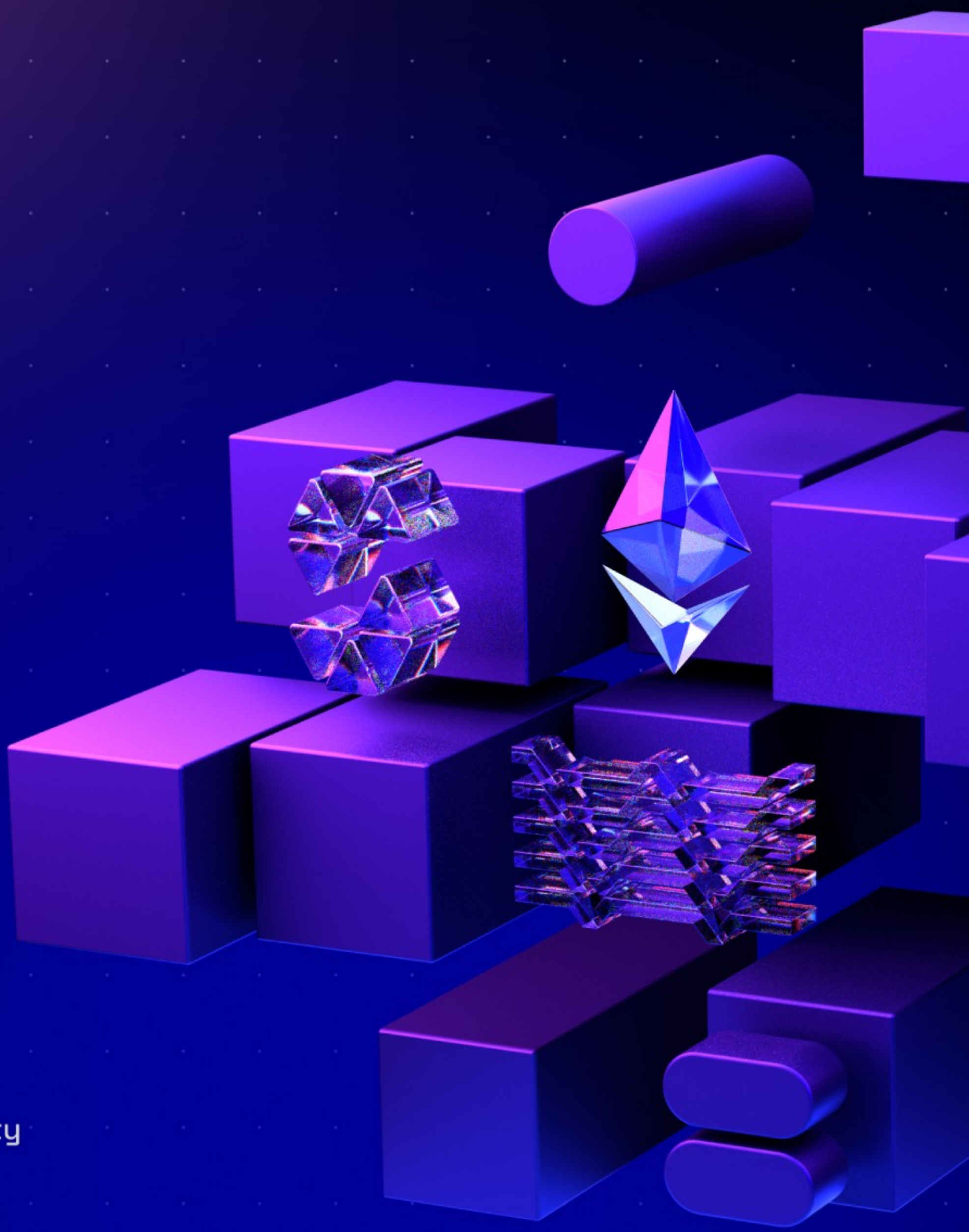
Lido

CSM v2

22.9.2025



Ackee Blockchain Security

Contents

1. Document Revisions

1.0-draft	Draft Report	03.07.2025
1.1-draft	Fix Review Draft	23.07.2025
1.1	Fix Review	24.07.2025
1.2	Mitigation Review	12.08.2025
2.0	Final Report	16.09.2025
2.1	Deployment Verification	22.09.2025

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.

Rohanske nabrezi 717/4

186 00 Prague, Czech Republic

https://ackee.xyz

hello@ackee.xyz

2.2. Audit Methodology

1. **Verification of technical specification**

   The audit scope is confirmed with the client, and auditors are onboarded to the project. Provided documentation is reviewed and compared to the audited system.

2. **Tool-based analysis**

   A deep check with Solidity static analysis tool Wake in companion with Solidity (Wake) extension is performed, flagging potential vulnerabilities for further analysis early in the process.

3. **Manual code review**

   Auditors manually check the code line by line, identifying vulnerabilities and code quality issues. The main focus is on recognizing potential edge cases and project-specific risks.

4. **Local deployment and hacking**

   Contracts are deployed in a local Wake environment, where targeted attempts to exploit vulnerabilities are made. The contracts' resilience against various attack vectors is evaluated.

5. **Unit and fuzz testing**

   Unit tests are run to verify expected system behavior. Additional unit or fuzz tests may be written using Wake framework if any coverage gaps are identified. The goal is to verify the system's stability under real-world conditions and ensure robustness against both expected and unexpected inputs.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Michal Převrátil	Lead Auditor
Lukáš Rajnoha	Auditor
Martin Veselý	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Lido's Community Staking Module v2 is the first major upgrade to the permissionless staking module of the Lido protocol. The upgrade introduces gates and extensions for creating and managing node operators, finer parameter configuration based on assigned bond curves, a new striking system with penalization logic, and support for EIP-7002 execution layer triggerable validator withdrawals.

Revision 1.0

Lido engaged Ackee Blockchain Security to perform a security review of Lido CSM v2 with a total time donation of 45 engineering days in a period between May 27 and July 4, 2025, with Michal Převrátil as the lead auditor. 12 engineering days were dedicated to manually-guided fuzzing using the Wake testing framework.

The audit was performed on commit `59dc784`[1] in the community-staking-module repository and the scope included all Solidity files in the `src` directory, except for `src/interfaces`.

We began our review by implementing and executing manually-guided differential fuzz tests in Wake testing framework to verify the correctness of the new code. More details about the fuzzing process can be found in Report Revision 1.0. Fuzzing was conducted both with re-deployed contracts and with the contracts forked from the mainnet and upgraded to the new version. Additionally, we fuzzed CSM v2 with the Lido core to ensure the compatibility and overall correctness of the system as a whole. Fuzzing of the contracts yielded findings H2, L1, L3, W8, and W10.

In parallel, we performed a thorough manual review of the code, especially focusing on new code changes since the last audit (commit `3469910`[2]). During the review, we focused on the following aspects:

- upgrade of existing contracts is smooth with no storage collisions;
- new format of bond curves is correctly implemented;
- there are no possible reentrancy attacks across the Lido protocol;
- striking and penalization logic cannot be abused;
- migration of queue slots is correctly implemented;
- CSM gates and extensions do not have too relaxed access controls;
- there are no griefing attacks possible;
- new functions for bond deposits cannot be abused; and
- there are no common issues such as data validation.

We performed the manual review in parallel and with regular communication with the Triggerable Withdrawals audit team from Ackee Blockchain Security. All issues of possibly low severity or higher were immediately reported to the Lido team. These issues include the report date in their descriptions in this document. We concluded the review using static analysis tools, including Wake.

