Lido Community Staking Module v2

Audited by: rubencrxz

Table of Contents

1. Protocol Overview

Lido CSM (Community Staking Module) v2 is a permissionless-by-design Ethereum validator staking module integrated into the Lido protocol via the `StakingRouter`. It enables community node operators to register

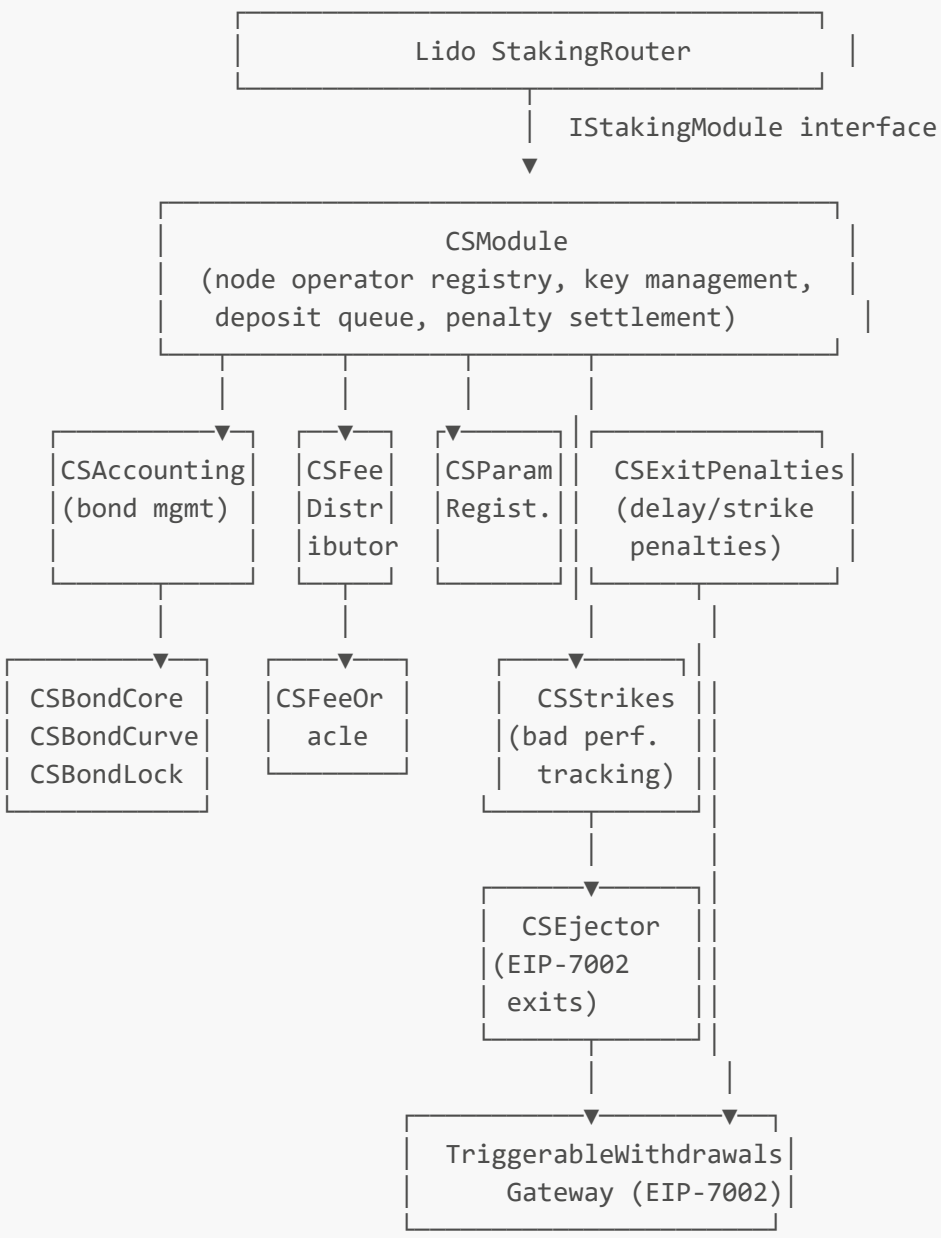
and run validators with **stETH-denominated bond collateral** instead of requiring large ETH upfront. Rewards are partially socialized across the Lido protocol and partially distributed per-operator through an oracle-driven Merkle fee tree.

Core design principles:

- Bond collateral in stETH shares (earns rebase yield while held).
- Piecewise linear bond curves: lower marginal bond cost for operators with many keys.
- Two onboarding gates: `VettedGate` (Merkle whitelist, lower bond) and `PermissionlessGate` (open, default bond).
- Bond-curve-based parameter differentiation: most protocol parameters (penalties, queue config, strikes thresholds) are tied to `curveId` rather than operator ID.
- EIP-7002 triggerable withdrawals for forced exits.
- Oracle-delivered Merkle trees for both fee distribution and bad-performance strikes.

---

2. Architecture Map



```
Onboarding:
  VettedGate ──────────┐
  PermissionlessGate ──┴──► CSModule.createNodeOperator()
  VettedGateFactory ──► deploys VettedGate via OssifiableProxy

Proof verification:
  CSVerifier ──► CSModule.submitWithdrawals()   (beacon withdrawal proofs)
```

---

3. Core Data Structures

3.1 NodeOperator (ICSModule.sol)

Packed into 8 storage slots:

```
struct NodeOperator {
    // Slot 1
    uint32 totalAddedKeys;        // increased on add, decreased on removal of
undeposited keys
    uint32 totalWithdrawnKeys;    // only increases (fully withdrawn from beacon)
    uint32 totalDepositedKeys;    // only increases (submitted to beacon)
    uint32 totalVettedKeys;       // increases/decreases with vetting status

    // Slot 2
    uint32 stuckValidatorsCount;
    uint32 depositableValidatorsCount;
    uint32 targetLimit;
    uint8  targetLimitMode;       // 0 = none, 1 = relative, 2 = forced

    // Slot 3
    uint32  totalExitedKeys;      // only increases (except unsafe updates)
    uint32  enqueuedCount;        // keys currently placed in deposit queues
    address managerAddress;

    // Slots 4-5
    address proposedManagerAddress;
    address rewardAddress;
    address proposedRewardAddress;
    bool    extendedManagerPermissions;
    bool    usedPriorityQueue;
}
```

