STATE MIND

Lido Staking Router V3

17-03-2026 - 01-07-2026

Table of contents

1. Project brief 4

2. Finding severity breakdown 6

3. Summary of findings 7

4. Conclusion 7

5. Findings report 9

Informational

- Inflated staking limit remains frozen when recovery is zero 9
- StakingRouter.topUp() does not emit an event after depositing ETH 9
- Old AccessControl storage not cleaned up after migration 10
- Incorrect storage layout comment for CL balances packed slot in Lido.sol 10
- Missing zero-value validation for maxDepositsPerBlock in module configuration 11
- DepositSecurityModule.canDeposit() can return true even when router deposit reverts with ZeroDeposits() 12
- StakingRouter.getNodeOperatorDigests() gas optimization 13
- Reserve sync check wider than reserve invariant 13
- Guardian attestation does not cover depositCalldata 14
- DepositedPostReportUpdated event declared but never emitted 14
- Precompute dummySignature root in top-up flow 15
- Unused internal helper StakingRouter._getAccountingOracle() 15
- Module active check is duplicated in router deposit path 15
- Gas optimization: cache SRStorage.getModulesCount() 15
- Unnecessary status-change flag handling in module status update flow 16

	StakingRouter.finalizeUpgrade_v4() does not enforce the previous contract version	16
	Pause intent with future block number reverts with arithmetic underflow	17
	SRLib._migrateStorage() silently skips when modules already exist	17
	delete of mapping base slots in migration cleanup is a no-op	18
	Incorrect parameter name in IncorrectRequestFinalization error	18
	Incorrect NatSpec on Accounting._distributeFee() function	19
	Simulated share rate reconstruction does not account for fee share divergence introduced by withdrawal queue finalization	20
Informational	Uninitialized _lastReportTimestamp creates inconsistent historical report metadata	21
	Unused accounting field and custom error	21
	Zero postInternalShares reverts in _simulateOracleReport() before reaching _sanityChecks()	22
	Missing explicit module existence checks in node-digest accessors	22
	Duplicate-batch detection happens after full batch validation	23
	SourceEqualsTarget error omits the source key index within the group	23
	ConsolidationMigrator keeps a stale copy of StakingRouter struct	24
	ConsolidationBus does not enforce an upper bound for executionDelay	24
	Consolidation preconditions do not enforce staking-module deposit status	25
	Rate limit reconfiguration nullifies accrued replenishment	25
	NatSpec documentation contains minor typos in ConsolidationGateway	26
	Quadratic duplicate validator index detection in TopUpGateway.topUp()	27
	Missing zero-address check on admin parameter in TopUpGateway.initialize()	28
	Activation epoch check rejects validators active in the proven epoch	28
	Stale lido.ValidatorsExitBus.maxExitRequestLimit storage not cleared on upgrade to v3	29
	Undelivered manual exit-hash entries become permanently stuck after upgrade	29
	maxValidatorsPerReport limit not enforced in oracle report path	30

Informational	Pending balance sanity check can be griefed by unaccounted direct top-up deposits	30
	TopUpGateway updates top-up rate-limit state even when no deposit is performed	31
	ValidatorsExitBus misses zero module validation during exit balance calculation	31
	Lido.collectRewardsAndProcessWithdrawals() ignores actual amount withdrawn from EL rewards vault	32
	ValidatorsExitBus._getTimestamp() returns uint32, will overflow around year 2106	32
	StakingRouter.topUp() can mutate module state while Lido cannot deposit	33

1. Project brief

Title	Description
Client	Lido
Project name	Lido Staking Router V3
Timeline	17-03-2026 – 01-07-2026

Project Log

Repository	https://github.com/lidofinance/core	
Date	**Commit Hash**	**Note**
23-03-2026	4920732c74010de7d4ba95cc950e78403d1d5442	Initial Commit
15-06-2026	2d089ef353067ab4ff493c49c1612aca85a3fec2	Reaudit commit
22-06-2026	a47dc0bb67f7bc8431d92e60e2e312bc29403232	Reaudit commit 2
29-06-2026	e21fe18fd9159f6c06e5d035168d18e6ba62b75e	Reaudit commit 3 with changes in comments
01-07-2026	af095e48bbc1c3841c2c9936219c8461af01056b	Reaudit commit 4

Short Overview

Staking Router V3 is a protocol-wide upgrade that prepares Lido for Max Effective Balance after Pectra/EIP-7251. It replaces the old "validator count × 32 ETH" assumptions with balance-based accounting, so the protocol can correctly handle modules using 0x02 withdrawal credentials and validators with balances up to 2048 ETH. The upgrade also adds the infrastructure for validator top-ups, CM V1 to CM V2 consolidation, deposit reserve based stake rebalancing, and updated exit/reporting logic.

The main features introduced in the audited scope include:

- Balance-based accounting in Lido, Accounting, and Staking Router, replacing clBalance / clValidators assumptions with clValidatorsBalance, clPendingBalance, and balance-based deposit tracking.
- Updated Accounting Oracle reports, including clValidatorsBalanceGwei, clPendingBalanceGwei, per-module validator balances, and consistency checks based on ETH amounts rather than validator counts.
- Pull-based deposits, where the Staking Router pulls depositable ETH from Lido and handles both 32 ETH predeposits and 0x02 validator top-ups.
- A new TopUpGateway flow that verifies validator state with Merkle proofs, calculates per-validator top-up limits, and routes eligible top-ups through the Staking Router.
- Deposit reserve / fast-lane mechanics to protect a configurable portion of buffered ETH for CL deposits and improve stake rebalancing across modules.

- Consolidation support for CM V1 to CM V2 migration, including EasyTrack approval, ConsolidationMigrator, ConsolidationBus, ConsolidationGateway, and WithdrawalVault support for EIP-7251 requests.
- Validator exit updates for MaxEB, including a new VEBO report format with key indices and ETH-denominated sanity limits.
- VEBO logic updates to account for validator balances, consolidation state, fast-lane deposits, CM V2 operator weights, and migrated CM V1/CM V2 operator relationships.

Project Scope

The audit covered the following files:

- Lido.sol
- Accounting.sol
- DepositSecurityModule.sol
- LidoLocator.sol
- WithdrawalVault.sol
- WithdrawalVaultEIP7685.sol
- AccountingOracle.sol
- ValidatorsExitBus.sol
- ValidatorsExitBusOracle.sol
- OracleReportSanityChecker.sol
- CLValidatorVerifier.sol
- TopUpGateway.sol
- ConsolidationBus.sol
- ConsolidationGateway.sol
- ConsolidationMigrator.sol
- BeaconChainDepositor.sol
- SRLib.sol
- SRStorage.sol
- SRTypes.sol
- SRUtils.sol
- StakingRouter.sol
- RateLimit.sol
- WithdrawalCredentials.sol

2. Finding severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds to be transferred to any party.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss of funds.
Informational	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Client regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Client is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

3. Summary of findings

Severity	# of Findings
Critical	0 (0 fixed, 0 acknowledged)
High	0 (0 fixed, 0 acknowledged)
Medium	0 (0 fixed, 0 acknowledged)
Informational	45 (33 fixed, 12 acknowledged)
Total	45 (33 fixed, 12 acknowledged)

4. Conclusion

During the audit of the codebase, 45 issues were found in total:

- 45 informational severity issues (33 fixed, 12 acknowledged)

The final reviewed commit is af095e48bbc1c3841c2c9936219c8461af01056b

Deployment

All deployments were successfully validated. The deployed bytecode of each contract within the audited scope matches the bytecode produced from the final audited commit: af095e48bbc1c3841c2c9936219c8461af01056b.