Our review resulted in 20 findings, ranging from Info to High severity. The most severe findings H1 and H2 pose a direct denial of service for critical functionality of the protocol — the ability to deposit validator keys and add new validator keys to a node operator.

The majority of severe findings revolve around the key migration functionality. The code otherwise is of high quality and is well-documented. A few findings, namely M1 and L1, have overlap with the core of the Lido protocol. The findings M1, W4 and W8 were also discovered by the Lido team during the audit.

Ackee Blockchain Security recommends Lido:

- ensure contract upgrade and initialization are atomic to prevent front-running attacks possibly leading to loss of control over the contract;
- perform a thorough security review of CSM extensions added in the future; and
- ensure `CSModule.submitWithdrawals` can never be called with zero items or more than 1 item permissionlessly in the future.

See Report Revision 1.0 for the system overview and trust model.

Revision 1.1

Lido engaged Ackee Blockchain Security to perform a fix review of the findings from the previous revision.

The review was performed between July 10 and July 23, 2025 on the commit `d63d123`[3]. Except for the fixes, the reviewed commit contained additional minor changes (e.g., documentation updates, typo fixes) and one major change related to processing historical withdrawal proofs in the `CSVerifier` contract. These changes were reviewed by Ackee Blockchain Security as well.

From the reported 20 findings:

- 16 findings were fixed;
- 3 findings were acknowledged; and
- 1 finding was fixed partially.

No new findings were discovered.

Revision 1.2

Ackee Blockchain Security reviewed the `CSVerifier` EIP-4788 proving logic on the commit `d63d123`[4] in the context of https://research.lido.fi/t/security-disclosure-post-mortem-csverifier-weak-validation-of-the-historical-block-gindex-user-funds-remain-safe/10466. The review focused on the looseness of all input parameters and ensuring all parameters are either necessary proof-related information restricted to only reasonable values or information that is derived and verified from the proof.

The review concluded that all non-proof related parameters are correctly verified and that the proof-related parameters are restricted to only a limited set of values needed for correct verification of the proof. The described attack vector was fully mitigated.

The `CSVerifier` code still allows for a well-documented, highly unlikely scenario when a partial validator withdrawal may pass through all checks and be submitted to the `CSModule` contract as a full withdrawal — https://github.com/lidofinance/community-staking-module/blob/d63d123f24e2ed2fb2f039238e7562a3d61532b2/src/CSVerifier.sol#L301-L322.

Revision 2.0

Lido engaged Ackee Blockchain Security to perform a review of changes made since the previous revision with a total time donation of 0.5 engineering days in a period between September 8 and September 16, 2025, with Michal Převrátil as the lead auditor.

The audit was performed on the commit `0e4b562`[5] in the community-staking-module repository and the scope was all changes made to the `src` directory since the previous revision, except for `src/interfaces`.

Except for documentation updates and refactoring, the changes included:

- additional data validation in voluntary key ejection functions;
- unified handling of soft and hard validator limit mode with respect to unbonded keys;

- reordering of external calls upon key ejection due to strikes, preventing attempts to exploit reentrancy;
- improved calculation of historical block root gIndex through third intermediary slot; and
- additional check preventing changes of node operator reward address to the same value.

The review began with deep analysis of the changes made since the previous revision. The manual review focused on ensuring that the changes did not introduce new vulnerabilities. After the manual review, fuzz tests prepared in previous revisions were updated and run to ensure the correctness of the changes. The review concluded with running static analysis tools, including Wake.

No new findings were discovered. The project is of high quality and is ready for deployment.

Revision 2.1

Lido engaged Ackee Blockchain Security to perform deployment verification of Community Staking Module v2 on the Ethereum mainnet. The verification was performed on the same commit as in the previous revision, `0e4b562`[6].

The verification concluded successfully with an exact bytecode match and reasonable initialization values for all of the following contracts on the Ethereum mainnet:

- QueueLib: 0x6eFF460627b6798C2907409EA2Fdfb287Eaa2e55
- NOAddresses: 0xE4d5a7be8d7c3db15755061053F5a49b6a67fFfc
- CSParametersRegistry: 0x25fdc3be9977cd4da679df72a64c8b6bd5216a78
    - proxy: 0x9d28ad303c90df524ba960d7a2dac56dcc31e428

- CSAccounting: 0x6f09d2426c7405c5546413e6059f884d2d03f449
- PermissionlessGate: 0xcf33a38111d0b1246a3f38a838fb41d626b454f0
- VettedGate: 0x65d4d92cd0eabaa05cd5a46269c24b71c21cfdc4
  - proxy for Identified Community Stakers: 0xb314d4a76c457c93150d308787939063f4cc67e0
- VettedGateFactory: 0xfdab48c4d627e500207e9af29c98579d90ea0ad4
- CSFeeDistributor: 0x5dcf7cf7c6645e9e822a379df046a8b0390251a1
- CSModule: 0x1eb6d4da13ca9566c17f526ae0715325d7a07665
- CSStrikes: 0x3e5021424c9e13fc853e523cd68ebbec848956a0
  - proxy: 0xaa328816027f2d32b9f56d190bc9fa4a5c07637f
- CSFeeOracle: 0xe0b234f99e413e27d9bc31abba9a49a3e570da97
- CSExitPenalties: 0xda22fa1cea40d05fe4cd536967afdd839586d546
  - proxy: 0x06cd61045f958a209a0f8d746e103ecc625f4193
- CSEjector: 0xc72b58aa02e0e98cf8a4a0e9dce75e763800802c
- CSVerifier: 0xdc5fe1782b6943f318e05230d688713a560063dc

The verification was performed before the final upgrade of existing contract proxies to CSM v2. The deployment verification script is available at https://github.com/Ackee-Blockchain/tests-lido-csm-v2.

The deployed contracts were tested through forking with fuzz tests prepared in the previous revisions, with an exception of the CSVerifier contract, which would result in undesirably slow fuzzing. No issues were detected during the fuzz testing.

[1] full commit hash: `59dc7845660bef7299bf8cc97f9c831f5588a8ba`, link to commit

[2] full commit hash: `3469910c0d29a54b37d0c4de3cf527a3e7be2099`, link to commit

[3] full commit hash: `d63d123f24e2ed2fb2f039238e7562a3d61532b2`, link to commit

[4] full commit hash: `d63d123f24e2ed2fb2f039238e7562a3d61532b2`, link to commit

[5] full commit hash: `0e4b562719cca51070c9cede5e5a8505eca18684`, link to commit

[6] full commit hash: `0e4b562719cca51070c9cede5e5a8505eca18684`, link to commit

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
0	2	1	3	10	4	20

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
H1: Deposits denial of service	High	1.0	Fixed
H2: Incorrect enqueued keys accounting	High	1.0	Fixed
M1: `processExitDelayReport` griefing	Medium	1.0	Fixed
L1: Bond burning denial of service	Low	1.0	Fixed
L2: Depositable count not updated in `migrateToPriorityQueue`	Low	1.0	Fixed

Finding title	Severity	Reported	Status
L3: Incorrect number of migrated keys	Low	1.0	Fixed
W1: Duplicate `getSigningKeys` call in `processBadPerformanceProof`	Warning	1.0	Acknowledged
W2: Depositable validators count can be updated on non-existing node operators	Warning	1.0	Acknowledged
W3: `ValidatorWithdrawalInfo` struct contains unused `isSlashed` field	Warning	1.0	Fixed
W4: `_validateKeyNumberValueIntervals` in `CSParametersRegistry` does not check for empty arrays	Warning	1.0	Fixed
W5: Node operators can self-refer	Warning	1.0	Fixed
W6: Invalid tagging of memory-safe assembly	Warning	1.0	Fixed
W7: Node operator creation transient flag not cleared	Warning	1.0	Fixed
W8: Division by zero in `processBadPerformanceProof`	Warning	1.0	Fixed
W9: `submitWithdrawals` griefing	Warning	1.0	Fixed
W10: Unconditional emission of `ReferralRecorded` event	Warning	1.0	Fixed