3.2 Batch (QueueLib.sol)

A single `uint256` encoding a deposit queue item:

```
Batch = [nodeOperatorId (64 bits) | keysCount (64 bits) | nextIndex (128 bits)]
```

3.3 BondCurveInterval (CSBondCurve.sol)

```
struct BondCurveInterval {
    uint32  minKeysCount;  // starting key count for this segment
    uint256 minBond;       // cumulative bond required at minKeysCount (in stETH
wei)
    uint256 trend;         // additional bond per key in this segment
}
```

Input uses `BondCurveIntervalInput { uint256 minKeysCount; uint256 trend; }` and the contract computes cumulative `minBond` values on storage.

3.4 BondLock (CSBondLock.sol)

```
struct BondLock {
    uint128 amount;   // ETH-denominated lock amount (max ~340 ETH due to uint128)
    uint128 until;    // Unix timestamp of lock expiry
}
```

3.5 KeyStrikes (ICSStrikes.sol)

```
struct KeyStrikes {
    uint256   nodeOperatorId;
    uint256   keyIndex;
    uint256[] data;    // per-oracle-frame strike counts
}
```

3.6 ExitPenaltyInfo (ICSExitPenalties.sol)

```
struct ExitPenaltyInfo {
    MarkedUint248 delayPenalty;
    MarkedUint248 strikesPenalty;
    MarkedUint248 withdrawalRequestFee;
}

struct MarkedUint248 {
    uint248 value;
    bool    isValue;  // distinguishes "not set" from "set to zero"
}
```

3.7 KeyNumberValueInterval (ICSParametersRegistry.sol)

```
struct KeyNumberValueInterval {
    uint256 minKeyNumber;   // first key in this range
    uint256 value;          // value in basis points for this range
}
// Example: [[1, 10000], [11, 8000]] = 100% for keys 1-10, 80% for keys 11+
```

3.8 QueueConfig

```
struct QueueConfig {
    uint32 priority;     // lower number = higher priority in deposit queue
    uint32 maxDeposits;  // max number of deposits via this priority queue per
operator
}
```

3.9 ValidatorWithdrawalInfo

```
struct ValidatorWithdrawalInfo {
    uint256 nodeOperatorId;
    uint256 keyIndex;
    uint256 amount;  // withdrawn ETH in wei
}
```

---

4. Onboarding & Key Registration

4.1 Gates

Two contracts control operator creation:

PermissionlessGate — Open to anyone. Assigns the default bond curve. Minimal logic: validates bond deposit, calls `CSModule.createNodeOperator()`, then `addValidatorKeys*()`.

VettedGate — Requires a valid Merkle proof against a DAO-managed `treeRoot`. Assigns a privileged `curveId` with lower bond requirements. Once a proof is consumed for an address (`_consumedAddresses[member] = true`), that address cannot create another operator via this gate with the same proof.

Additional VettedGate features:

- **Referral program**: tracked per-season (`_referralCounts[keccak256(referrer, season)]`). A referrer who reaches `referralsThreshold` in the active season can claim a better `referralCurveId` via `claimReferrerBondCurve()`.
- **Season management**: DAO calls `startNewReferralProgramSeason(curveId, threshold)` and `endCurrentReferralProgramSeason()`.

- **Curve claim for existing operators**: `claimBondCurve(nodeOperatorId, proof)` lets a whitelisted address claim the vetted curve for an already-created operator.

VettedGateFactory — Deploys new `VettedGate` instances via `OssifiableProxy`. Each deployment is a clone with its own `treeRoot`, `curveId`, and referral state.

4.2 Operator Creation

```
Gate → CSModule.createNodeOperator(from, managementProperties, referrer)
     → assigns nodeOperatorId (sequential)
     → stores managerAddress, rewardAddress, extendedManagerPermissions
     → emits NodeOperatorAdded, ReferrerSet
```

4.3 Key Addition

Three entry points differing only in bond token:

```
addValidatorKeysETH   (msg.value)
addValidatorKeysStETH (ERC20 transfer / EIP-2612 permit)
addValidatorKeysWstETH (unwraps wstETH → stETH internally)
```

Validation sequence:

1. `CSAccounting.getRequiredBondForNextKeys(nodeOperatorId, additionalKeys)` — returns shortfall in stETH wei.
2. Bond deposited to `CSAccounting` → stETH shares stored in `CSBondCore`.
3. `SigningKeys.saveKeysSigs()` — packed assembly writes pubkeys and signatures into dedicated storage slots.
4. `totalAddedKeys += keysCount`.
5. Vetting: if operator is within `keysLimit` (from registry) and bond is sufficient, `totalVettedKeys` advances.
6. `_updateDepositableValidatorsCount()` recalculates depositable count.
7. `_enqueueNodeOperatorKeys()` places new keys into the appropriate priority queue.

Key removal (`removeKeys(nodeOperatorId, startIndex, keysCount)`):

- Only undeposited keys (`index >= totalDepositedKeys`) can be removed.
- Uses swap-delete: moves last key to the removed slot.
- Charges `keyRemovalCharge` (from registry) per removed key via `CSAccounting.chargeFee()`.

4.4 Address Management (NOAddresses.sol)

Two-step pattern for both manager and reward address changes:

1. Current holder calls `propose*AddressChange(nodeOperatorId, proposed)`.
2. Proposed address calls `confirm*AddressChange(nodeOperatorId)`.

Special rules:

- `resetNodeOperatorManagerAddress()` — reward address can reset manager to itself (when `extendedManagerPermissions = false`).
- `changeNodeOperatorRewardAddress()` — direct change by manager (only when `extendedManagerPermissions = true`).