Contract	Address
Accounting (Implementation)	0x3aa937Ac2ab89CDd363EdC6b5A4d4A42dF5bc043
AccountingOracle (Implementation)	0xe4f03D1107d1905B6F2A28FCb6Af221E0CE19136
Lido (Implementation)	0x028271E30a695c0527A0C50cA30603feD004cDb0
BeaconChainDepositor	0xf98AC162eAB766bDB9507c3584c00C535B8F6216
ConsolidationBus (Proxy)	0xd907CE33B4Be423823d1CFFe80BD147E8b8554C8

ConsolidationBus (Implementation)	0xFfDe8Acab9D7037f29198Ad03ad6d05bac8B0a2E
ConsolidationGateway	0x17be979344f2c2cC806229a532D92f8742C10462
ConsolidationMigrator (Proxy)	0x9Dc70b5A4f4F5E4AF9058C983D560564F031f1D7
ConsolidationMigrator (Implementation)	0x6Fb4c152F092373dD71f0C07C83c1E77406599aB
DepositSecurityModule	0xF573E9E3de1f86B085417ab294f56E7920B4e9Be
LidoLocator (Implementation)	0x0360002bf51DCae1c0267aE0AFDaBacAF7De686b
MinFirstAllocationStrategy	0x98F1da239a199574A4F7f371fD8fc0a872022c73
OracleReportSanityChecker	0x147f8d3cf3004FAf9Bf94E88B54b6C06De507be9
SRLib	0xc0be9942Fd8f54aB126A5F0Ba649A90049ccad14
StakingRouter (Implementation)	0xDD76927045435C7605cf6f5F978cfb8CABDb5F80
TopUpGateway (Proxy)	0x3FC2C71579D80790Aaa3fc7Be8B66ac39dC57374
TopUpGateway (Implementation)	0xb08dBc68C521cD7A4318dc4C807a42bEB20f1106
ValidatorsExitBusOracle (Implementation)	0x2C3386b39db89eef0F362A3BE0C05a6811E809E3
WithdrawalVault (Implementation)	0xfB4521BD151BFB45DB6045D2d07e58e0f597e340

5. Findings report

INFORMATIONAL-01	Inflated staking limit remains frozen when recovery is zero	Acknowledged

Description

Line: Lido.sol#L948

The staking-limit accounting may retain an inflated value above the configured ceiling when recovery is disabled. In **Lido.burnExternalShares()**, the contract restores staking capacity through **Lido._increaseStakingLimit()**, which adds the burned amount to the current limit and stores the result as **prevStakeLimit** without constraining it to **maxStakeLimit**.

When **Lido.setStakingLimit()** is configured with **_stakeLimitIncreasePerBlock = 0**, **StakeLimitUtils.calculateCurrentStakeLimit()** derives a zero recovery rate. If **prevStakeLimit** has already been increased above **maxStakeLimit**, the function continues returning that inflated value because the decay branch subtracts zero and then takes **max(prevStakeLimit, maxStakeLimit)**. Under this configuration, the current staking limit may therefore remain frozen above the configured maximum instead of returning to the intended cap.

Recommendation

We recommend ensuring that the zero-recovery configuration preserves **maxStakeLimit**. This could be achieved by normalizing the derived current limit when automatic recovery is disabled.

Client's comments

> We are not going to fix this issue right now. Instead, we will create an issue in the Core repo and address it separately if changes are necessary.

INFORMATIONAL-02	StakingRouter.topUp() does not emit an event after depositing ETH	Fixed at: 26d2366

Description

The **StakingRouter.topUp()** function pulls ETH from Lido and makes beacon chain top-up deposits, but does not emit any event upon completion. In contrast, **StakingRouter.deposit()** calls **StakingRouter._updateModuleLastDepositState()**, which emits **StakingRouterETHDeposited** with the module ID and deposited value.

Impact: Off-chain services and monitoring tools have no way to track top-up deposits through events, which may affect observability of ETH flows through the Staking Router.

Recommendation

We recommend emitting an event at the end of **StakingRouter.topUp()** to maintain consistency with **StakingRouter.deposit()** and improve off-chain observability.

INFORMATIONAL-03	Old AccessControl storage not cleaned up after migration	Acknowledged

Description

Lines: StakingRouter.sol#L146-L153

The **StakingRouter.finalizeUpgrade_v4()** function migrates role data from the old **_roleMembers** mapping at **keccak256("openzeppelin.AccessControlEnumerable._roleMembers")** to the new ERC-7201 namespaced storage by iterating over all 9 roles and calling **_grantRole()** for each member. However, after the migration completes, neither the old **_roleMembers** mapping nor the old **_roles** mapping at **keccak256("openzeppelin.AccessControl._roles")** are cleared. This leaves stale role data permanently in contract storage. While no code in the upgraded contract references these old slot positions, a future upgrade or new library could inadvertently resolve to the same storage slots, interpreting outdated role assignments as authoritative and potentially granting unintended access.

Impact: Residual stale role data at well-known OpenZeppelin storage positions may cause confusion or unintended behavior if a future upgrade inadvertently references the old slots.

Recommendation

We recommend clearing old storage at slots **keccak256("openzeppelin.AccessControl._roles")** and **keccak256("openzeppelin.AccessControlEnumerable._roleMembers")**.

Client's comments

> We do not expect that we will switch back to the old access control, which is why we decided to skip the clean-up.

INFORMATIONAL-04	Incorrect storage layout comment for CL balances packed slot in Lido.sol	Fixed at: 557a7b6

Description

Line: Lido.sol#L140

The NatSpec comment documenting the packed storage layout of **CL_VALIDATORS_BALANCE_AND_CL_PENDING_BALANCE_POSITION** incorrectly describes the positions of the two fields within the 256-bit slot:

```
/// |----- 128 bit ------------|------ 128 bit -------|
/// | CL validators balance | CL pending balance |
```

This comment indicates that **CL validators balance** occupies the high 128 bits and **CL pending balance** occupies the low 128 bits. However, the actual getter and setter functions **Lido._getClValidatorsBalanceAndClPendingBalance()** and **Lido._setClValidatorsBalanceAndClPendingBalance()** use **UnstructuredStorageExt.getLowAndHighUint128()** and **UnstructuredStorageExt.setLowAndHighUint128()** respectively, which place **clValidatorsBalance** in the low 128 bits and **clPendingBalance** in the high 128 bits. This is the opposite of what the comment states.

Recommendation

We recommend correcting the comment to match the actual storage layout:

```
/// @dev CL validators balance and deposited balance since last report
/// |----- 128 bit ------------|------ 128 bit -------|
- /// | CL validators balance | CL pending balance |
+ /// | CL pending balance | CL validators balance |
```

INFORMATIONAL-05	Missing zero-value validation for maxDepositsPerBlock in module configuration	Fixed at: a388fdb

Description

Line: SRLib.sol#L264

In **SRLib._updateModuleParams()**, the **_maxDepositsPerBlock** parameter is validated only against the upper bound (**type(uint64).max**), but zero is accepted as a valid value:

```
if (_maxDepositsPerBlock > type(uint64).max) revert ISRBase.InvalidMaxDepositPerBlockValue();
```

This is inconsistent with the adjacent **_minDepositBlockDistance** parameter, which explicitly rejects zero:

```
if (_minDepositBlockDistance == 0 || _minDepositBlockDistance > type(uint64).max) {
    revert ISRBase.InvalidMinDepositBlockDistance();
}
```

When **maxDepositsPerBlock** is set to zero for a module, the **StakingRouter.deposit()** function computes **maxDepositsCount** as **Math.min(Math.min(state.deposits.maxDepositsPerBlock, depositableValidatorsCount), ...)**, which evaluates to zero, causing all deposit calls for that module to revert with **ZeroDeposits()**. This effectively blocks the WC 0x01 deposit path for the module without any explicit warning at configuration time.

