attack scenario:

1. Vault is unhealthy with obligations shortfall; partial withdrawals are blocked
2. Vault owner sends ETH to the vault (via receive() or fund()), making _availableBalance >= obligationsAmount and shortfall becomes 0
3. Partial withdrawals are now allowed; vault owner calls triggerValidatorWithdrawals() with partial amounts, re-enabling the CL queue clogging attack (a partial on a 0x02 validator with minimal excess gets trimmed by Electra to ~1 gwei, poisoning pending_balance_to_withdraw and blocking subsequent forced full exits)
4. Vault owner back-runs with rebalance(), which consumes the deposited ETH, restoring the shortfall
5. Repeat every report cycle, indefinitely preventing forced validator exits

The same deposit front-running also blocks forceValidatorExit() directly, since it requires obligationsShortfallAmount > 0.

Recommendations: Add an additional check that prevents partial withdrawals for vaults flagged as requiring forced exits, independent of the instantaneous shortfall value.

Lido's response: Fixed in PR #1743 by adding a jail check that blocks partial withdrawals for jailed vaults regardless of shortfall status.

Fix Review: Fixed. The jail mechanism prevents the partial withdrawal bypass variant. Once the committee jails a misbehaving vault, partial withdrawals are blocked even if the vault owner deposits ETH to eliminate the shortfall, preventing CL queue clogging.

Informational Severity Issues

I-01 - Forced Validator Exit Griefing Persists for Jailed Vaults with Pre-Minted stETH

Description:

The jail mechanism (PR #1743) blocks partial withdrawals for jailed vaults but does not prevent griefing of forceValidatorExit(). A jailed vault owner with pre-minted stETH can repeatedly front-run forceValidatorExit() by depositing ETH to the vault (making obligationsShortfallAmount temporarily zero, causing revert at line 943), then back-running with forceRebalance() to consume the deposit. This indefinitely delays forced validator exits without requiring any partial withdrawals.

Recommendations: Consider allowing forceValidatorExit() to proceed for jailed vaults regardless of instantaneous shortfall.

Lido's response: Acknowledged. Fixing this might give the committee too much power. The risk can be mitigated by increasing fees and adding redemptions to accelerate debt repayment, or by upgrading the contract as a last resort.

Fix review: Acknowledged

I-02 - Unsafe Downcast of Oracle-Reported Slashing Reserve to uint128

Description:

In _applyVaultReport(), the oracle-reported _reportSlashingReserve value (uint256) is downcast to uint128 without using SafeCast, despite the contract importing it. Solidity 0.8.25 does not check explicit narrowing type conversions -- higher-order bits are silently truncated. If _reportSlashingReserve exceeds type(uint128).max, the minimalReserve would be set to an incorrect (lower) value, undermining slashing reserve enforcement.

```
_record.minimalReserve = uint128(Math256.max(CONNECT_DEPOSIT,
_reportSlashingReserve));
```

While type(uint128).max (~3.4 * 10^38 wei, ~3.4 * 10^20 ETH) far exceeds any realistic value, the inconsistency with the contract's own SafeCast import suggests the check was intended but overlooked. Similar unchecked downcasts exist for _reportCumulativeLidoFees and other fields in the Report struct.

Recommendations: Use SafeCast.toUint128() for the downcast to ensure a clean revert with a descriptive error rather than silent truncation in edge cases. Apply the same treatment to other unchecked downcasts in _applyVaultReport().

Lido's response: Fixed in #1755.

Fix review: Fixed.

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.