---

5. Bond System

5.1 CSBondCore — Storage and Operations

Storage slot via unstructured storage pattern:

```
struct CSBondCoreStorage {
    mapping(uint256 nodeOperatorId => uint256 shares) bondShares;
    uint256 totalBondShares;
}
```

Bond is denominated in **stETH shares** (not stETH amount), so it passively earns rebase yield. Conversions to ETH use the live Lido share ratio at query time.

Key operations:

Function	Action
`_depositETH(from, noId)`	Stakes ETH with Lido, stores resulting shares
`_depositStETH(from, noId, amount)`	Transfers stETH shares directly
`_depositWstETH(from, noId, amount)`	Unwraps wstETH → stETH, stores shares
`_burn(noId, amount)`	Requests burn from IBurner; pending until next rebase
`_charge(noId, amount, recipient)`	Transfers shares to `chargePenaltyRecipient`
`_claimStETH/WstETH/UnstETH(noId, amount, to)`	Withdraws claimable excess bond

`_getClaimableBondShares()` is virtual — overridden by `CSAccounting` to exclude required bond coverage.

5.2 CSBondCurve — Piecewise Linear Curves

Storage:

```
struct CSBondCurveStorage {
    mapping(uint256 nodeOperatorId => uint256 bondCurveId) operatorBondCurveId;
    BondCurve[] bondCurves;   // array of curves; index 0 = default
}
```

Forward lookup `_getBondAmountByKeysCount(keys, curve)`:

1. Binary search for the interval where `minKeysCount <= keys`.
2. `result = minBond[i] + (keys - minKeysCount[i]) * trend[i]`

Reverse lookup `_getKeysCountByBondAmount(amount, curve)`:

1. Binary search for interval where `minBond[i] <= amount`.
2. `keys = minKeysCount[i] + (amount - minBond[i]) / trend[i]`
3. Division truncates — keys are floor-rounded.

`_setBondCurve(noId, curveId)` triggers `MODULE.updateDepositableValidatorsCount(noId)` since curve change alters required bond.

5.3 CSBondLock — Temporary Bond Lock

Used for MEV/EL rewards stealing penalty investigation period.

Lock creation / extension (`_lock(noId, amount)`):

- If an active lock exists: `newAmount = existing.amount + amount`, timer resets to `block.timestamp + bondLockPeriod`.
- Creates new lock otherwise.
- Amount stored as `uint128` — max ~340 ETH per operator.

Expiry check (`getActualLockedBond(noId)`):

- Returns `lock.amount` only if `lock.until > block.timestamp`, else returns 0.

Settlement (`settleLockedBondETH(noId)`) in CSAccounting:

- If lock expired: `_burn(noId, lock.amount)`, `_remove(noId)`, returns `(true, amount)`.
- If still active: returns `(false, 0)`.

5.4 Required Bond Calculation

The critical formula used in `getRequiredBondForNextKeys()` and `getBondSummary()`:

```
requiredBond = getBondAmountByKeysCount(totalDepositedKeys - totalWithdrawnKeys +
additionalKeys, curve)
             + getActualLockedBond(noId)
```

This means locked bond **adds to** the required bond, creating an additional collateral demand on top of the key-coverage requirement.

`_getUnbondedKeysCount(noId, includeLockedBond)`:

```
coveredKeys = getKeysCountByBondAmount(currentBond - lockedBond, curve)
unbonded = max(0, nonWithdrawnKeys - coveredKeys)
```

A 10 wei rounding margin is applied in the reverse curve lookup to avoid dust-related misclassification.

---

6. Deposit Queue & Priority System

6.1 Queue Structure

Each priority level has its own `Queue` struct:

```
struct Queue {
    uint128 head;
    uint128 tail;
    mapping(uint128 => Batch) queue;
}
```

Priority levels are set per `curveId` in `CSParametersRegistry.getQueueConfig()`. Lower numeric priority = higher precedence when `StakingRouter` requests deposits. There are also:

- `QUEUE_LOWEST_PRIORITY` — default fallback queue.
- `QUEUE_LEGACY_PRIORITY` — reserved for v1 migrated operators.

6.2 Key Enqueueing

`_enqueueNodeOperatorKeys(noId, keysCount)`:

1. Fetches `(priority, maxDeposits)` from registry for operator's `curveId`.
2. If `usedPriorityQueue = false` and `maxDeposits > 0`: enqueues up to `maxDeposits` in the priority queue, rest in lowest priority queue.
3. Sets `usedPriorityQueue = true` after first priority usage.
4. `enqueuedCount += keysCount` on the `NodeOperator` struct.

`migrateToPriorityQueue(noId)`:

- Allows an operator who has received a privileged curveId post-creation to move existing enqueued keys to the priority queue (one-time operation).

6.3 Deposit Extraction

`obtainDepositData(depositsCount)` — called by StakingRouter:

1. Iterates priority queues from highest to lowest priority.
2. For each `Batch` at queue head: reads `(noId, keys, next)`.
3. Takes min(available, needed) keys from the batch.
4. Updates `totalDepositedKeys`, decrements `enqueuedCount`.
5. Loads pubkeys + signatures from `SigningKeys` storage.
6. Returns packed keys/signatures to StakingRouter.

Invariant maintained: `NO.enqueuedCount >= sum(keys in all NO batches across all queues)`.

6.4 Queue Cleaning

`cleanDepositQueue(maxItems)`:

- Iterates up to `maxItems` from queue head.
- Uses `TransientUintUintMapLib` as a transient key-value map (`queueLookup`) shared across all priority levels during one call.
- A batch is removed if `queueLookup.get(noId) >= NO.depositableValidatorsCount`.
- On removal: `enqueuedCount -= batchKeys`.
- The transient map tracks already-seen operators to avoid re-processing.