Finding title	Severity	Reported	Status
I1: `Lido` and `StETH` ambiguous naming	Info	1.0	Acknowledged
I2: Typos	Info	1.0	Fixed
I3: Inconsistent `unchecked` use in `CSBondCurve`	Info	1.0	Fixed
I4: Unused code	Info	1.0	Partially fixed

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Michal Převrátil	Lead Auditor
Lukáš Rajnoha	Auditor
Martin Veselý	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

Community Staking Module v2 is the first major upgrade to the first permissionless staking module of the Lido protocol. It allows node operators to create Ethereum validators with much lower costs than when doing solo staking. Node operators provide ETH, stETH or wstETH to increase their bond, serving as security collateral, to cover their validator keys. The bond is internally converted to stETH, so node operators receive rewards in form of stETH rebases. The required bond amount depends on the bond curve assigned to the node operator.

Validator keys and signatures must be valid to be picked up by Lido's Staking Router and deposited. Node operators are allowed to remove their validator keys if not deposited yet, but with a charge applied to the bond to cover operational costs of Lido and prevent denial of service attacks.

Apart from stETH rebases, node operators receive rewards for operating validators on their behalf. In Lido, rewards are socialized among all staking modules. CSM rewards are then distributed based on the performance of each operator's validators. Received rewards and excessive bond can be withdrawn using the pull mechanism.

Trust Model

CSM allows permissionless entry of node operators. Identified independent community members can join through the vetted gate offering additional benefits. Node operators are allowed to add new keys, remove keys that are not yet deposited, increase their bond and claim rewards. Node operators may change their manager address and the address that receives rewards.

Lido is expected to run an off-chain service to validate if execution layer rewards and MEV are sent to Lido's vault. In the opposite case, Lido is allowed to report execution layer rewards stealing, locking the adequate part of the operator's bond. The stealing report then may be compensated by the node operator, either cancelled or settled by Lido, or timed out effectively unlocking the bond.

Another off-chain service run by Lido is responsible for validation of validator keys and signatures waiting to be deposited. Invalid keys with signatures are marked as unvetted, which prevents them and all keys added after them to be deposited. Node operators are then required to remove the invalid keys and pay a charge to cover the operational costs of Lido.

A new off-chain service tracks validators performance and may strike them for bad performance. Validators with enough strikes can be permissionlessly ejected from the module through EIP-7002.

Validators may also eject voluntarily with EIP-7002.

Lido is allowed to change the bond curves describing the relationship between the number of validator keys and the required bond amount. Changing the bond curve to less beneficial conditions may trigger forced exits of validators. Lido may also change the bond curve of a single node operator. The assignment of the bond curve also influences various other parameters, such as the strikes threshold for ejection.

Validator withdrawals may be reported permissionlessly through EIP-4788 proofs. Validators who were requested to exit and did not do so within the specified time can be permissionlessly reported using the EIP-4788 proofs. Such validators are charged an additional penalty.

Arbitrary extensions may be attached to CSM allowing for custom logic of creating and managing node operators. These extensions have to be trusted and approved by Lido. A security review of the extensions is required.

Fuzzing

Manually-guided differential fuzz tests were developed during the review to test the correctness and robustness of the system. The fuzz tests employ fork testing technique to test the system with external contracts exactly as they are deployed on the mainnet. This is crucial to detect any potential integration issues.

The differential fuzz tests keep their own Python state according to the system's specification. Assertions are used to verify the Python state against the on-chain state in contracts.

The list of all implemented execution flows and invariants is available in Appendix B.

The fuzz test was integrated with a Triggerable Withdrawals fuzz test prepared by Ackee Blockchain Security during a parallel audit to ensure compatibility and integration of the two systems.

One of the most severe findings, H2, was discovered using fuzz testing in Wake testing framework.

The full source code of all fuzz tests is available at https://github.com/Ackee-Blockchain/tests-lido-csm-v2.

Findings

The following section presents the list of findings discovered in this revision.

For the complete list of all findings, Go back to Findings Summary

H1: Deposits denial of service

High severity issue

Impact:	High	Likelihood:	Medium
Target:	CSModule.sol	Type:	Denial of service
Reported on:		June 7, 2025	

Description

The `migrateToPriorityQueue` function in the `CSModule` contract allows migration of validator keys into a higher priority queue. The function only reverts if the node operator has already performed the migration. However, it is possible to permissionlessly call the function with the ID of a node operator that has no keys to migrate or does not even exist.

Listing 1. Excerpt from [CSModule.migrateToPriorityQueue]()

```
NodeOperator storage no = _nodeOperators[nodeOperatorId];

if (no.usedPriorityQueue) {
    revert PriorityQueueAlreadyUsed();
}

uint256 curveId = accounting().getBondCurveId(nodeOperatorId);
(uint32 priority, uint32 maxDeposits) = PARAMETERS_REGISTRY
    .getQueueConfig(curveId);

if (priority < QUEUE_LEGACY_PRIORITY) {
    uint32 deposited = no.totalDepositedKeys;

    if (maxDeposits > deposited) {
        uint32 toMigrate = uint32(
            Math.min(maxDeposits - deposited, no.enqueuedCount)
        );

        unchecked {
            no.enqueuedCount -= toMigrate;
        }
        _enqueueNodeOperatorKeys(nodeOperatorId, priority, toMigrate);
```

```
    }

    no.usedPriorityQueue = true;
}
_incrementModuleNonce();
```

The `migrateToPriorityQueue` function increments the Community Staking Module (CSM) nonce. The nonce is used to track validator deposit data changes. [Deposit Security Module]() guardians validate the deposit data of each validator and provide a signature guaranteeing the deposit data is correct and can be deposited into the Eth 2.0 [Deposit Contract]().

Listing 2. Excerpt from [DepositSecurityModule.depositBufferedEther]()

```
/// @dev The second most likely reason for the signature to go stale
uint256 onchainNonce =
    STAKING_ROUTER.getStakingModuleNonce(stakingModuleId);
if (nonce != onchainNonce) revert ModuleNonceChanged();
```

Exploit scenario

Given the permissionless nature of the `migrateToPriorityQueue` function and the fact that the nonce is incremented on each call even with non-existing node operator, an attacker can perform a denial of service attack by repeatedly calling the function and invalidating the Deposit Security Module signatures. This will prevent new validators from being added to the network using the CSM.

Recommendation

Increment the CSM nonce only if the node operator migrated a non-zero number of keys. Revert the execution of the function if the node operator does not exist.

Fix 1.1

The issue was fixed by following the recommendation.

[Go back to Findings Summary]

H2: Incorrect enqueued keys accounting

High severity issue

Impact:	Medium	Likelihood:	High
Target:	CSModule.sol	Type:	Logic error
Reported on:		June 7, 2025	

Description

The `CSModule.migrateToPriorityQueue` function can be used by existing node operators eligible for higher priority queue slots. The function first computes the number of queue slots to migrate and then subtracts that number from the `enqueued` count.

Listing 3. Excerpt from [CSModule.migrateToPriorityQueue]()

```
603 uint32 toMigrate = uint32(
604     Math.min(maxDeposits - deposited, no.enqueuedCount)
605 );
606
607 unchecked {
608     no.enqueuedCount -= toMigrate;
609 }
610 _enqueueNodeOperatorKeys(nodeOperatorId, priority, toMigrate);
```

The subtraction is a hack to enforce the following `_enqueueNodeOperatorKeys` function call to migrate the correct number of queue slots, ignoring the legacy queue slots that are being migrated.