Recommendation

We recommend adding a zero-value check for **_maxDepositsPerBlock** consistent with the existing validation for **_minDepositBlockDistance**:

```
- if (_maxDepositsPerBlock > type(uint64).max) revert ISRBase.InvalidMaxDepositPerBlockValue();
+ if (_maxDepositsPerBlock == 0 || _maxDepositsPerBlock > type(uint64).max) revert
ISRBase.InvalidMaxDepositPerBlockValue();
```

INFORMATIONAL-06	DepositSecurityModule.canDeposit() can return true even when router deposit reverts with ZeroDeposits()	Fixed at: 7b199d1

Description

Lines:

- DepositSecurityModule.sol#L405-L425

Description

Current codebase no longer has **Lido.deposit()**, and real path in **DepositSecurityModule.depositBufferedEther()** goes through **StakingRouter.deposit()**. Also, real router deposit path requires non-zero **maxDepositsCount**. If count is zero, call reverts with **ZeroDeposits()**.

Impact: **DepositSecurityModule.canDeposit()** can return **true** while fresh attested call to **DepositSecurityModule.depositBufferedEther()** still cannot work in current state. For example, Lido has no depositable ether, or chosen module has zero allocatable deposit capacity.

Recommendation

We recommend making **DepositSecurityModule.canDeposit()** also check whether current state gives non-zero deposit allocation for module and fix comment block.

```
 /**
- * @notice Returns whether LIDO.deposit() can be called, given that the caller
+ * @notice Returns whether StakingRouter.deposit()`can be called, given that the caller
```

```
 interface ILido {
+    function getDepositableEther() external view returns (uint256);
 }

 interface IStakingRouter {
+    function getStakingModuleMaxDepositsCount(uint256 _stakingModuleId, uint256 _maxDepositsValue)
+        external
+        view
+        returns (uint256);
 }

 function canDeposit(uint256 stakingModuleId) external view returns (bool) {
     if (!STAKING_ROUTER.canDeposit(stakingModuleId)) return false;

     bool isDepositDistancePassed = _isMinDepositDistancePassed(stakingModuleId);
     bool isLidoCanDeposit = LIDO.canDeposit();
+    uint256 depositableEther = LIDO.getDepositableEther();
+    bool hasModuleDeposits =
+        STAKING_ROUTER.getStakingModuleMaxDepositsCount(stakingModuleId, depositableEther) > 0;

-    return (!isDepositsPaused && quorum > 0 && isDepositDistancePassed && isLidoCanDeposit);
+    return (!isDepositsPaused && quorum > 0 && isDepositDistancePassed && isLidoCanDeposit &&
hasModuleDeposits);
 }
```

Client's comments

Method removed

INFORMATIONAL-07	StakingRouter.getNodeOperatorDigests() gas optimization	Fixed at: 3d82cfa

Description

Lines: StakingRouter.sol#L534-L536

Description **StakingRouter.getNodeOperatorDigests()**

StakingRouter.getNodeOperatorDigests() resolves **IStakingModule stakingModule = _stakingModuleId.getIStakingModule()** inside **_nodeOperatorIds** loop, even though **_stakingModuleId** stays constant for whole call.

Recommendation

We recommend reusing cached **stakingModule** for every iteration.

```
    digests = new NodeOperatorDigest[](_nodeOperatorIds.length);
+    IStakingModule stakingModule = _stakingModuleId.getIStakingModule();
    for (uint256 i = 0; i < _nodeOperatorIds.length; ++i) {
        uint256 nodeOperatorId = _nodeOperatorIds[i];
-        IStakingModule stakingModule = _stakingModuleId.getIStakingModule();

        digests[i].id = nodeOperatorId;
```

INFORMATIONAL-08	Reserve sync check wider than reserve invariant	Fixed at: 23a77a6

Description

Line: Lido.sol#L1115-L1116

Lido._updateBufferedEtherAllocation() uses **depositsReserve != depositsReserveTarget** before restoring reserve to target. Surrounding reserve flow seems to allow only two reachable states before and after report processing: **depositsReserve == depositsReserveTarget** or **depositsReserve < depositsReserveTarget**, while **>** case sits outside intended invariant.

Recommendation

We recommend tightening condition so code matches invariant.

```
-if (depositsReserve != depositsReserveTarget) {
+if (depositsReserve < depositsReserveTarget) {
_setDepositsReserve(depositsReserveTarget);
```

INFORMATIONAL-09	Guardian attestation does not cover depositCalldata	Fixed at: f315b45

Description

Lines: DepositSecurityModule.sol#L527

In **DepositSecurityModule.depositBufferedEther**, guardians sign over **(ATTEST_MESSAGE_PREFIX, blockNumber, blockHash, depositRoot, stakingModuleId, nonce)**, but **depositCalldata** is not part of signed message. Any relayer holding valid guardian signatures can submit arbitrary **depositCalldata** to staking module.

```
bytes32 msgHash = keccak256(
    abi.encodePacked(ATTEST_MESSAGE_PREFIX, blockNumber, blockHash, depositRoot, stakingModuleId, nonce)
);
// depositCalldata NOT in the hash
```

depositCalldata is then forwarded directly into module call:

```
IStakingModule(stakingModuleAddress).obtainDepositData(maxDepositsCount, _depositCalldata);
```

Recommendation

We recommend including a hash of **depositCalldata** in guardian attestation message, so guardians also attest to specific deposit parameters being used.

Client's comments

> We removed **depositCalldata** from DSM, but kept it for compatibility.

INFORMATIONAL-10	DepositedPostReportUpdated event declared but never emitted	Fixed at: 46102df

Description

Lines:

- Lido.sol#L326
- Lido.sol#L1011

1. **_spendDepositableEther()** updates **depositedPostReport** via **_setBufferedEtherAndDepositedPostReport(...)**, but only emits **Unbuffered**.
2. **processClStateUpdate()** resets **depositedPostReport** via **_setDepositedPostReport(depositedNextReport)**, but also emits no **DepositedPostReportUpdated**.

Recommendation

We recommend emitting **DepositedPostReportUpdated** after every modification of **depositedPostReport**: in **_spendDepositableEther()** and in **processClStateUpdate()**.

INFORMATIONAL–11	Precompute dummySignature root in top-up flow	Fixed at: e260f05

Description

Line: BeaconChainDepositor.sol#L103

BeaconChainDepositor.makeBeaconChainTopUp() uses the same **dummySignature** (**new bytes(96)**, all zeros) for every deposit in the loop, but **BeaconChainDepositor._computeDepositDataRootWithAmount()** recomputes its **signatureRoot** on each iteration.

Recommendation

We recommend precomputing and reusing the root for the zeroed dummy signature.

INFORMATIONAL–12	Unused internal helper StakingRouter._getAccountingOracle()	Fixed at: 8c4843c

Description

Line: StakingRouter.sol#L1132-L1134

StakingRouter._getAccountingOracle() is defined but never referenced.

Recommendation

We recommend removing the **StakingRouter._getAccountingOracle()** function.

INFORMATIONAL–13	Module active check is duplicated in router deposit path	Fixed at: dbba130

Description

Line: StakingRouter.sol#L926

DepositSecurityModule.depositBufferedEther() already checks module activity through **StakingRouter.canDeposit()** before calling **StakingRouter.deposit()**. **StakingRouter.deposit()** then checks the same condition again.