6.5 Depositable Count Update

`_updateDepositableValidatorsCount(noId)`:

```
nonDeposited = totalVettedKeys - totalDepositedKeys
unbondedKeys = _getUnbondedKeysCount(noId, true)

if (unbondedKeys >= totalAddedKeys - totalDepositedKeys):
    newCount = 0
elif (unbondedKeys > totalAddedKeys - totalVettedKeys):
    newCount = nonDeposited - unbondedKeys
else:
    newCount = nonDeposited

if (targetLimitMode > 0):
    activeKeys = totalDepositedKeys - totalWithdrawnKeys
    limit = targetLimit - activeKeys
    newCount = min(newCount, max(0, limit))
```

Called on: key addition, key removal, bond deposit, bond curve change, penalty, vetting update, target limit update.

---

7. Fee Distribution & Oracle

7.1 CSFeeDistributor

State:

```
bytes32 treeRoot;
string  treeCid;   // IPFS CID
string  logCid;
uint256 totalClaimableShares;
mapping(uint256 noId => uint256 shares) distributedShares;  // monotonically
increasing
```

`processOracleReport(treeRoot, treeCid, logCid, distributed, rebate, refSlot)`:

- Called by `CSFeeOracle` after consensus.
- Validates: root/CID not previously used.
- `totalClaimableShares += distributed`.
- Sends `rebate` stETH shares to `rebateRecipient`.
- Stores snapshot in `_distributionDataHistory`.

`distributeFees(noId, cumulativeFeeShares, proof)`:

- Verifies Merkle proof: `hashLeaf(noId, cumulativeFeeShares)` against `treeRoot`.
- `delta = cumulativeFeeShares - distributedShares[noId]`.
- `distributedShares[noId] = cumulativeFeeShares` (monotonic, cannot decrease).
- Transfers `delta` stETH shares from distributor to `CSAccounting` → increases operator's bond.
- Subtracts `delta` from `totalClaimableShares`.

Leaf hash (double-hashed for second pre-image resistance):

```
hashLeaf(noId, shares) = keccak256(keccak256(abi.encode(noId, shares)))
```

7.2 CSFeeOracle

Extends Lido's `BaseOracle` (consensus slot tracking, versioning).

`submitReportData(ReportData data, contractVersion)`:

- Validates consensus quorum via BaseOracle.
- Calls `CSFeeDistributor.processOracleReport()`.
- Calls `CSStrikes.processOracleReport()`.

`ReportData` contains:

- `treeRoot`, `treeCid`, `logCid` — fee Merkle tree.
- `distributed` — total stETH shares to mark as distributable.
- `rebate` — shares returned to protocol.
- `refSlot` — reference beacon slot.
- `strikesTreeRoot`, `strikesTreeCid` — bad performance Merkle tree.

7.3 Reward Claiming

`CSAccounting.claimRewardsStETH(noId, stETHAmount, cumulativeFeeShares, proof)`:

1. `_pullFeeRewards(noId, cumulativeFeeShares, proof)` → calls `CSFeeDistributor.distributeFees()`.
2. `claimableShares = _getClaimableBondShares(noId)` = `currentShares - requiredShares`.
3. Transfers `min(requested, claimable)` shares as stETH/wstETH/unstETH depending on method.

`pullFeeRewards()` — standalone version that only distributes without claiming (useful for topping up bond before adding keys).

---

8. Exit System

8.1 Exit Types

Defined in `ExitTypes.sol`:

- `VOLUNTARY_EXIT_TYPE_ID = 0` — operator-initiated.
- `STRIKES_EXIT_TYPE_ID = 1` — bad performance ejection.

8.2 CSEjector

Interfaces with EIP-7002 `TriggerableWithdrawalsGateway` (fetched from `LidoLocator`).

Voluntary exit (`voluntaryEject(noId, startFrom, keysCount, refundRecipient)`):

1. Access: manager or reward address of the operator.
2. Validates: keys are deposited, not already withdrawn.
3. Calls `CSExitPenalties.processTriggeredExit(noId, pubkey, paidFee, VOLUNTARY_EXIT_TYPE_ID)`.
4. Calls `gateway.triggerFullWithdrawals(pubkeys)` with ETH fee.
5. Excess ETH refunded to `refundRecipient`.

Array variant (`voluntaryEjectByArray(noId, keyIndices[], refundRecipient)`):

- Ejects non-sequential keys by index array.
- `msg.value` must be exactly `keyIndices.length * feePerWithdrawal`.

Bad performer ejection (`ejectBadPerformer(noId, keyIndex, refundRecipient)`):

- `onlyStrikes` modifier.
- Single-key variant.
- Records `STRIKES_EXIT_TYPE_ID` in `CSExitPenalties`.

8.3 CSVerifier — Beacon Proof Verification

Immutable beacon chain parameters:

```
BEACON_ROOTS            // EIP-4788 precompile
SLOTS_PER_EPOCH = 32
SLOTS_PER_HISTORICAL_ROOT = 8192
GI_FIRST_WITHDRAWAL_*   // generalized indices for withdrawal SSZ paths
GI_FIRST_VALIDATOR_*    // generalized indices for validator SSZ paths
GI_FIRST_HISTORICAL_SUMMARY_*
FIRST_SUPPORTED_SLOT    // min slot accepted (anti-replay)
PIVOT_SLOT              // fork boundary (Capella)
WITHDRAWAL_ADDRESS      // Lido withdrawal credentials
```

`processWithdrawalProof(ProvableBeaconBlockHeader, WithdrawalWitness, noId, keyIndex)`:

1. Fetches `stateRoot` from EIP-4788 precompile using `header.slot`.

2. Verifies `BeaconBlockHeader` against `stateRoot` via SSZ Merkle proof.
3. Verifies `Withdrawal` struct against block body root via SSZ Merkle proof at `GI_FIRST_WITHDRAWAL + index`.
4. Verifies `Validator` struct (pubkey, withdrawalCredentials, exitEpoch, withdrawableEpoch, slashed) against state root.
5. Validates: `withdrawal.withdrawalAddress == WITHDRAWAL_ADDRESS`.
6. Validates: proof is for full withdrawal (epoch >= withdrawableEpoch OR validator is slashed).
7. Calls `MODULE.submitWithdrawals([{noId, keyIndex, amount: gweiToWei(withdrawal.amount)}])`.