However, the legacy queue slots remain in place, and the `enqueued` count is not incremented by the value originally decremented at line 608. This causes a discrepancy between the `enqueued` count and the number of actual queue slots of a given node operator.

Exploit scenario

Due to the incorrect accounting of the `enqueued` count, one of the following scenarios will occur:

1. The `enqueued` count causes an underflow revert in the `QueueLib.clean` function, posing a denial of service for the cleaning functionality.
2. The `enqueued` count silently underflows in the `CSModule.obtainDepositData` function. The node operator will not be able to deposit new keys as the `enqueued` count will be always greater than the `depositable` count.

Since the `migrateToPriorityQueue` function can be permissionlessly called for any node operator, any malicious actor aware of the issue may intentionally break the accounting of the `enqueued` count for all node operators with a non-zero amount of queue slots allowed to migrate.

Recommendation

Either add the missing increment to the `enqueued` count to achieve the consistent accounting of the `enqueued` count or remove the problematic subtraction and adjust the `_enqueueNodeOperatorKeys` function to migrate the correct number of queue slots.

Fix 1.1

The `migrateToPriorityQueue` function was significantly reworked and now internally uses a direct function to enqueue the keys so that no hack is needed and the issue no longer exists.

Go back to Findings Summary

M1: `processExitDelayReport` griefing

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	CSExitPenalties.sol	Type:	Griefing
Reported on:		June 26, 2025	

Description

The `CSExitPenalties.processExitDelayReport` function is called from the `ValidatorExitDelayVerifier` contract, allowing permissionless reporting of validator exit delays in batches:

Listing 4. Excerpt from *ValidatorExitDelayVerifier.verifyValidatorExitDelay*

```
for (uint256 i = 0; i < validatorWitnesses.length; i++) {
    ValidatorWitness calldata witness = validatorWitnesses[i];

    (bytes memory pubkey, uint256 nodeOpId, uint256 moduleId, uint256 valIndex) = veb.unpackExitRequest(
        exitRequests.data,
        exitRequests.dataFormat,
        witness.exitRequestIndex
    );

    uint256 eligibleToExitInSec = _getSecondsSinceExitIsEligible(
        deliveredTimestamp,
        witness.activationEpoch,
        proofSlotTimestamp
    );

    _verifyValidatorExitUnset(beaconBlock.header, validatorWitnesses[i], pubkey, valIndex);

    stakingRouter.reportValidatorExitDelay(moduleId, nodeOpId, proofSlotTimestamp, pubkey, eligibleToExitInSec);
}
```

The `reportValidatorExitDelay` function reverts if the validator was already reported and a penalty was set.

Listing 5. Excerpt from [CSExitPenalties.processExitDelayReport]()

```
bytes32 keyPointer = _keyPointer(nodeOperatorId, publicKey);
ExitPenaltyInfo storage exitPenaltyInfo = _exitPenaltyInfo[keyPointer];
if (exitPenaltyInfo.delayPenalty.isValue) {
    revert ValidatorExitDelayAlreadyReported();
}
```

Exploit scenario

The permissionless nature of the report function allows an attacker to grief the system by front-running transactions that submit reports with multiple validators. The attacker can submit a report with one of the validators from the front-run transaction. This will cause the front-run transaction to revert and prevent other validators from being reported. The gas for the reverted transaction is wasted.

Recommendation

Modify the `processExitDelayReport` function to return early instead of reverting when a validator was already reported and a penalty was set.

Fix 1.1

The issue was fixed by following the recommendation. The `ValidatorExitDelayAlreadyReported` error was removed from the codebase.

[Go back to Findings Summary]()

L1: Bond burning denial of service

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	CSBondCore.sol	Type:	Denial of service
Reported on:		May 31, 2025	

Description

The `CSBondCore._burn` function is invoked upon execution layer rewards stealing settlement or upon submission of a validator exit proof with application of penalties.

The function first computes the amount of stETH shares to burn. If the amount is zero, the function returns early. Otherwise, the share amount is converted to stETH amount to burn and `Burner.requestBurnMyStETH` is called.

Listing 6. Excerpt from [CSBondCore._burn]()

```
uint256 sharesToBurn = _sharesByEth(amount);
uint256 burnedShares = _reduceBond(nodeOperatorId, sharesToBurn);
// If no bond already
if (burnedShares == 0) {
    return;
}

uint256 burnedAmount = _ethByShares(burnedShares);
IBurner(LIDO_LOCATOR.burner()).requestBurnMyStETH(burnedAmount);
```

The `Burner.requestBurnMyStETH` function first transfers the stETH and then converts the amount to stETH shares again.

Listing 7. Excerpt from [Burner]()

```
function requestBurnMyStETH(uint256 _stETHAmountToBurn) external
onlyRole(REQUEST_BURN_MY_STETH_ROLE) {
```

```
    IStETH(STETH).transferFrom(msg.sender, address(this), _stETHAmountToBurn);
    uint256 sharesAmount = IStETH(STETH).getSharesByPooledEth(_stETHAmountToBurn);
    _requestBurn(sharesAmount, _stETHAmountToBurn, false /* _isCover */);
}
```

Additionally, the `Burner` contract reverts if the amount of stETH shares to burn is zero.

Listing 8. Excerpt from [Burner._requestBurn]()

```
if (_sharesAmount == 0) revert ZeroBurnAmount();
```

Exploit scenario

Alice, a node operator, has bonds that need to be burned due to penalties. Bob, the protocol, processes the burning through `CSBondCore._burn` function. Due to multiple conversions between stETH shares and stETH causing rounding errors, and the fact that the `Burner` contract reverts if the amount of stETH shares to burn is zero, the function may unintentionally revert when `burnedAmount` is less than or equal to one. This conversion between stETH shares and stETH and the consequent loss of precision additionally may lead to dust shares of stETH not belonging to any node operator being left in the contract.

Recommendation

Return early from the `CSBondCore._burn` function if the requested amount of stETH to burn is less than or equal to one. Alternatively, implement `Burner.requestBurnMyShares` function that would take the amount of stETH shares to burn as an argument and would not perform any internal conversions causing rounding errors.

Fix 1.1

The issue was fixed by reverting the new logic and using `Burner.requestBurnShares`.

The suggested function `Burner.requestBurnMyShares` is expected to be implemented in Lido core v3 and to be used in the next version of CSM.

[Go back to Findings Summary]

L2: Depositable count not updated in `migrateToPriorityQueue`

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	CSModule.sol	Type:	Logic error
Reported on:		June 9, 2025	

Description

The `migrateToPriorityQueue` function in the `CSModule` contract allows for migration of validator keys into a higher priority queue. The number of migrated keys also depends on the `depositable` count.

Listing 9. Excerpt from [CSModule._enqueueNodeOperatorKeys]()

```
uint32 count = depositable - enqueued;
count = uint32(Math.min(count, maxKeys));
```

The `depositable` count stored in the contract may be outdated, especially when the node operator's bond lock has expired or when the parameters of the bond curve assigned to the node operator have been changed.

Exploit scenario

A node operator can call `CSModule.updateDepositableValidatorsCount` to update his `depositable` count. However, since the `migrateToPriorityQueue` function is permissionless and may be called with any node operator ID, any malicious actor may intentionally call `migrateToPriorityQueue` without a prior `updateDepositableValidatorsCount` call when the node operator's `depositable` count is outdated and lower than the actual value.