Recommendation

We recommend keeping this check in one place to save gas.

INFORMATIONAL–14	Gas optimization: cache SRStorage.getModulesCount()	Fixed at: a2abba5

Description

Lines:

- 0.8.25/sr/SRLib.sol#L193
- 0.8.25/sr/SRLib.sol#L201

In **SRLib._addModule()**, the **modulesCount** variable saves the **SRStorage.getModulesCount()** value on the second **SLOAD**.

Recommendation

We recommend saving the module count value on a first read.

INFORMATIONAL-15	Unnecessary status-change flag handling in module status update flow	Fixed at: 59ba5ec

Description

Line: 0.8.25/sr/SRLib.sol#L365-L368

SRLib._setModuleStatus() currently returns a boolean **isChanged** and conditionally updates storage, while **StakingRouter.setStakingModuleStatus()** immediately converts the **false** path into a revert. While the **SRLib._notifyStakingModulesOfWithdrawalCredentialsChange** function cannot receive a **false** flag value (sets pause for an active module).

This creates an avoidable boolean-return branch and duplicates decision logic across the library and the caller instead of enforcing the invariant at the source.

Recommendation

We recommend replacing the flag-based pattern with direct revert behavior inside **SRLib._setModuleStatus()** and simplifying **StakingRouter.setStakingModuleStatus()** to call **SRLib._setModuleStatus()** without a boolean check.

INFORMATIONAL-16	StakingRouter.finalizeUpgrade_v4() does not enforce the previous contract version	Fixed at: da65aa8

Description

Lines:

- StakingRouter.sol#L115
- StakingRouter.sol#L123

The previous version of the contract used the custom **Versioned** base, where **StakingRouter.initialize()** called **Versioned._initializeContractVersionTo(3)** and **StakingRouter.finalizeUpgrade_v3()** explicitly called **Versioned._checkContractVersion(2)**, making the two entry points mutually exclusive and pinning each to a specific pre-state.

The new contract switches to OpenZeppelin's **Initializable** and guards both **StakingRouter.initialize()** and **StakingRouter.finalizeUpgrade_v4()** with **reinitializer(4)**. The **reinitializer(version)** modifier only checks that the current **_initialized** value is strictly less than the requested version. It does not enforce any specific lower bound. As a result, the two entry points are no longer distinguishable by their pre-state:

Impact

- On a v3 proxy, **StakingRouter.initialize()** can be called instead of **StakingRouter.finalizeUpgrade_v4()**, overwriting state and skipping the migration entirely.
- On a freshly deployed proxy, **StakingRouter.finalizeUpgrade_v4()** can be called instead of **StakingRouter.initialize()**. The legacy-slot reads inside **SRLib._migrateStorage()** then run against zero values and finalize the proxy at version 4 with no admin and zero withdrawal credentials.

Recommendation

We recommend that **StakingRouter.finalizeUpgrade_v4()** require the legacy version slot to equal **3**, and that **StakingRouter.initialize()** require it to equal **0**.

INFORMATIONAL-17	Pause intent with future block number reverts with arithmetic underflow	Acknowledged

Description

Line: DepositSecurityModule.sol#L387

In **DepositSecurityModule.pauseDeposits()** the validity-period check is implemented as **block.number - blockNumber > pauseIntentValidityPeriodBlocks**. When a guardian (or a relayer using a guardian-signed message) supplies a **blockNumber** that is greater than the current **block.number**, the unsigned subtraction reverts due to underflow.

Recommendation

We recommend rejecting future block numbers explicitly with a dedicated error so the comparison cannot underflow:

```
+ error InvalidPauseIntentBlockNumber();
...
+ if (block.number < blockNumber) revert InvalidPauseIntentBlockNumber();
  if (block.number - blockNumber > pauseIntentValidityPeriodBlocks) revert PauseIntentExpired();
```

Client's comments

The underflow error is expected. We do not plan to change the current logic or add too many validations for all possible invalid inputs.

INFORMATIONAL-18	SRLib._migrateStorage() silently skips when modules already exist	Fixed at: bfc40df

Description

Line: SRLib.sol#L56-L58

SRLib._migrateStorage() early-returns silently when **SRStorage.getModulesCount() > 0** instead of reverting. If the per-module migration is ever skipped due to unexpected pre-state, the surrounding **finalizeUpgrade_v4()** steps still run, leaving the contract in a partially-migrated state with no on-chain signal that something went wrong.

Recommendation

We recommend converting the silent no-op into an explicit revert so any unexpected pre-state aborts the upgrade:

```
-       if (SRStorage.getModulesCount() > 0) {
-           return;
-       }
+       if (SRStorage.getModulesCount() > 0) revert AlreadyMigrated();
```

INFORMATIONAL-19	delete of mapping base slots in migration cleanup is a no-op	Fixed at: c44ca44

Description

Lines: SRLib.sol#L148-L149

After iterating over the legacy modules mapping, **_migrateStorage()** attempts to clean up the legacy mapping storage with:

```
delete STAKING_MODULES_MAPPING_POS.getBytes32Slot().value;
delete STAKING_MODULE_INDICES_POS.getBytes32Slot().value;
```

For Solidity **mapping** types, the slot returned by the canonical position is unused: every entry is stored at **keccak256(abi.encode(key, slot))**.

Recommendation

We recommend removing the redundant deletes:

```
- delete STAKING_MODULES_MAPPING_POS.getBytes32Slot().value;
- delete STAKING_MODULE_INDICES_POS.getBytes32Slot().value;
```

INFORMATIONAL-20	Incorrect parameter name in IncorrectRequestFinalization error	Fixed at: b368435

Description

Line: OracleReportSanityChecker.sol#L1490

In **OracleReportSanityChecker._checkLastFinalizableId()**, the **IncorrectRequestFinalization** error is declared with a parameter named **requestCreationBlock**, but is emitted with **statuses[0].timestamp**. The **WithdrawalRequestStatus** struct exposes no block number field, so the name is incorrect.

Impact: Off-chain tooling decoding this revert via ABI may misinterpret the timestamp as a block number, producing incorrect output in monitoring or alerting systems.

Recommendation

We recommend renaming the error parameter to reflect the actual value:

```
- error IncorrectRequestFinalization(uint256 requestCreationBlock);
+ error IncorrectRequestFinalization(uint256 requestCreationTimestamp);
```

INFORMATIONAL-21	Incorrect NatSpec on Accounting._distributeFee() function	Fixed at: e4dcd4d

Description

Line: Accounting.sol#L454

The **@dev** NatSpec for **Accounting._distributeFee()** says:

"mints protocol fees to the treasury and node operators and calls back to stakingRouter"

Both claims are inaccurate. The actual mint (**LIDO.mintShares**) and the **stakingRouter.reportRewardsMinted()** callback both occur in the caller function, **Accounting._applyOracleReportContext()**.

Recommendation

We recommend updating the NatSpec to accurately describe the function's behavior:

```
- /// @dev mints protocol fees to the treasury and node operators and calls back to stakingRouter
+ /// @dev transfers pre-minted fee shares to the treasury and node operator recipients
  function _distributeFee(FeeDistribution memory _feeDistribution) internal {
```

INFORMATIONAL-22	Simulated share rate reconstruction does not account for fee share divergence introduced by withdrawal queue finalization	Acknowledged

Description

Lines:

- Accounting.sol#L210
- Accounting.sol#L312
- OracleReportSanityChecker.sol#L841

The accounting flow computes **simulatedShareRate** off-chain by invoking **Accounting.simulateOracleReport()** with an empty **withdrawalFinalizationBatches** array (so that **etherToFinalizeWQ == 0** and **sharesToFinalizeWQ == 0**), derives the withdrawal batches from that rate, and then submits the real report, where **_simulateOracleReport()** runs again with non-zero WQ values. After real-report execution, **OracleReportSanityChecker.checkSimulatedShareRate()** attempts to reconstruct the simulation state from the real-report state by adding **etherToFinalizeWQ** back to **postInternalEther** and **sharesToBurnForWithdrawals** back to **postInternalShares**.