`processHistoricalWithdrawalProof(..., HistoricalHeaderWitness, ...)`:

- Same as above but first proves the historical block header via `historical_summaries` Merkle path.
- Used for withdrawals older than EIP-4788's 8192-block window.

8.4 Withdrawal Processing in CSModule

`submitWithdrawals(ValidatorWithdrawalInfo[])` — callable only by `VERIFIER_ROLE`:

For each withdrawal:

1. Verify operator exists, key is deposited, not already withdrawn.
2. `_isValidatorWithdrawn[_keyPointer(noId, keyIndex)] = true`.
3. `totalWithdrawnKeys++`.
4. Collect all pending exit penalties from `CSExitPenalties.getExitPenaltyInfo()`:
   - `delayPenalty` (if set).
   - `strikesPenalty` (if set).
   - `withdrawalRequestFee` (if set, i.e., triggered exit).
5. Calculate `shortfall = max(0, requiredBond - withdrawalAmount)`.
6. Total penalty = delayPenalty + strikesPenalty + withdrawalRequestFee + shortfall.
7. `CSAccounting.penalize(noId, totalPenalty)` — burns stETH shares.
8. Emit `WithdrawalSubmitted`.
9. `_updateDepositableValidatorsCount(noId)`.

Key pointer computation:

```
_keyPointer(noId, keyIndex) = (noId << 128) | keyIndex
```

---

9. Penalty System

9.1 EL Rewards Stealing Penalty

Report phase (`reportELRewardsStealingPenalty(noId, blockHash, amount)`):

- Role: `REPORT_EL_REWARDS_STEALING_PENALTY_ROLE` (CSM Committee Multisig).
- `lockAmount = amount + PARAMETERS_REGISTRY.getElRewardsStealingAdditionalFine(curveId)`.

- Calls `CSAccounting.lockBondETH(noId, lockAmount)`.
- Creates/extends `BondLock` with `bondLockPeriod` expiry.

Compensation (`compensateELRewardsStealingPenalty(noId)`):

- Anyone can pay ETH to compensate.
- ETH forwarded to `elRewardsVault`.
- `releaseLockedBondETH(noId, msg.value)` reduces lock by payment.

Cancellation (`cancelELRewardsStealingPenalty(noId, amount)`):

- Committee role only.
- Reduces lock without burning.

Settlement (`settleELRewardsStealingPenalty(noIds[])`):

- Role: `SETTLE_EL_REWARDS_STEALING_PENALTY_ROLE` (Easy Track).
- For each operator: `CSAccounting.settleLockedBondETH(noId)`.
- If lock expired: burns locked amount, removes lock, updates depositable count.
- If still active: skipped.

9.2 Exit Delay Penalty

Tracked in `CSExitPenalties`:

`processExitDelayReport(noId, pubkey, eligibleToExitInSec)` — called by CSModule:

1. Check `isValidatorExitDelayPenaltyApplicable()`: returns true if `eligibleToExitInSec > allowedExitDelay(curveId)`.
2. If applicable and not already set: `delayPenalty.value = PARAMETERS_REGISTRY.getExitDelayPenalty(curveId)`.
3. Mark `delayPenalty.isValue = true` (idempotent — only set once per validator).

`allowedExitDelay` is a per-curve parameter in seconds (default from registry).

9.3 Strikes Penalty (Bad Performance)

`processStrikesReport(noId, pubkey)` — called by CSEjector during `ejectBadPerformer()`:

1. If not already set: `strikesPenalty.value = PARAMETERS_REGISTRY.getBadPerformancePenalty(curveId)`.
2. Mark `strikesPenalty.isValue = true`.

9.4 Withdrawal Request Fee

`processTriggeredExit(noId, pubkey, withdrawalRequestPaidFee, exitType)`:

- If `exitType != VOLUNTARY_EXIT_TYPE_ID`:
  - `recordedFee = min(withdrawalRequestPaidFee, maxWithdrawalRequestFee(curveId))`.
  - Stores `withdrawalRequestFee = {value: recordedFee, isValue: true}`.
- Voluntary exits do not record any fee.

All three penalties are applied together at `submitWithdrawals()` time after proof of withdrawal.

9.5 Penalize vs Charge

Two distinct bond reduction methods in CSAccounting:

Method	Mechanism	Use case
`penalize(noId, amount)`	Burns stETH shares via IBurner	Slashing penalties, shortfalls
`chargeFee(noId, amount)`	Transfers to `chargePenaltyRecipient`	Key removal charges, withdrawal fees

10. Parameters Registry

`CSParametersRegistry` stores **default** and **per-curveId** values for all protocol parameters. The pattern for each parameter is:

1. `defaultX` — fallback if no curve-specific value set.
2. `_xData[curveId]` — optional override per bond curve.
3. `getX(curveId)` — returns custom if `isValue = true`, else default.
4. `setX(curveId, value)` / `unsetX(curveId)` — DAO-controlled setters.

Full parameter list:

Parameter	Type	Description
`keyRemovalCharge`	`uint256`	stETH penalty per removed key
`elRewardsStealingAdditionalFine`	`uint256`	Additional fine on top of stolen amount
`keysLimit`	`uint256`	Max keys per operator on this curve
`rewardShare`	`KeyNumberValueInterval[]`	% of rewards retained by operator (BP, tiered by key count)
`performanceLeeway`	`KeyNumberValueInterval[]`	Performance tolerance before strike (BP, tiered)
`strikesParams`	`StrikesParams`	`(lifetime: oracle frames, threshold: strike count to eject)`
`badPerformancePenalty`	`uint256`	Penalty applied when ejected for bad performance

Parameter	Type	Description
`performanceCoefficients`	`PerformanceCoefficients`	Weights for attestations/blocks/sync in performance scoring
`allowedExitDelay`	`uint256`	Seconds before exit delay penalty applies
`exitDelayPenalty`	`uint256`	stETH penalty for exceeding exit delay
`maxWithdrawalRequestFee`	`uint256`	Cap on withdrawal request fee charged to operator
`queueConfig`	`QueueConfig`	Queue priority and max deposits for this curve

`KeyNumberValueInterval[]` arrays must be sorted by `minKeyNumber` with values ≤ 10000 BP. Validated by `_validateKeyNumberValueIntervals()` on set.