Since the migration of node operator keys may be performed only once per node operator, the malicious actor will enforce migration of lower number of keys than possible.

Recommendation

Call `updateDepositableValidatorsCount` from `migrateToPriorityQueue` or make the migrated number of keys independent of the `depositable` count.

Fix 1.1

The number of migrated keys in the `migrateToPriorityQueue` function no longer depends on the `depositable` count and so the issue no longer exists.

[Go back to Findings Summary]

L3: Incorrect number of migrated keys

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	CSModule.sol	Type:	Logic error
Reported on:		June 15, 2025	

Description

The `CSModule.migrateToPriorityQueue` function can be used by existing node operators eligible for higher priority queue slots. The function first computes the number of keys to migrate as `min(maxDeposits - deposited, enqueuedCount)`, decreases the enqueued count by the number of migrated keys, and calls `CSModule._enqueueNodeOperatorKeys`.

Listing 10. Excerpt from [CSModule.migrateToPriorityQueue]()

```
uint256 curveId = accounting().getBondCurveId(nodeOperatorId);
(uint32 priority, uint32 maxDeposits) = PARAMETERS_REGISTRY
    .getQueueConfig(curveId);

if (priority < QUEUE_LEGACY_PRIORITY) {
    uint32 deposited = no.totalDepositedKeys;

    if (maxDeposits > deposited) {
        uint32 toMigrate = uint32(
            Math.min(maxDeposits - deposited, no.enqueuedCount)
        );

        unchecked {
            no.enqueuedCount -= toMigrate;
        }
        _enqueueNodeOperatorKeys(nodeOperatorId, priority, toMigrate);
    }

    no.usedPriorityQueue = true;
}
```

The `_enqueueNodeOperatorKeys` function then compares the number of keys to migrate with `depositable - enqueued` and uses the minimum of the two values as the final number of keys to migrate.

Listing 11. Excerpt from [CSModule._enqueueNodeOperatorKeys]()

```
function _enqueueNodeOperatorKeys(
    uint256 nodeOperatorId,
    uint256 queuePriority,
    uint32 maxKeys
) internal {
    NodeOperator storage no = _nodeOperators[nodeOperatorId];
    uint32 depositable = no.depositableValidatorsCount;
    uint32 enqueued = no.enqueuedCount;

    if (enqueued < depositable) {
        unchecked {
            uint32 count = depositable - enqueued;
            count = uint32(Math.min(count, maxKeys));

            no.enqueuedCount = enqueued + count;

            QueueLib.Queue storage q = _getQueue(queuePriority);
            q.enqueue(nodeOperatorId, count);
            emit BatchEnqueued(queuePriority, nodeOperatorId, count);
        }
    }
}
```

Exploit scenario

Alice, a node operator, attempts to migrate keys to the priority queue when `maxDeposits - deposited` is less than `enqueuedCount` in the `migrateToPriorityQueue` function. The updated `enqueued` count becomes `enqueued - (maxDeposits - deposited)`, which remains non-zero. The `_enqueueNodeOperatorKeys` function then subtracts the new enqueued count from `depositable` and uses this value as the number of keys to migrate if it is less than `maxDeposits - deposited`. As a result, the number of migrated keys becomes `min(depositable - enqueued + toMigrate, toMigrate)` (where `enqueued` refers to the original value) while the expected number of migrated keys is `min(depositable, toMigrate)`.

Recommendation

Do not account for `enqueued` count in the `_enqueueNodeOperatorKeys` function and compute the number of keys to migrate as `min(depositable, toMigrate)`.

Fix 1.1

The number of migrated keys in the `migrateToPriorityQueue` function no longer depends on the `depositable` count and so the issue no longer exists.

[Go back to Findings Summary]

W1: Duplicate `getSigningKeys` call in `processBadPerformanceProof`

Impact:	Warning	Likelihood:	N/A
Target:	CSStrikes.sol, CSEjector.sol	Type:	Gas optimization

Description

The `processBadPerformanceProof` function from the `CSStrikes` contract first calls the `getSigningKeys` function from the `CSModule` contract to derive the `pubkey` value for the given validator to be ejected. Subsequently, when ejecting validators via the internal `_ejectByStrikes` function, it invokes the `ejectBadPerformer` function on the `CSEjector` contract, where the `pubkey` value is computed again.

Listing 12. Excerpt from [CSStrikes]()

```
function _ejectByStrikes(
    KeyStrikes calldata keyStrikes,
    bytes memory pubkey,
    uint256 value,
    address refundRecipient
) internal {
    uint256 strikes = 0;
    for (uint256 i; i < keyStrikes.data.length; ++i) {
        strikes += keyStrikes.data[i];
    }

    uint256 curveId = ACCOUNTING.getBondCurveId(keyStrikes.nodeOperatorId);

    (, uint256 threshold) = PARAMETERS_REGISTRY.getStrikesParams(curveId);
    if (strikes < threshold) {
        revert NotEnoughStrikesToEject();
    }

    ejector.ejectBadPerformer{ value: value }(
        keyStrikes.nodeOperatorId,
        keyStrikes.keyIndex,
        refundRecipient
    );
```

```
    EXIT_PENALTIES.processStrikesReport(keyStrikes.nodeOperatorId, pubkey);
}
```

Listing 13. Excerpt from [CSEjector]()

```
function ejectBadPerformer(
    uint256 nodeOperatorId,
    uint256 keyIndex,
    address refundRecipient
) external payable whenResumed onlyStrikes {
    // A key must be deposited to prevent ejecting unvetted keys that can intersect with
    // other modules.
    if (
        keyIndex >= MODULE.getNodeOperatorTotalDepositedKeys(nodeOperatorId)
    ) {
        revert SigningKeysInvalidOffset();
    }
    // A key must be non-withdrawn to restrict unlimited exit requests consuming sanity checker
    // limits, although a deposited key can be requested to exit multiple times. But, it will
    // eventually be withdrawn, so potentially malicious behaviour stops when there are no
    // active keys available
    if (MODULE.isValidatorWithdrawn(nodeOperatorId, keyIndex)) {
        revert AlreadyWithdrawn();
    }

    ValidatorData[] memory exitsData = new ValidatorData[](1);
    bytes memory pubkey = MODULE.getSigningKeys(
        nodeOperatorId,
        keyIndex,
        1
    );
```

The `ejectBadPerformer` function contains the `onlyStrikes` modifier and is thus not intended to be called from elsewhere, meaning the `CSStrikes.processBadPerformanceProof` function is its only possible entrypoint.

Since the `processBadPerformanceProof` function might process multiple entries, removing the duplicate computation can save gas.

Recommendation

Pass the `pubkey` value via a function parameter into the `ejectBadPerformer` function instead of recomputing it to save gas.

Acknowledgment 1.1

The Lido team acknowledged the issue with the following comment:

> *Since the `CSEjector` has the authority to eject any key from the protocol, we intentionally retrieve the pubkey again within both `CSStrikes` and `CSEjector`. This redundancy ensures correctness and trust boundaries between contracts, even at the cost of increased gas consumption.*

[Go back to Findings Summary]

W2: Depositable validators count can be updated on non-existing node operators

Impact:	Warning	Likelihood:	N/A
Target:	CSModule.sol	Type:	Logic error

Description

The `updateDepositableValidatorsCount` function in the `CSModule` contract can be called for non-existing node operators. In such a case, the function does nothing. Even though it does not cause any effects, this might not be the intended behavior.