The reconstruction compensates for the ether locked on the WQ and the shares burnt for withdrawals, but it does not compensate for the divergence in **sharesToMintAsFees** between the two runs. In **Accounting._calculateTotalProtocolFeeShares()** the fee shares depend on **postInternalEther** through the denominator **postInternalEther – feeEther**, and **postInternalEther** differs between the simulation and the real run by exactly **etherToFinalizeWQ**. Because the simulation sees a larger **postInternalEther**, it mints fewer fee shares than the real execution, and the gap (**sharesToMintAsFees_real – sharesToMintAsFees_sim**) is not added back in the reconstruction. As a result, the **actualShareRate** recomputed inside **OracleReportSanityChecker._checkSimulatedShareRate()** is strictly lower than the internal share rate the oracle actually observed during off-chain simulation, and **simulatedShareRate** supplied by the oracle is compared against a value that is not equivalent to the one it was derived from.

Impact: Honest oracle submissions may trigger **IncorrectSimulatedShareRate()** when **etherToFinalizeWQ** is sizeable relative to **postInternalEther**, causing the report to revert. The **simulatedShareRateDeviationBPLimit** value is intended to allow small differences between **simulatedShareRate** and the actual share rate that come from new user deposits and **stETH** minting happening between the reference slot and the moment the report is submitted. The fee share mismatch described above always shifts **actualShareRate** in one direction regardless of user activity, so a portion of the allowed deviation is already used up by this fixed shift before any real user activity is taken into account, and the check tolerates less actual deviation than the configured limit suggests.

Recommendation

We recommend passing the real-run **sharesToMintAsFees** (and the simulated fee shares, or the components needed to recompute them) to **OracleReportSanityChecker.checkSimulatedShareRate()**.

Client's comments

simulatedShareRate in **Accounting.simulateOracleReport()** used in WQ **prefinalize()** is only an upper bound for withdrawal finalization. Therefore **etherToFinalizeWQ** is always less than or equal to **sharesToFinalizeWQ * simulatedShareRate / 1e27**.

If all finalized requests are discounted at exactly the submitted simulated rate, then the WQ ether and shares are reduced proportionally, and the fee-share calculation remains aligned with the no-WQ simulation.

If requests are finalized nominally, **etherToFinalizeWQ** may be lower than **sharesToFinalizeWQ * simulatedShareRate / 1e27**, because those requests were created at a lower historical share rate. This changes the proportionality, but it is not the fee-divergence described in the finding.

A discounted finalization means the current simulated share rate is below the request's historical share rate. That reflects a loss in share rate relative to that request, but it does not by itself guarantee that the current oracle report mints no fees. Fee minting is skipped only when the current report is not profitable according to **unifiedClBalance <= principalClBalance**.

INFORMATIONAL-23	Uninitialized _lastReportTimestamp creates inconsistent historical report metadata	Acknowledged

Description

Lines:

- OracleReportSanityChecker.sol#L237
- OracleReportSanityChecker.sol#L532-L534
- OracleReportSanityChecker.sol#L1114-L1116

OracleReportSanityChecker._lastReportTimestamp is declared but never initialized in **OracleReportSanityChecker.constructor()** and is also used as-is during **OracleReportSanityChecker.migrateBaselineSnapshot()**, so baseline snapshots are recorded with timestamp 0 instead of a real migration-time value.

Impact: The CL decrease window logic may continue to operate correctly because it relies on relative elapsed time, but on-chain historical report metadata may not accurately represent real report time.

Recommendation

We recommend initializing **OracleReportSanityChecker._lastReportTimestamp** to a real baseline time during deployment or migration.

Client's comments

> The only issue is that the name **timestamp** may mislead an external reader. If someone expects it to contain a real Unix timestamp, they will see **0** and assume the data is historically incorrect. But for the validation itself, this is not a bug, because the algorithm depends on time differences rather than the absolute date.

INFORMATIONAL-24	Unused accounting field and custom error	Fixed at: 64e650a

Description

Lines:

- Accounting.sol#L56
- AccountingOracle.sol#L77

The **PreReportState** struct declared in the **Accounting** contract contains a **depositedValidators** field that is never written or read throughout the contract.

AccountingOracle declares **InvalidClBalancesData()**, but the contract never reverts with this custom error.

Recommendation

We recommend removing mentioned unused entities.

INFORMATIONAL-25	Zero postInternalShares reverts in _simulateOracleReport() before reaching _sanityChecks()	Fixed at: 7f2cbd9

Description

Lines:

- Accounting.sol#L228
- Accounting.sol#L419

Accounting._sanityChecks() includes an explicit check **if (_update.postInternalShares == 0) revert InternalSharesCantBeZero();** intended to surface a dedicated error when the post-report internal share count reaches zero. The check is placed inside **Accounting._applyOracleReportContext()**, which **Accounting.handleOracleReport()** calls only after **Accounting._simulateOracleReport()** has already produced the **CalculatedValues** struct consumed by the sanity checks.

The issue is that **Accounting._simulateOracleReport()** itself performs **postExternalShares * update.postInternalEther / update.postInternalShares** while computing **update.postTotalPooledEther**. When **postInternalShares == 0**, this division reverts before **_sanityChecks()** is reached, so the **InternalSharesCantBeZero** error is never emitted.

Recommendation

We recommend moving the zero-share check into **Accounting._simulateOracleReport()** so that it runs before the division.

INFORMATIONAL-26	Missing explicit module existence checks in node-digest accessors	Fixed at: 79f9636

Description

Lines:

- StakingRouter.sol#L499
- StakingRouter.sol#L516

StakingRouter.getAllNodeOperatorDigests() and the three-argument overload of **StakingRouter.getNodeOperatorDigests()** resolve the staking module address via **SRStorage.getIStakingModule()** (invoked through the **using** directive as **_stakingModuleId.getIStakingModule()**) and immediately forward it to **StakingRouter._getStakingModuleNodeOperatorsCount()** / **StakingRouter._getStakingModuleNodeOperatorIds()** without first invoking **SRUtils._requireModuleIdExists()**. For an unknown **_stakingModuleId**, the storage lookup behind **SRStorage.getIStakingModule()** returns the default zero address, and the subsequent external call to the zero address reverts before the protocol-specific **StakingModuleUnregistered** error is reached. Sibling accessors such as the two-argument overload of **StakingRouter.getNodeOperatorDigests()** at StakingRouter.sol#L527-L532 and **StakingRouter.getAllStakingModuleDigests()** at StakingRouter.sol#L480-L482 apply the existence check before any module interaction.

Recommendation

We recommend adding an explicit **SRUtils._requireModuleIdExists(_stakingModuleId)** call at the start of both accessors so the failure mode for unknown module ids is uniform across the contract.

INFORMATIONAL-27	Duplicate-batch detection happens after full batch validation	Fixed at: 91dd2da

Description

Line: ConsolidationBus.sol#L367

In **ConsolidationBus.addConsolidationRequests()**, the duplicate-batch check at ConsolidationBus.sol#L367 runs only after the full per-group and per-pubkey validation loop at ConsolidationBus.sol#L343-L361 has completed. When a publisher resubmits a batch that is already tracked in **_pendingBatches**, the transaction traverses every group, verifies each source and target pubkey length and computes **keccak256** over each source pubkey.