---

11. Supporting Libraries

11.1 SigningKeys.sol

Keys and signatures stored in dedicated contract storage using a computed slot:

```
baseSlot = keccak256(abi.encodePacked(SIGNING_KEYS_POSITION, nodeOperatorId,
keyIndex))
```

Per-key layout (48-byte pubkey + 96-byte signature across 5 slots):

- Slot 0: `pubkey[0:32]`
- Slot 1: `pubkey[32:48]` (left-shifted 128 bits) | `sig[0:16]`
- Slots 2-4: `sig[16:96]`

Assembly-based read/write for gas efficiency. `removeKeysSigs()` uses swap-delete with the last key.

11.2 QueueLib.sol

FIFO queue with batch tracking. `enqueue()` appends to tail; `dequeue()` / `peek()` operate on head. `clean()` removes stale batches using a caller-provided lookup map.

11.3 GIndex.sol + SSZ.sol

GIndex — typed `uint256` representing generalized indices in SSZ Merkle trees. Operations: `parent()`, `sibling()`, `concat()`, `isLeft()`, `height()`.

SSZ — Merkle proof verification for beacon chain types:

- `verifyProof(leaf, proof, root, gIndex)` — single-leaf proof.
- `hashTreeRoot(Withdrawal)`, `hashTreeRoot(Validator)`, `hashTreeRoot(BeaconBlockHeader)` — SSZ container hashing per spec.
- Uses SHA-256 via Yul `staticcall` to precompile `0x02`.

11.4 TransientUintUintMapLib.sol

EIP-1153 transient storage map (`uint256 → uint256`). Used in `cleanDepositQueue()` to track per-operator key counts within a single transaction without persisting to storage.

```
function set(mapping uint256 => uint256 storage, uint256 key, uint256 value) //
tstore
function get(mapping uint256 => uint256 storage, uint256 key)                //
tload
```

11.5 ValidatorCountsReport.sol

Parsing utility for the compact binary format used in `updateExitedValidatorsCount()`. Decodes `(nodeOperatorId, exitedCount)` pairs from packed byte arrays.

11.6 AssetRecovererLib.sol

Provides `recoverEther()`, `recoverERC20()`, `recoverStETHShares()`, `recoverERC721()`, `recoverERC1155()` callable by designated `RECOVERER_ROLE`. Uses `SafeERC20`, `ILido.transferShares()` for shares-based stETH recovery.

---

12. Access Control & Roles

All role-based access uses `AccessControlEnumerable` (OpenZeppelin).

CSModule Roles

Role	Holder	Purpose
`STAKING_ROUTER_ROLE`	StakingRouter	Deposit data, validator count updates
`REPORT_EL_REWARDS_STEALING_PENALTY_ROLE`	CSM Committee Multisig	Report MEV theft
`SETTLE_EL_REWARDS_STEALING_PENALTY_ROLE`	Easy Track	Settle expired locks
`VERIFIER_ROLE`	CSVerifier	Submit withdrawal proofs
`CREATE_NODE_OPERATOR_ROLE`	Gate contracts	Create operators
`PAUSE_ROLE` / `RESUME_ROLE`	Lido DAO / GateSeal	Emergency pause
`RECOVERER_ROLE`	Lido DAO	Asset recovery

CSAccounting Roles

Role	Purpose
`MANAGE_BOND_CURVES_ROLE`	Add / update bond curves
`SET_BOND_CURVE_ROLE`	Assign curve to operator
`PAUSE_ROLE` / `RESUME_ROLE`	Emergency pause
`RECOVERER_ROLE`	Asset recovery

CSEjector Roles

Role	Purpose
`PAUSE_ROLE` / `RESUME_ROLE`	Pause voluntary/strike ejections
`RECOVERER_ROLE`	Asset recovery

Trusted Entities Summary

- **Lido DAO**: Admin functions, upgrades, parameter management.
- **CSM Committee Multisig**: MEV theft reporting, bond curve and parameter management.
- **Easy Track**: Settlement of expired EL stealing penalties.
- **Oracle Members**: Submit fee distribution and strikes Merkle trees.
- **CSVerifier**: Submit withdrawal proofs (permissioned at contract level).

---

13. Cross-Contract Interaction Flows

Flow A: Operator Onboarding & Key Activation

```
User
 │
 ├─ (ETH/stETH/wstETH) ──► VettedGate (or PermissionlessGate)
 │                          │
 │                          ├─ verify Merkle proof (vetted only)
 │                          ├─ CSModule.createNodeOperator(from, props, referrer)
 │                          │     └─ emits NodeOperatorAdded
 │                          └─ CSModule.addValidatorKeysETH(from, noId, count,
keys, sigs)
 │                                ├─ CSAccounting.getRequiredBondForNextKeys()
 │                                │     └─
CSBondCurve._getBondAmountByKeysCount()
 │                                ├─ CSAccounting.depositETH() → stETH shares
stored
 │                                ├─ SigningKeys.saveKeysSigs()
 │                                ├─ _updateDepositableValidatorsCount()
 │                                └─ _enqueueNodeOperatorKeys() →
QueueLib.enqueue()
```

```
StakingRouter
 └─ CSModule.obtainDepositData(count)
       ├─ QueueLib.peek() / dequeue() per priority queue
       ├─ SigningKeys.loadKeysSigs()
       └─ returns (pubkeys, sigs) to StakingRouter → DepositContract
```

Flow B: Voluntary Exit

```
Node Operator
 └─ CSEjector.voluntaryEject(noId, startFrom, count, refundRecipient) {payable}
       ├─ validate: manager/rewardAddress caller
       ├─ validate: keys deposited, not withdrawn
       ├─ CSExitPenalties.processTriggeredExit(noId, pubkey, paidFee, VOLUNTARY=0)
       │     └─ stores withdrawalRequestFee = 0 (voluntary → no fee)
       └─ TriggerableWithdrawalsGateway.triggerFullWithdrawals(pubkeys) [EIP-7002]

[Beacon Chain processes exit, credits ETH to Lido withdrawal vault]

CSVerifier
 └─ processWithdrawalProof(header, witness, noId, keyIndex)
       ├─ EIP-4788 lookup: stateRoot for header.slot
       ├─ SSZ.verifyProof(withdrawal, withdrawalProof, stateRoot)
       ├─ SSZ.verifyProof(validator, validatorProof, stateRoot)
       └─ CSModule.submitWithdrawals([{noId, keyIndex, amount}])
            ├─ mark _isValidatorWithdrawn[pointer] = true
            ├─ totalWithdrawnKeys++
            ├─ collect penalties from CSExitPenalties
            ├─ CSAccounting.penalize(noId, totalPenalty)
            └─ _updateDepositableValidatorsCount(noId)
```

Flow C: Bad Performance Ejection

```
CSM Performance Oracle (off-chain)
 └─ CSFeeOracle.submitReportData(ReportData{strikesTreeRoot, ...})
       └─ CSStrikes.processOracleReport(strikesTreeRoot, treeCid)
            └─ treeRoot = strikesTreeRoot

Anyone (permissionless)
 └─ CSStrikes.processBadPerformanceProof(keyStrikesList, proof, flags,
refundRecipient) {payable}
       ├─ verify Merkle multi-proof against treeRoot
       ├─ for each KeyStrikes: sum(data) >= threshold(curveId)?
       ├─ CSExitPenalties.processStrikesReport(noId, pubkey)
       │     └─ strikesPenalty.value = getBadPerformancePenalty(curveId)
       └─ CSEjector.ejectBadPerformer(noId, keyIndex, refundRecipient)
            └─ TriggerableWithdrawalsGateway.triggerFullWithdrawals([pubkey])
```

Flow D: EL Rewards Stealing Penalty Full Lifecycle

```
CSM Committee
 └─ CSModule.reportELRewardsStealingPenalty(noId, blockHash, stolenAmount)
       └─ CSAccounting.lockBondETH(noId, stolenAmount + additionalFine)
             └─ CSBondLock._lock(noId, totalAmount)  [sets until = now + period]

Option A: Operator compensates
 └─ CSModule.compensateELRewardsStealingPenalty(noId) {payable:
compensationAmount}
       ├─ ETH → elRewardsVault
       └─ CSAccounting.releaseLockedBondETH(noId, amount)
             └─ CSBondLock._reduceAmount(noId, amount)

Option B: Committee cancels
 └─ CSModule.cancelELRewardsStealingPenalty(noId, amount)
       └─ CSAccounting.releaseLockedBondETH(noId, amount)

Option C: Lock expires, Easy Track settles
 └─ CSModule.settleELRewardsStealingPenalty([noId])
       └─ CSAccounting.settleLockedBondETH(noId)
             ├─ if lock.until < block.timestamp: burn lock.amount shares via
IBurner
             └─ _updateDepositableValidatorsCount(noId)
```

Flow E: Fee Distribution & Reward Claiming

```
CSM Fee Oracle
 └─ CSFeeOracle.submitReportData({treeRoot, distributed, rebate, ...})
       └─ CSFeeDistributor.processOracleReport(treeRoot, treeCid, logCid,
distributed, rebate, refSlot)
             ├─ totalClaimableShares += distributed
             └─ transfer rebate shares to rebateRecipient

Node Operator
 └─ CSAccounting.claimRewardsStETH(noId, amount, cumulativeFeeShares, proof)
       ├─ _pullFeeRewards(noId, cumulativeFeeShares, proof)
       │     └─ CSFeeDistributor.distributeFees(noId, cumulativeFeeShares, proof)
       │           ├─ verify Merkle proof (leaf = keccak256(keccak256(noId ||
shares)))
       │           ├─ delta = cumulativeFeeShares - distributedShares[noId]
       │           ├─ distributedShares[noId] = cumulativeFeeShares
       │           └─ transfer delta shares to CSAccounting → _increaseBond(noId,
delta)
       ├─ claimable = currentShares - requiredShares
       └─ transfer min(requested, claimable) shares to caller
```

---

14. Key Invariants

Bond Accounting

- `sum(bondShares[noId]) == CSBondCore.totalBondShares` (not including pending burns).
- `getBond(noId) >= getBondAmountByKeysCount(nonWithdrawnDepositedKeys, curve)` when no active lock.
- When a lock is active: `getBond(noId) >= getBondAmountByKeysCount(keys, curve) + lockedAmount`.

Key Lifecycle

- `totalDepositedKeys >= totalWithdrawnKeys` always.
- `totalVettedKeys >= totalDepositedKeys` always (vetting cannot go below deposited).
- `totalAddedKeys >= totalVettedKeys >= totalDepositedKeys >= totalWithdrawnKeys`.
- Deposited keys cannot be removed — `removeKeys()` enforces `startIndex >= totalDepositedKeys`.

Queue Integrity

- `NO.enqueuedCount >= sum(keys in NO's active batches)` across all queues.
- A key is in at most one queue (priority or lowest) at any given time.
- `enqueuedCount = 0` when operator has no keys in any queue.

Withdrawal Idempotency

- `_isValidatorWithdrawn[pointer]` set once and never unset — prevents double-counting.
- `distributedShares[noId]` is strictly monotonic — fees can only be claimed forward.

Exit Penalty Idempotency

- `delayPenalty.isValue`, `strikesPenalty.isValue`, `withdrawalRequestFee.isValue` — each set once per validator key; subsequent calls are no-ops.

Oracle Tree Integrity

- `treeRoot` in CSFeeDistributor and CSStrikes can only be updated by oracle consensus.
- CSFeeDistributor rejects duplicate `(treeRoot, treeCid)` pairs.
- CSStrikes can receive an empty root (wiping tree) due to `strikesLifetime` rollover.

---

15. My Findings

This section documents audit reports that I submitted during the Code4rena contest.