Listing 14. Excerpt from [CSModule]

```
function updateDepositableValidatorsCount(uint256 nodeOperatorId) external {
    _updateDepositableValidatorsCount({
        nodeOperatorId: nodeOperatorId,
        incrementNonceIfUpdated: true
    });
}
```

Recommendation

Add the `_onlyExistingNodeOperator` function check to revert if the node operator does not exist.

Acknowledgment 1.1

The Lido team acknowledged the issue with the following comment:

> *The behavior of this function has remained unchanged since v1, where it was previously named* `normalizeQueue` *([link to source]). At the moment, all contract calls to this function are made in conjunction with* `_onlyExistingNodeOperator`*, so this*

> *behavior is expected and intentional.*

Go back to Findings Summary

W3: `ValidatorWithdrawalInfo` struct contains unused `isSlashed` field

Impact:	Warning	Likelihood:	N/A
Target:	CSModule.sol	Type:	Unused code

Description

The `ValidatorWithdrawalInfo` struct, which is passed into the `submitWithdrawals` function in the `CSModule` contract, contains an unused `isSlashed` field. This field was used before in CSM v1; however, slashing reporting when submitting withdrawal proofs has been removed in CSM v2, making the field obsolete.

Listing 15. Excerpt from [ICSModule]

```
struct ValidatorWithdrawalInfo {
    uint256 nodeOperatorId; // @dev ID of the Node Operator
    uint256 keyIndex; // @dev Index of the withdrawn key in the Node
Operator's keys storage
    uint256 amount; // @dev Amount of withdrawn ETH in wei
    bool isSlashed; // @dev If validator is slashed or not
}
```

Recommendation

Remove the `isSlashed` field from the `ValidatorWithdrawalInfo` struct.

Fix 1.1

The `isSlashed` field was removed from the `ValidatorWithdrawalInfo` struct.

[Go back to Findings Summary]

W4: `_validateKeyNumberValueIntervals` in `CSParametersRegistry` does not check for empty arrays

Impact:	Warning	Likelihood:	N/A
Target:	CSParametersRegistry.sol	Type:	Data validation

Description

The `_validateKeyNumberValueIntervals` function in the `CSParametersRegistry` contract does not check for empty arrays before accessing index 0. This internal function is called by the `setRewardShareData` and `setPerformanceLeewayData` functions, which also do not check for empty arrays. This can lead to an out-of-bounds revert when an empty array is passed.

Listing 16. Excerpt from [CSParametersRegistry]()

```
function _validateKeyNumberValueIntervals(
    KeyNumberValueInterval[] calldata intervals
) private pure {
    if (intervals[0].minKeyNumber != 1) {
        revert InvalidKeyNumberValueIntervals();
    }

    if (intervals[0].value > MAX_BP) {
        revert InvalidKeyNumberValueIntervals();
    }

    for (uint256 i = 1; i < intervals.length; ++i) {
        unchecked {
            if (
                intervals[i].minKeyNumber <= intervals[i - 1].minKeyNumber
            ) {
                revert InvalidKeyNumberValueIntervals();
            }
            if (intervals[i].value > MAX_BP) {
                revert InvalidKeyNumberValueIntervals();
            }
```

```
        }
    }
}
```

Recommendation

Add a check for empty arrays before accessing index 0.

Fix 1.1

The issue was fixed by following the recommendation.

Go back to Findings Summary

W5: Node operators can self-refer

Impact:	Warning	Likelihood:	N/A
Target:	CSVettedGate.sol, CSModule.sol	Type:	Logic error

Description

Referrers can be set to the same address as `msg.sender` (i.e., to self), effectively making the referrer a self-referral. This applies to both `CSVettedGate` and `CSModule` contracts.

Recommendation

Add a check to prevent self-referrals.

Fix 1.1

A check was added to the `CSVettedGate` contract to prevent self-referrals when creating a node operator.

Self-referral possibility is an intended behavior in the `CSModule` contract.

[Go back to Findings Summary]

W6: Invalid tagging of memory-safe assembly

Impact:	Warning	Likelihood:	N/A
Target:	SSZ.sol	Type:	N/A

Description

The `SSZ` library contains multiple inline assembly blocks. These blocks are tagged as `memory-safe` with a simple one-line comment. However, the Solidity compiler only recognizes the `memory-safe` tag in NatSpec comments.

Listing 17. Excerpt from [SSZ]

```
// @solidity memory-safe-assembly
```

Listing 18. Excerpt from [SSZ]

```
// @solidity memory-safe-assembly
```

Listing 19. Excerpt from [SSZ]

```
// @solidity memory-safe-assembly
```

Recommendation

Change

```
// @solidity memory-safe-assembly
```

to

```
/// @solidity memory-safe-assembly
```

in the `SSZ` library.

Fix 1.1

All respective inline assembly blocks were decorated with `assembly ("memory-safe")` for even better clarity.

Go back to Findings Summary

W7: Node operator creation transient flag not cleared

Impact:	Warning	Likelihood:	N/A
Target:	CSModule.sol	Type:	Logic error

Description

Community Staking Module (CSModule) allows creation of node operators and adding validator keys through third-party extensions. Extensions, the vetted gate, and the permissionless gate are allowed to add validator keys only to node operators being created by them in the current transaction.

The validation is performed through a boolean flag saved in the transient storage of the `CSModule` contract.

Listing 20. Excerpt from CSModule

```
function _isOperatorCreatedInTX(
    uint256 nodeOperatorId
) internal view returns (bool) {
    TransientUintUintMap map = TransientUintUintMapLib.load(
        OPERATORS_CREATED_IN_TX_MAP_TSLOT
    );
    return map.get(nodeOperatorId) == 1;
}
```

The flag is set upon the creation of a node operator but is never cleared. This opens the possibility for an attacker to add unintended validator keys to a node operator being created in the current transaction. The attack would be conducted through a less permissioned extension that allows adding validator keys without creating a node operator. As a second precondition, the attacker would need the victim's transaction (actually [ERC-4337](ERC-4337) user operation) bundled with the attacker's user operation into the same transaction, with the victim's transaction executed first.

Alternatively, the attacker could wait for the victim's transaction that performs a multicall, creating a node operator as one of the first calls, and doing an external call to an untrusted contract controlled by the attacker in one of the subsequent calls.

Recommendation

Introduce a limitation that allows extensions and gates to add validator keys only once after the node operator is created. Clear the flag after the keys are added.

Fix 1.1

The implementation was changed so that the node operator creator (`msg.sender`) is saved in the transient storage (instead of a boolean flag) and checked when adding validator keys. The transient storage slot is cleared after the first batch of validator keys is added.

[Go back to Findings Summary]

W8: Division by zero in `processBadPerformanceProof`

Impact:	Warning	Likelihood:	N/A
Target:	CSStrikes.sol	Type:	Data validation

Description

The `processBadPerformanceProof` function in the `CSStrikes` contract performs a modulus operation on line 136 without checking if the divisor is zero:

Listing 21. Excerpt from [CSStrikes.processBadPerformanceProof]()

```
136 if (msg.value % keyStrikesList.length > 0) {
137     revert ValueNotEvenlyDivisible();
138 }
```

When `keyStrikesList.length` is zero, the expression `msg.value % keyStrikesList.length` causes a division by zero panic, causing the transaction to revert with a panic error rather than a descriptive revert message. This results in poor user experience and unclear error reporting.

Recommendation

Add a check to ensure `keyStrikesList.length` is not zero before performing the modulus operation.

Fix 1.1

The issue was fixed by following the recommendation. The function now also checks if `msg.value` is zero and reverts if it is.

[Go back to Findings Summary]()

W9: `submitWithdrawals` griefing

Impact:	Warning	Likelihood:	N/A
Target:	CSModule.sol	Type:	Griefing

Description

The `CSModule.submitWithdrawals` function is called from the `CSVerifier` contract. Currently, the function is always called with exactly one array item.