Impact: The ordering is suboptimal for gas consumption in the duplicate-submission case. Because the batch hash depends only on the calldata layout of **groups**, it can be derived before per-element validation, and an earlier duplicate check would allow the call to revert without performing work that will be discarded.

Recommendation

We recommend computing **batchHash** and performing the **_pendingBatches[batchHash].publisher** check earlier in the function — after the cheap structural checks (**EmptyBatch**, **TooManyGroups**, **BatchTooLarge**) but before the per-group hashing loop.

INFORMATIONAL-28	SourceEqualsTarget error omits the source key index within the group	Fixed at: eb00083

Description

Line: ConsolidationBus.sol#L358

In **ConsolidationBus.addConsolidationRequests()**, the validation loop checks each source pubkey against the group's target and reverts with **SourceEqualsTarget(i)** using only the group index, while the inner source key index **j** is discarded:

```
for (uint256 j = 0; j < group.length; ++j) {
    bytes calldata sourcePubkey = group[j];
    if (sourcePubkey.length != PUBKEY_LENGTH) {
        revert InvalidSourcePubkeyLength(i, j, sourcePubkey.length); // <- reports both indices
    }

    if (keccak256(sourcePubkey) == targetHash) {
        revert SourceEqualsTarget(i); // <- only group index, source index j is lost
    }
}
```

The **InvalidSourcePubkeyLength** error in the same loop correctly reports both **groupIndex** and **sourceIndex**, making the omission inconsistent. The error definition accepts a single **uint256 index** and its NatSpec describes it as "Index of the invalid pair in the batch," which is ambiguous given the two-level nesting.

Recommendation

We recommend adding the source index to the error signature to match the pattern established by **InvalidSourcePubkeyLength**:

```
- error SourceEqualsTarget(uint256 index);
+ error SourceEqualsTarget(uint256 groupIndex, uint256 sourceIndex);
```

```
- revert SourceEqualsTarget(i);
+ revert SourceEqualsTarget(i, j);
```

INFORMATIONAL-29	ConsolidationMigrator keeps a stale copy of StakingRouter struct	Fixed at: e4a0b44

Description

Lines: ConsolidationMigrator.sol#L15-L33

ConsolidationMigrator redefines its own **IStakingRouter.StakingModule** struct instead of importing **SRTypes.sol**. The local copy omits the trailing **validatorsBalanceGwei** field.

Recommendation

We recommend importing the shared **StakingModule** definition from **contracts/0.8.25/sr/SRTypes.sol**.

```
+import {StakingModule} from "contracts/0.8.25/sr/SRTypes.sol";

 interface IStakingRouter {
-    struct StakingModule {
-        ...
-    }
-
     function getStakingModule(uint256 _stakingModuleId) external view returns (StakingModule memory);
 }

-    IStakingRouter.StakingModule memory sm = STAKING_ROUTER.getStakingModule(moduleId);
+    StakingModule memory sm = STAKING_ROUTER.getStakingModule(moduleId);
```

INFORMATIONAL-30	ConsolidationBus does not enforce an upper bound for executionDelay	Acknowledged

Description

Line: ConsolidationBus.sol#L431

ConsolidationBus._setExecutionDelay() accepts any **uint256** value and stores it without bounds. A **MANAGE_ROLE** misconfiguration may freeze or severely delay execution of all pending batches until governance corrects the parameter.

Recommendation

```
+error ExecutionDelayTooLarge(uint256 delay, uint256 maxDelay);
+uint256 internal constant MAX_EXECUTION_DELAY = 30 days;

 function _setExecutionDelay(uint256 delay) internal {
+    if (delay > MAX_EXECUTION_DELAY) revert ExecutionDelayTooLarge(delay, MAX_EXECUTION_DELAY);
     _executionDelay = delay;
     emit ExecutionDelayUpdated(delay);
 }
```

Client's comments

We do not see a reason why we might need such a **MAX_EXECUTION_DELAY**.
All batches should be processed on time, without exceptions. If any batch gets stuck in the consolidation bus, alerts will be triggered.

INFORMATIONAL-31	Consolidation preconditions do not enforce staking-module deposit status	Acknowledged

Description

Lines:

- ConsolidationGateway.sol#L271-L283
- StakingRouter.sol#L650-L653

ConsolidationGateway._checkConsolidationPreconditions() currently gates new consolidation requests only on DSM-wide pause and **Lido.canDeposit()**. This check does not cover module-level deposit disablement in **StakingRouter**, where **StakingRouter.canDeposit(uint256)** returns **false** for non-active modules. As a result, consolidations may still be accepted while the target validator's staking module is paused/inactive at the router layer.
Impact: Consolidation requests may be admitted for targets that belong to modules that are intentionally not depositable, which may bypass module-level operational controls and create inconsistent behavior between consolidation intake and staking-router deposit policy.

Recommendation

We recommend extending consolidation precondition checks to include **StakingRouter.canDeposit(stakingModuleId)** for each target validator's module before accepting requests.

Client's comments

The module **canDeposit** status is not a requirement for starting or pausing the consolidation process.
A module can be paused or unpaused for many reasons unrelated to consolidation. At the same time, consolidation may still be used within the module for operator rebalancing.

INFORMATIONAL-32	Rate limit reconfiguration nullifies accrued replenishment	Acknowledged

Description

Lines: RateLimit.sol#L106-L117
RateLimit.setLimits() updates **prevTimestamp** to the current **timestamp** unconditionally and only adjusts **prevLimit** in the branch where **maxLimit < _data.prevLimit** or the previous limit was unlimited. This means previously accrued refill between **_data.prevTimestamp** and **timestamp** is not materialized before state is overwritten, and a reconfiguration can effectively restart replenishment from the current instant instead of preserving already earned quota.
Impact: Administrative limit updates may unintentionally reduce currently available capacity by dropping accrued refill, which can cause temporary under-allocation and inconsistent behavior versus expected time-based restoration semantics.

Recommendation

We recommend materializing the accrued limit before applying new parameters.

Client's comments

This is known behavior, and it’s accepted. We do not change the limit frequently.

INFORMATIONAL-33	NatSpec documentation contains minor typos in ConsolidationGateway	Fixed at: 92adbf3

Description

Lines:

- ConsolidationGateway.sol#L50
- ConsolidationGateway.sol#L64
- ConsolidationGateway.sol#L144

The NatSpec comments in **ConsolidationGateway** contain three minor typos that may affect documentation clarity:

1. The **InsufficientFee** error description reads "Thrown when a consolidation fee insufficient" - missing the word "is".
2. The **ConsolidationRequestsLimitExceeded** error references "at current day", but the rate limiter operates in terms of frames, not days. The parameter description should read "at current frame".
3. The **resume()** function NatSpec reads "Reverts if contracts is not paused" — should be singular "contract" to match the function's behavior.

Recommendation

We recommend correcting the three typos:

```
-     * @notice Thrown when a consolidation fee insufficient
+     * @notice Thrown when a consolidation fee is insufficient
```

```
-     * @param remainingLimit Amount of requests that still can be processed at current day
+     * @param remainingLimit Amount of requests that still can be processed at current frame
```

```
-     * @dev Reverts if contracts is not paused
+     * @dev Reverts if contract is not paused
```

INFORMATIONAL-34	Quadratic duplicate validator index detection in TopUpGateway.topUp()	Fixed at: 85be813

Description

Lines: TopUpGateway.sol#L135-L141

TopUpGateway.topUp() rejects duplicate validator indices through a nested **for**-loop that compares every pair of entries in **_topUps.validatorIndices**. The cost is O(n²) in calldata reads and comparisons, where **n** is bounded by **maxValidatorsPerTopUp**. The accompanying inline comment notes that the cost is acceptable because of this bound, but **TopUpGateway._setMaxValidatorsPerTopUp()** permits values up to **type(uint64).max**, so the protection relies entirely on the configured value.