[M-01] CSModule::migrateToPriorityQueue() Missing Access Control Allows Unauthorized Economic Manipulation

Status: Rejected (non-valid).

Description:
The `migrateToPriorityQueue()` function in `CSModule.sol` has no access control, allowing any external user to force node operator migration to priority queues without owner consent. This breaks the protocol's permission model and enables targeted griefing. The function is `external` with no access modifiers:

```
function migrateToPriorityQueue(uint256 nodeOperatorId) external {
    // No access control
    NodeOperator storage no = _nodeOperators[nodeOperatorId];
    // ...
}
```

Impact:
Unauthorized economic manipulation: attackers can alter victim operators' staking strategies (control loss, targeted griefing, competitive disadvantage).

Proof of Code:
Minimal changes to base `PoC.t.sol`: add import `NodeOperatorManagementProperties` from `ICSModule.sol`; in `setUp()` comment out `deployParams = parseDeployParams(env.DEPLOY_CONFIG)`; change `test_PoC` from `view` to non-view. Test `test_RandomUserCanForceMigrateToPriorityQueue` creates a node operator via permissionless gate, adds keys, configures a priority queue via parameters registry, then calls `csm.migrateToPriorityQueue(nodeOperatorId)` as a random attacker to show unauthorized migration.

Recommended mitigation:
Restrict to operator owner only (e.g. only the operator's `managerAddress` or authorized role can call `migrateToPriorityQueue(nodeOperatorId)`).

[H-01] Node Operators Can Evade Penalties Through Gas Based Front-Running of CSEjector::ejectBadPerformer

Status: Rejected (non-valid).

Description:
The Ejector contract is susceptible to front-running: when the strikes role triggers `ejectBadPerformer` to penalize a validator, an operator can submit `voluntaryEject` with a higher gas price. Miners execute `voluntaryEject` first, resetting validator state and avoiding the penalty. There is no locking mechanism during penalty evaluation.

Impact:
Penalty avoidance, unpunished underperformance, potential slashing losses, eroded trust. Likelihood high where mempool front-running is feasible; economic risk significant.

Proof of Code:
Test `test_PenaltyEvasionViaGasFrontRunning_Simplified`: setup operator, deposited keys via staking router; step 1 execute `ejectBadPerformer` at 20 gwei; step 2 front-run with `voluntaryEject` at 50 gwei to show evasion.

Recommended mitigation:
Temporary node lock on penalty evidence: when evidence for a penalty is submitted, set a flag (e.g. `isPenaltyPending`) so that `voluntaryEject` reverts for that operator until the penalty is resolved; clear the lock after penalty is applied.

15.2 Valid Low findings

[L-01] Batch Operation Fully Reverts on Invalid Ejection Entry

Type: Logic Error / Design Flaw

Summary:
`processBadPerformanceProof` in `CSStrikes` reverts the **entire** transaction if any single validator in the batch does not meet the strike threshold for ejection. The revert happens inside `_ejectByStrikes` when `strikes < threshold` (revert `NotEnoughStrikesToEject()`). All other valid ejections in the same batch are rolled back.

Location:
`CSStrikes.sol` (e.g. L243–L245):

```
if (strikes < threshold) {
    revert NotEnoughStrikesToEject();
}
```

Impact:
Operational inefficiency: entire batch must be filtered off-chain and resubmitted; delays enforcement against poorly performing validators. A single invalid entry causes unnecessary tx failures.

PoC:
Implemented in `CSStrikesProofTest`. Batch: Key1 (sufficient strikes), Key2 (sufficient strikes), Key3 (insufficient strikes). `processBadPerformanceProof` reverts on Key3; no validator is ejected. Test: `test_processBadPerformanceProof_RevertWhen_OneOfManyHasNotEnoughStrikes`.

Recommendation:
Process all valid entries and **skip** invalid ones instead of reverting; emit an event for each skipped entry so valid reports are handled without delay and skipped entries can be tracked off-chain:

```
if (strikes >= threshold) {
    ejector.ejectBadPerformer{ value: value }(...);
    EXIT_PENALTIES.processStrikesReport(...);
} else {
    // emit event for skipped entry
}
```

[L-02] Mid-Season Merkle Root Update Invalidates Previously Valid Proofs

Type: Operational / Design Limitation

Summary:
`VettedGate` stores a single active `merkleRoot` and verifies referral claims against it. If the root is updated mid-season (e.g. via `setTreeParams(newRoot, newCid)`), all proofs generated against the **previous** root

become invalid and cannot be used to claim rewards. Merkle proofs are root-specific; any root change requires new proofs for all eligible participants.

Location:
`VettedGate.sol` L291–297: `vettedGate.setTreeParams(newRoot, newCid)`. `claimReferrerBondCurve` always verifies against the current root; historical roots are not stored.

Example:
Season starts with tree `[NodeOperator, Stranger, AnotherNodeOperator]`. Stranger becomes eligible. Admin updates root to `[Stranger, NodeOperator]` (e.g. to ban AnotherNodeOperator). Stranger's old proof (index 1 in old tree) → `InvalidProof`. Stranger must obtain new proof (e.g. index 0 in new tree).

Impact:
Operational: every root change forces all still-eligible participants to obtain a new proof before claiming. Timing risk: claims can fail until new proofs are distributed. UX: unexpected `InvalidProof` for users unaware of the update.

PoC:
In `VettedGateReferralProgramTest`: `test_proofBreaksAfterRootUpdate_whenIndexShifts` — get proof for index 1, build new tree with different order, call `setTreeParams`, then old proof reverts with `InvalidProof` and new proof succeeds.

Recommendations:

- Document in admin/operator playbook: off-chain systems should re-generate and distribute proofs upon root update.
- Optionally consider keyed / index-stable design (e.g. sparse Merkle tree) to reduce proof regeneration cost.
- Optionally store previous root(s) temporarily and accept them for a grace period.
- A stricter option: block `setTreeParams` when `isReferralProgramSeasonActive == true` (reduces mid-season root updates but limits operational flexibility, e.g. urgent malicious address removal).