Listing 22. Excerpt from [CSModule](#)

```
function submitWithdrawals(
    ValidatorWithdrawalInfo[] calldata withdrawalsInfo
) external onlyRole(VERIFIER_ROLE) {
```

However, if the `CSVerifier` implementation changes in the future, two potential vulnerabilities may arise.

1. If a zero-length array is passed, the module nonce is incremented. This may lead to denial of service attacks for the validator key deposit functionality (see H1).
2. If more than one array item is passed, a front-running griefing attack is possible. Since the `submitWithdrawals` logic reverts if the validator was already reported, the attacker may submit a withdrawal of a single validator from the list of validators in the front-run transaction. This causes the front-run transaction to revert without reporting other validators and waste the sender's gas.

Recommendation

Do not increment the module nonce if the array is empty. Skip processing the current array item instead of reverting if the validator was already reported as withdrawn.

Fix 1.1

The issue was fixed by following the recommendation. The `AlreadyWithdrawn` error was removed from the codebase.

Go back to Findings Summary

W10: Unconditional emission of `ReferralRecorded` event

Impact:	Warning	Likelihood:	N/A
Target:	VettedGate.sol	Type:	Logic error

Description

The `ReferralRecorded` event is emitted unconditionally in the `VettedGate.recordReferral` function, even if the referral season is not active.

Listing 23. Excerpt from [VettedGate]()

```
function _bumpReferralCount(
    address referrer,
    uint256 referralNodeOperatorId
) internal {
    uint256 season = referralProgramSeasonNumber;
    if (isReferralProgramSeasonActive && referrer != address(0)) {
        _referralCounts[_seasonedAddress(referrer, season)] += 1;
    }
    emit ReferralRecorded(referrer, season, referralNodeOperatorId);
}
```

The unconditional emission may lead to complicated off-chain tracking of referrals.

Recommendation

Emit the `ReferralRecorded` event only if the referral season is active.

Fix 1.1

The event emission was moved into the conditional block that checks if the referral season is active.

[Go back to Findings Summary]()

I1: `Lido` and `StETH` ambiguous naming

Impact:	Info	Likelihood:	N/A
Target:	CSModule.sol, CSFeeDistributor.sol, CSBondCurve.sol, CSAccounting.sol	Type:	Code quality

Description

The `CSModule` and `CSFeeDistributor` contracts save StETH's contract address under the `STETH` immutable variable, while `CSBondCurve` and `CSAccounting` contracts store it under the `LIDO` variable instead.

This ambiguity might potentially cause confusion for others verifying the code, as the different naming suggests the two contracts are different.

Recommendation

Unify the naming of the `LIDO` and `STETH` variables across all contracts.

Acknowledgment 1.1

The Lido team acknowledged the issue with the following comment:

> *Lido.sol contract inherits from stETH.sol contract. In some cases, when only stETH token functionality is required, it is reasonable to use stETH.sol interface, while in the other cases, usage of Lido.sol interface is needed. Since the approach is aligned with the rest of the protocol, we prefer to keep it unchanged.*

Go back to Findings Summary

I2: Typos

Impact:	Info	Likelihood:	N/A
Target:	ICSBondCurve.sol, CSModule.sol, SigningKeys.sol	Type:	Code quality

Description

The following typos were found in the codebase:

- `intervals` misspelled as `internals`;

Listing 24. Excerpt from [ICSBondCurve]()

```
interface ICSBondCurve {
    /// @dev Bond curve structure.
    ///
    /// It contains:
    ///  - internals    |> intervals-based representation of the bond curve
```

- `Invariant` misspelled as `Invariat`;

Listing 25. Excerpt from [CSModule]()

```
// @dev Invariat sum(no.totalExitedKeys for no in nos) ==
 _totalExitedValidators.
_totalExitedValidators =
    (_totalExitedValidators - totalExitedKeys) +
    uint64(exitedValidatorsCount);
```

- `keys` misspelled as `kes`;

Listing 26. Excerpt from [SigningKeys]()

```
/// @param pubkeys preallocated kes buffer to read in
/// @param signatures preallocated signatures buffer to read in
/// @param bufOffset start offset in `pubkeys`/`signatures` buffer to place
    values (in number of keys)
```

```
168 function loadKeysSigs(
```

Recommendation

Fix the typos in the codebase.

Fix 1.1

The typos were fixed.

[Go back to Findings Summary]

I3: Inconsistent `unchecked` use in `CSBondCurve`

Impact:	Info	Likelihood:	N/A
Target:	CSBondCurve.sol	Type:	Code quality

Description

Some out-of-bounds checks for input `curveId` parameters in `CSBondCurve.sol` are wrapped in `unchecked` blocks (`_setBondCurve`), while others are not (`_updateBondCurve`, `_getCurveInfo`).

Checks not wrapped in `unchecked` blocks:

Listing 27. Excerpt from CSBondCurve

```
function _getCurveInfo(
    uint256 curveId
) private view returns (BondCurve storage) {
    CSBondCurveStorage storage $ = _getCSBondCurveStorage();
    if (curveId > $.bondCurves.length - 1) {
        revert InvalidBondCurveId();
    }

    return $.bondCurves[curveId];
}
```

Listing 28. Excerpt from CSBondCurve

```
function _updateBondCurve(
    uint256 curveId,
    BondCurveIntervalInput[] calldata intervals
) internal {
    CSBondCurveStorage storage $ = _getCSBondCurveStorage();
    if (curveId > $.bondCurves.length - 1) {
        revert InvalidBondCurveId();
    }

    _checkBondCurve(intervals);

    delete $.bondCurves[curveId];
```

```

    _addIntervalsToBondCurve($.bondCurves[curveId], intervals);

    emit BondCurveUpdated(curveId, intervals);
}
```

Checks wrapped in `unchecked` blocks:

Listing 29. Excerpt from [CSBondCurve]()

```
function _setBondCurve(uint256 nodeOperatorId, uint256 curveId) internal {
    CSBondCurveStorage storage $ = _getCSBondCurveStorage();
    unchecked {
        if (curveId > $.bondCurves.length - 1) {
            revert InvalidBondCurveId();
        }
    }
    $.operatorBondCurveId[nodeOperatorId] = curveId;
    emit BondCurveSet(nodeOperatorId, curveId);
}
```

Recommendation

Maintain consistency in the codebase by either wrapping all out-of-bounds checks in `unchecked` blocks or removing all `unchecked` blocks for these checks.

Fix 1.1

All respective code segments were wrapped in `unchecked` blocks.

[Go back to Findings Summary]()

I4: Unused code

Impact:	Info	Likelihood:	N/A
Target:	ICSBondCurve.sol, PausableUntil.sol	Type:	Unused code

Description

The codebase contains multiple occurrences of unused code. See Appendix B for more details.

Recommendation

Consider removing the unused code to improve readability and maintainability of the codebase.

Partial solution 1.1

The unused error `ICSBondCurve.InvalidBondCurveMaxLength` was removed.

The unused modifier `PausableUntil.whenPaused` and function `PausableUntil._pauseUntil` were intentionally kept in the codebase to maintain consistency and allow for potential future use.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, Lido: CSM v2, 22.9.2025.

Appendix B: Wake Findings

This section lists the outputs from the Wake framework used for testing and static analysis during the audit.

B.1. Fuzzing

The following table lists all implemented execution flows in the Wake fuzzing framework.