Impact: Higher per-call gas cost than necessary on routine top-up batches. The cost grows quadratically with batch size, and combined with a permissive **maxValidatorsPerTopUp** cap it can be driven into a regime where **TopUpGateway.topUp()** exceeds the block gas limit.

Recommendation

We recommend requiring strictly ascending validator indices and replacing the quadratic scan with a single linear pass:

```
- // Check for duplicate validatorIndices (O(n^2) acceptable since bounded by maxValidatorsPerTopUp)
- for (uint256 i; i < validatorsCount; ++i) {
-     for (uint256 j = i + 1; j < validatorsCount; ++j) {
-         if (_topUps.validatorIndices[i] == _topUps.validatorIndices[j]) {
-             revert DuplicateValidatorIndex();
-         }
-     }
- }
+ // validatorIndices must be supplied in strictly ascending order
+ for (uint256 i = 1; i < validatorsCount; ++i) {
+     if (_topUps.validatorIndices[i] <= _topUps.validatorIndices[i - 1]) {
+         revert UnsortedOrDuplicateValidatorIndex();
+     }
+ }
```

With new error such as **UnsortedOrDuplicateValidatorIndex()**.

INFORMATIONAL-35	Missing zero-address check on admin parameter in TopUpGateway.initialize()	Fixed at: 85be813

Description

Line: TopUpGateway.sol#L100

TopUpGateway.initialize() grants **DEFAULT_ADMIN_ROLE** to the supplied **_admin** without verifying that the address is non-zero. If the deployment script passes **address(0)**- the role is granted to the zero address.

Because **TopUpGateway.initialize()** is guarded by the **initializer** modifier, the function cannot be re-invoked once it has executed. With **DEFAULT_ADMIN_ROLE** held only by **address(0)**, no further role grants, parameter changes, or operational adjustments are possible.

Impact: A deployment with a zero admin address renders the proxy permanently unmanageable. Recovery would require redeploying behind a fresh proxy, reinitializing all parameters, and re-granting all operational roles.

Recommendation

We recommend adding an explicit zero-address check at the start of **TopUpGateway.initialize()**:

```
  ) external initializer {
+     if (_admin == address(0)) revert AdminCannotBeZero();
      __AccessControlEnumerable_init();
      _grantRole(DEFAULT_ADMIN_ROLE, _admin);
```

With new error **AdminCannotBeZero()**.

INFORMATIONAL-36	Activation epoch check rejects validators active in the proven epoch	Fixed at: 2ace438

Description

Line: TopUpGateway.sol#L353

TopUpGateway._verifyValidatorWasActivated() reverts whenever the validator's **activationEpoch** is greater than or equal to the epoch derived from the proven slot. According to the consensus layer specification, a validator is active during epoch **e** if **activation_epoch <= e < exit_epoch**, so a validator activated exactly at epoch **e** is already active and eligible for participation.

The current non-strict comparison rejects this boundary case. Operators must therefore wait at least one additional epoch (~6.4 minutes) after activation before topping up the validator, even though the consensus layer already considers it active. There is no security benefit to the additional buffer — the on-chain check rejects a validator state that the protocol itself recognises as live.

The check still correctly excludes pending-activation validators, since their **activation_epoch** is **FAR_FUTURE_EPOCH**, which is strictly greater than any realistic proven epoch. Switching to strict **>** does not weaken that guarantee.

Impact: Top-up batches that include validators activated exactly at the proven epoch revert with **ValidatorIsNotActivated**. Operators incur an avoidable one-epoch latency for every newly activated validator, with no corresponding security gain.

Recommendation

We recommend aligning the comparison with the consensus layer specification by treating **activation_epoch == epoch** as active:

```
- if (_w.activationEpoch >= epoch) revert ValidatorIsNotActivated();
+ if (_w.activationEpoch > epoch) revert ValidatorIsNotActivated();
```

INFORMATIONAL-37	Stale lido.ValidatorsExitBus.maxExitRequestLimit storage not cleared on upgrade to v3	Fixed at: 1a25ee4

Description

Lines: ValidatorsExitBusOracle.sol#L90-L96

ValidatorsExitBusOracle.finalizeUpgrade_v3() handles the upgrade from v2 to v3. In v2, exit rate limiting was tracked per-validator-count and stored at the unstructured storage slot **keccak256("lido.ValidatorsExitBus.maxExitRequestLimit")**. In v3, the rate limiting mechanism changed to a balance-based model backed by a new slot **keccak256("lido.ValidatorsExitBus.exitBalanceLimitEth")**. The upgrade initialises the new slot via **_setExitRequestLimit()**, but the old **EXIT_REQUEST_LIMIT_POSITION** slot is never zeroed out and retains its v2 data after the upgrade completes.

Recommendation

We recommend zeroing the old storage slot inside **ValidatorsExitBusOracle.finalizeUpgrade_v3()**.

INFORMATIONAL-38	Undelivered manual exit-hash entries become permanently stuck after upgrade	Acknowledged

Description

Lines:

- ValidatorsExitBus.sol#L350
- ValidatorsExitBus.sol#L709

ValidatorsExitBus.submitExitRequestsHash()stores the current **contractVersion** alongside the hash. **ValidatorsExitBus.submitExitRequestsData()** requires the stored version to match the live contract version at delivery time. After **finalizeUpgrade_v3()** increments the contract version, any undelivered pre-upgrade hash entries fail this check and revert. Re-submitting the same hash via **submitExitRequestsHash()** is also blocked: **_storeNewHashRequestStatus()** reverts with **ExitHashAlreadySubmitted** whenever the stored **contractVersion** is nonzero, regardless of delivery status.
Impact: undelivered manual exit-hash entries from before the upgrade cannot be delivered or resubmitted, creating a liveness issue for the manual submission flow.

Recommendation

We recommend allowing resubmission of a hash entry that was previously stored under an older contract version but never delivered or documenting this case as a potential operational risk.

Client's comments

> This is expected behavior. During the upgrade, we intentionally want to invalidate all old requests, because the upgrade may fix issues in the request logic or introduce new logic that is incompatible with requests created under the previous version.

INFORMATIONAL-39	maxValidatorsPerReport limit not enforced in oracle report path	Acknowledged

Description

Lines:

- ValidatorsExitBusOracle.sol#L231-L253
- ValidatorsExitBus.sol#L353-L357

The **MAX_VALIDATORS_PER_REPORT_POSITION** storage slot is documented as "the maximum number of validator exit requests allowed per processing report" and is configurable by the **EXIT_REQUEST_LIMIT_MANAGER_ROLE**. The limit is enforced in the manual submission path inside **ValidatorsExitBus.submitExitRequestsData()**. However, the oracle report path in **ValidatorsExitBusOracle._handleConsensusReportData()** performs no such check. A consensus report containing more exit requests than **maxValidatorsPerReport** is accepted and processed without any revert.

Recommendation

We recommend resolving the inconsistency between the two exit request submission paths: either enforce **maxValidatorsPerReport** in **ValidatorsExitBusOracle._handleConsensusReportData()** as it is in **ValidatorsExitBus.submitExitRequestsData()**, or explicitly document that the limit is intentionally not applied to the oracle path

Client's comments

> MAX_VALIDATORS_PER_REPORT was designed only for manual report submissions. For the VEBO flow, there is a separate sanity check for the total exit balance per report.

INFORMATIONAL-40	Pending balance sanity check can be griefed by unaccounted direct top-up deposits	Fixed at: 5bd760f

Description

Lines:

- OracleReportSanityChecker.sol#L925-L928
- Lido.sol#L747-L751

OracleReportSanityChecker._checkCLPendingBalanceAndCalculateActivatedBalanceWithGap() enforces **postCLPendingBalance <= preCLPendingBalance + deposits** and reverts otherwise. The **deposits** value is sourced from **Lido.getBalanceStats()** and reflects protocol-accounted deposits for the reporting window. A third party can perform a direct top-up deposit to a Lido validator, increasing CL pending balance while not being reflected in this accounting path for the same report, causing **IncorrectTotalPendingBalance** and preventing report finalization.

Impact: A third party may grief accounting oracle report processing for a frame by donating ETH via direct top-up deposits that move CL pending above the sanity-check upper bound. In low-deposit periods, the lower-bound cost is one minimum top-up-sized deposit, while sustained multi-frame disruption may require additional funding to keep unaccounted pending growth ahead of naturally accounted deposits.

Recommendation

We recommend documenting this as an operational risk: if a reporting frame is blocked by IncorrectTotalPendingBalance, the incident response should include checking whether the excess pending balance originates from a third-party deposit to a Lido validator pubkey, in which case resubmitting the report in the next frame is sufficient.

INFORMATIONAL-41	TopUpGateway updates top-up rate-limit state even when no deposit is performed	Fixed at: b530934

Description

Lines:

- TopUpGateway.sol#L188
- TopUpGateway.sol#L143-L149
- TopUpGateway.sol#L286
- StakingRouter.sol#L723

TopUpGateway.topUp() computes per-validator **topUpLimits** and forwards them to **StakingRouter.topUp()** even when every computed limit is zero. **StakingRouter.topUp()** may legitimately complete without pulling ETH or making any deposit because it only withdraws from Lido when the final **amount > 0**, while still invoking **allocateDeposits()** for module-side processing. After that call returns, **TopUpGateway._setLastTopUpData()** is executed unconditionally, updating both **lastTopUpTimestamp** and **lastTopUpBlock**. Because **TopUpGateway._requireBlockDistancePassed()** and **TopUpGateway._verifyRootAge()** both depend on these values, a no-op top-up still consumes the global rate-limit window and advances the freshness threshold for future proofs.

Impact: A caller with **TOP_UP_ROLE** may submit a no-op top-up that performs no deposit yet still blocks subsequent valid top-ups until **minBlockDistance** passes and may invalidate already prepared witnesses whose **childBlockTimestamp** is no longer strictly newer than **lastTopUpTimestamp**. This may create unnecessary operational friction, allow griefing of the top-up workflow under privileged misuse, and trigger module-side cursor advancement without any corresponding validator funding.

Recommendation

We recommend updating **lastTopUpTimestamp** and **lastTopUpBlock** only when a positive amount is actually deposited.

INFORMATIONAL-42	ValidatorsExitBus misses zero module validation during exit balance calculation	Fixed at: 6188879

Description

ValidatorsExitBus._calculateTotalExitBalanceEth() decodes **moduleId** from each exit request and calls **ValidatorsExitBus._getModuleMaxEBWeight()** without rejecting **moduleId == 0**.

Recommendation

We recommend applying zero moduleId validation in **ValidatorsExitBus._calculateTotalExitBalanceEth()**, consistent with **ValidatorsExitBus._processExitRequestsListV1()** and **ValidatorsExitBus._processExitRequestsListV2()**.

```
+   if (moduleId == 0) {
+       revert InvalidModuleId();
+   }

    if (moduleId != cachedModuleId) {
```

INFORMATIONAL-43	Lido.collectRewardsAndProcessWithdrawals() ignores actual amount withdrawn from EL rewards vault	Acknowledged

Description

Line: Lido.sol#L1075

LidoExecutionLayerRewardsVault.withdrawRewards() returns the actual withdrawn amount, capped by the vault's balance. In **Lido.collectRewardsAndProcessWithdrawals()**, this return value is discarded and **_elRewardsToWithdraw** parameter is used directly to compute **postBufferedEther**. If the vault's balance were lower than **_elRewardsToWithdraw** at execution time, buffered ether would be overstated.

While the current oracle flow derives **_elRewardsToWithdraw** from the vault's reported balance, making this scenario unreachable under normal conditions, the defensive check is missing.

Recommendation

We recommend using the return value of **withdrawRewards()** for the buffered ether calculation:

```
+ uint256 elRewardsReceived;
  if (_elRewardsToWithdraw > 0) {
-     _elRewardsVault(locator).withdrawRewards(_elRewardsToWithdraw);
+     elRewardsReceived = _elRewardsVault(locator).withdrawRewards(_elRewardsToWithdraw);
  }

  uint256 postBufferedEther = _getBufferedEther()
-     .add(_elRewardsToWithdraw)
+     .add(elRewardsReceived)
      .add(_withdrawalsToWithdraw)
      .sub(_etherToLockOnWithdrawalQueue);
```

Client's comments

In the **_applyOracleReportContext** method, **_sanityChecks** is called first. It calls **oracleReportSanityChecker.checkAccountingOracleReport**, which checks **elRewardsVaultBalance** against the current value. Only after that does the code call **Lido.collectRewardsAndProcessWithdrawals()**. Therefore, the described situation does not occur.

INFORMATIONAL-44	ValidatorsExitBus._getTimestamp() returns uint32, will overflow around year 2106	Acknowledged

Description

Line: ValidatorsExitBus.sol#L628

The **ValidatorsExitBus._getTimestamp()** function in returns **uint32(block.timestamp)**. This truncates **block.timestamp** to 32 bits, which overflows around February 2106.

Recommendation

We recommend using **uint256** for timestamps to avoid the year 2106 overflow.

Client's comments

This limitation is known. We store the **uint32** timestamp in a single storage slot together with other fields.

INFORMATIONAL-45	StakingRouter.topUp() can mutate module state while Lido cannot deposit	Fixed at: 076092d

Description

Lines:

- StakingRouter.sol#L690-L700
- StakingRouter.sol#L723-L727
- Lido.sol#L856-L858

StakingRouter.topUp() reads **LIDO.getDepositableEther()**, computes the target module allocation, and then unconditionally calls **IStakingModuleV2.allocateDeposits()** with **smDepositableEthAmountRounded**.
The only deposit availability check on this execution path is **require(canDeposit(), "CAN_NOT_DEPOSIT")** in **Lido.withdrawDepositableEther()**. However, **StakingRouter.topUp()** reaches this check only if **amount** is greater than zero. If **Lido.canDeposit()** is false and **smDepositableEthAmountRounded == 0**, the router still allows the stateful module allocation call to execute.

Recommendation

We recommend enforcing Lido deposit availability before any stateful module allocation in **StakingRouter.topUp()**.

```
 interface ILido {
+    function canDeposit() external view returns (bool);
     function getDepositableEther() external view returns (uint256);
     function withdrawDepositableEther(uint256 _amount, uint256 _depositsCount) external;
 }
```

```
     uint256 depositableEther = LIDO.getDepositableEther();
     uint256 smDepositableEthAmount = _getModuleDepositAllocation(_stakingModuleId, depositableEther, true);
+    if (!LIDO.canDeposit()) revert LidoDepositsPaused();

     // Call allocateDeposits on the staking module to determine for what amount deposit each key
```

STATE
MIND