ID	Flow	Added
F1	Creation of new node operators with all supported bond token types both through a vetted gate and a permissionless gate	1.0
F2	Addition of new validator keys to existing node operators with all supported token types	1.0
F3	Deposit of additional tokens into node operator bonds	1.0
F4	Removal of validator keys from node operators	1.0
F5	Update of vetted node operator addresses	1.0
F6	Claiming of vetted bond curve from existing node operators	1.0
F7	Reporting of execution layer rewards stealing	1.0
F8	Cancellation of execution layer rewards stealing reports	1.0
F9	Settlement of execution layer rewards stealing reports	1.0
F10	Compensation of stolen execution layer rewards	1.0
F11	Cleaning of node operator deposit queue	1.0

ID	Flow	Added
F12	Migration of existing node operator queue slots to higher priority slots	1.0
F13	Obtaining of validator keys to be deposited by Staking Router	1.0
F14	Addition of new consensus members responsible for rewards distribution report voting	1.0
F15	Removal of consensus members	1.0
F16	Submission and voting on rewards distribution and node operator strikes reports	1.0
F17	Pulling of rewards by node operators	1.0
F18	Claiming of rewards by node operators with all supported token types	1.0
F19	Update of target validator limits and modes	1.0
F20	Update of exited validator count	1.0
F21	Decrease of vetted signing key count	1.0
F22	Unsafe change of exited validator count	1.0
F23	Processing of historical validator withdrawal proofs	1.0
F24	Processing of non-historical validator withdrawal proofs	1.0
F25	Addition of new bond curves	1.0
F26	Update of existing bond curves	1.0
F27	Change of node operator bond curve	1.0
F28	Start of referral seasons	1.0
F29	Claiming of referrer bond curve	1.0
F30	End of referral seasons	1.0

ID	Flow	Added
F31	Setting of rebate recipient	1.0
F32	Processing of bad performance proofs	1.0
F33	Voluntary ejection of validator keys by node operator by starting index and length	1.0
F34	Voluntary ejection of validator keys by node operator by list of indices	1.0
F35	Reporting of validator exit delays	1.0
F36	Updating of node validator depositable keys count	1.0
F37	Setting of default key removal charge	1.0
F38	Setting of curve-specific key removal charge	1.0
F39	Unsetting of curve-specific key removal charge	1.0
F40	Setting of default execution rewards stealing additional fine	1.0
F41	Setting of curve-specific execution rewards stealing additional fine	1.0
F42	Unsetting of curve-specific execution rewards stealing additional fine	1.0
F43	Setting of default node operator keys limit	1.0
F44	Setting of curve-specific node operator keys limit	1.0
F45	Unsetting of curve-specific node operator keys limit	1.0
F46	Setting of default node operator striking parameters	1.0
F47	Setting of curve-specific node operator striking parameters	1.0
F48	Unsetting of curve-specific node operator striking parameters	1.0

ID	Flow	Added
F49	Setting of default validator bad performance penalty	1.0
F50	Setting of curve-specific validator bad performance penalty	1.0
F51	Unsetting of curve-specific validator bad performance penalty	1.0
F52	Setting of default priority queue configuration	1.0
F53	Setting of curve-specific priority queue configuration	1.0
F54	Unsetting of curve-specific priority queue configuration	1.0
F55	Setting of default allowed validator exit delay	1.0
F56	Setting of curve-specific allowed validator exit delay	1.0
F57	Unsetting of curve-specific allowed validator exit delay	1.0
F58	Setting of default validator exit delay penalty	1.0
F59	Setting of curve-specific validator exit delay penalty	1.0
F60	Unsetting of curve-specific validator exit delay penalty	1.0
F61	Setting of default maximum withdrawal request fee	1.0
F62	Setting of curve-specific maximum withdrawal request fee	1.0
F63	Unsetting of curve-specific maximum withdrawal request fee	1.0

Table 4. Wake fuzzing flows

The following table lists the invariants checked after each flow.

ID	Invariant	Added	Status
IV1	Transactions do not revert except where explicitly expected and with the expected data	1.0	**Fail** (H2, L1, W8)
IV2	Contracts emit expected events with correct parameters only when expected	1.0	**Fail** (W10)
IV3	The difference between the expected amount of tokens needed for new validator keys and the actual amount is within 10 wei	1.0	Success
IV4	`cleanDepositQueue` returns the correct number of removed items and last removal position	1.0	Success
IV5	`obtainDepositData` returns the correct deposit data	1.0	Success
IV6	`CSFeeOracle` returns the correct consensus report information	1.0	Success
IV7	The difference between the expected amount of rewards pulled by node operator and the actual amount is within 11 stETH shares	1.0	Success
IV8	All claimed unstETH, stETH and wstETH balances match expected values	1.0	Success
IV9	Newly created bond curves are assigned the correct ID	1.0	Success
IV10	`isValidatorExitDelayPenaltyApplicable` return value correctly reflects the behavior of `reportValidatorExitDelay` with the same parameters	1.0	Success

ID	Invariant	Added	Status
IV11	All important account native ETH balances match expected values	1.0	Success
IV12	All important account stETH share balances match expected values	1.0	Success
IV13	Bond information (including locked bonds) matches expected values	1.0	Success
IV14	Rewards distribution data history is stored correctly	1.0	Success
IV15	Node operator signing keys and signatures stored in the `CSModule` contract match expected values	1.0	Success
IV16	Numbers of node operator added keys, withdrawn keys, deposited keys, exited keys, vetted keys, enqueued keys, unbonded keys match expected values	1.0	Success
IV17	Normalized node operator depositable key counts match expected values	1.0	Success
IV18	Node operator target validator limits and modes match expected values	1.0	Success
IV19	Node operator rewards addresses and manager addresses match expected values	1.0	Success
IV20	Node operator queue migration flag is tracked correctly	1.0	Success
IV21	Node operator claimable bond shares match expected values	1.0	Success
IV22	Node operator claimable bond shares with rewards pulled match expected values	1.0	Success

ID	Invariant	Added	Status
IV23	Value reported by `getClaimableRewardsAndBondShares` reflects the actual amount of stETH shares being claimed by `claimRewardsStETH` with the same parameters	1.0	Success
IV24	Node operator withdrawal and exit penalty status is correct	1.0	Success
IV25	Module nonce is incrementing correctly	1.0	Success
IV26	Total accounting bond shares are equal to the sum of bond shares of all node operators	1.0	Success
IV27	Node operators priority queue follows the expected structure	1.0	**Fail** (L3)

Table 5. Wake fuzzing invariants

B.2. Detectors

```
wake detect unused-error

[INFO][HIGH] Unused error [unused-error]
    event BondCurveSet(uint256 indexed nodeOperatorId, uint256 curveId);

    error InvalidBondCurveLength();
❯   error InvalidBondCurveMaxLength();
    error InvalidBondCurveValues();
    error InvalidBondCurveId();
    error InvalidInitializationCurveId();
src/interfaces/ICSBondCurve.sol
```

Figure 1. Unused error

```
wake detect unused-modifier

[INFO][HIGH] Unused modifier [unused-modifier]
    error PauseUntilMustBeInFuture();

    /// @notice Reverts when resumed
    modifier whenPaused() {
        _checkPaused();
        _;
    }
src/lib/utils/PausableUntil.sol
```

Figure 2. Unused modifier

```
wake detect unused-function

[INFO][HIGH] Unused function [unused-function]
        _setPausedState(resumeSince);
    }

    function _pauseUntil(uint256 pauseUntilInclusive) internal {
        _checkResumed();
        if (pauseUntilInclusive < block.timestamp) {
            revert PauseUntilMustBeInFuture();
src/lib/utils/PausableUntil.sol
```

Figure 3. Unused function

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz