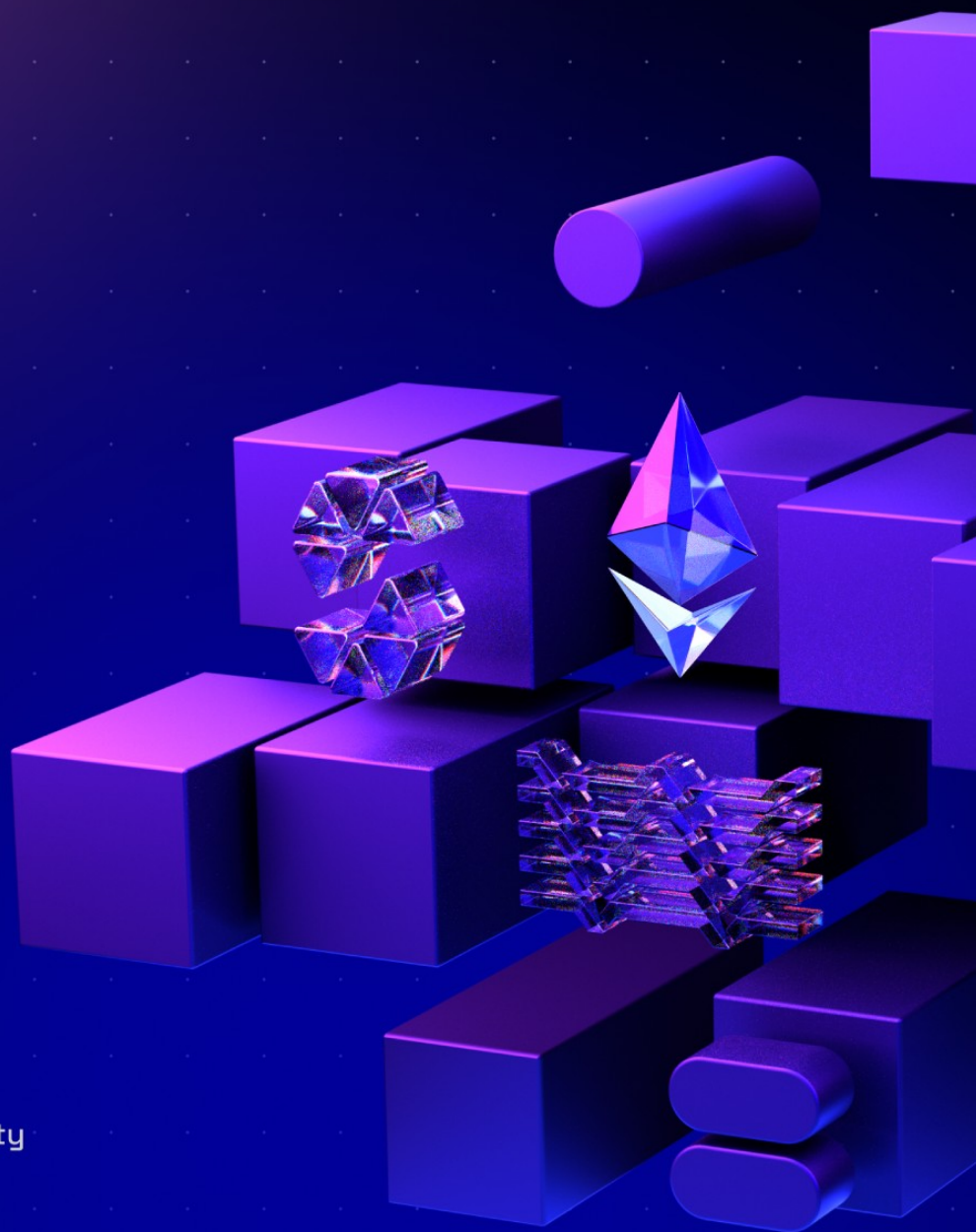
ackee
blockchain security

Lido

Community Staking Module

8.4.2025



Ackee Blockchain Security

Contents

1. Document Revisions

1.0-draft	Draft Report	06.09.2024
1.0	Final Report	12.09.2024
2.0-draft	Draft Report	04.10.2024
2.0	Final Report	12.10.2024
2.1	Deployment Verification	14.10.2024
3.0-draft	Draft Report	30.01.2025
3.0	Final Report	08.04.2025

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
Dmytro Khimchenko	Auditor
Lukáš Rajnoha	Auditor
Štěpán Šonský	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 Community Staking Module (CSM) is a permissionless module allowing community stakers to operate Ethereum validators with lower entry costs. Stakers provide stETH bonds, serving as security collateral, and receive rewards in the form of bond rebase and staking rewards (including execution layer rewards), which are socialized across Lido's staking modules.

Revision 1.0

Lido engaged Ackee Blockchain Security to perform a security review of the Lido protocol with a total time donation of 74 engineering days in a period between July 16 and September 6, 2024, with Michal Převrátil as the lead auditor.

The audit was performed on the commits `8ce944`[1] and `13f78f`[2] in the community-staking-module and easy-track repositories, respectively.

The scope of the audit included:

- all files in `src` directory, excluding `src/interfaces`, in the community-staking-module repository,
- `contracts/EVMScriptFactories/CSMSettleELStealingPenalty.sol` in the easy-track repository.

Community Staking Module heavily relies on off-chain components, most notably deposit key validation service and execution layer rewards stealing detection, that are critical for the security model of the module and were not in scope of this audit. The off-chain components were considered to work as expected according to the provided documentation.

We began our review by implementing and executing manually-guided differential stateful fuzz tests in Wake testing framework to verify the

correctness of the system. More details on the fuzzing process can be found in Report revision 1.0. Fuzzing of the contracts yielded findings M1, L3, L4, L6, L7 and L8. Then, we performed a thorough manual review of the code, focusing on the following aspects:

- bond supplied by node operators is correctly accounted and cannot be stolen by any account,
- the system contains no underflow/overflow issues that could lead to invalid state updates,
- permissionless functions cannot be abused to bring node operators into unintended states and prevent them from performing actions on the contracts,
- node operators are unable to deposit invalid keys and break the system functionality,
- CSM cannot cause denial of service to other staking modules and to the logic responsible for depositing keys,
- node operators are able to withdraw their rewards and unbonded funds without any issues,
- it is not possible to forge proofs of validator slashing or withdrawal that are not true but accepted by the smart contracts,
- the system cannot suffer from denial of service attacks by node operators spamming transactions,
- access controls are properly enforced in all critical functions without being overly restrictive or loose,
- node operators are unable to gain more rewards than stated in the rewards distribution report,
- rewards distribution report cannot be accepted if there is not enough votes to reach the configured quorum,

- node operators cannot bring validators into unexpected states, breaking the module's logic.

The manual review was performed in parallel and in sync with the Staking Router audit performed by Ackee Blockchain Security. All issues of possibly medium severity or higher were immediately reported to the Lido team. These issues include the report date in their descriptions in this document. The review was concluded using static analysis tools, including Wake.

Two issues were discovered by the Lido team during the audit, with L7 being one of them, later also discovered by Ackee Blockchain Security using fuzzing. The second issue poses a possibility to set the zero address as a node operator reward address.

Our review resulted in 39 findings, ranging from Info to Medium severity. The most severe one M1 results in valid keys covered bond not being deposited, incorrectly preventing node operators from creating new validators under certain circumstanes.

Ackee Blockchain Security recommends Lido:

- ensure the off-chain services are working as expected to achieve the security guarantees of the system,
- be cautious when using unsafe functions and functions that do not properly update all internal state, such as `CSAccounting.updateBondCurve` or `CSAccounting.setBondCurve`,
- avoid using unchecked blocks that heavily rely on correctness of external contracts, where breaking the assumptions may lead to critical vulnerabilities,
- ensure the contracts are deployed and initialized atomically so no front-running of initialization functions is possible.

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

Revision 2.0

Lido engaged Ackee Blockchain Security to perform an incremental review of fixes for previously reported findings in Community Staking Module with a total time donation of 3 engineering days in a period between September 30 and October 2, 2024, with Michal Převrátil as the lead auditor.

The audit was performed on the commit `347496`[3] in the community-staking-module repository, with the scope being all changes to the files in the `src` directory since the previous revision.

The review began with updating the fuzz tests from the previous revision to ensure the properties of the codebase were preserved. A manual review followed, focusing on the integration of the fixes into the existing codebase.

25 findings were fixed, and the remaining 12 were acknowledged. No new findings were discovered.

See Report Revision 2.0 for the description of changes in this revision and updates to the fuzz tests.

Revision 2.1

Lido engaged Ackee Blockchain Security to perform deployment verification of Community Staking Module on the Ethereum mainnet. The verification was performed on the same commit as in the previous revision, `347496`[4].

The verification concluded successfully with an exact bytecode match achieved for all scoped contracts at the following addresses on the Ethereum mainnet:

- AssetRecovererLib: 0xa74528edc289b1a597Faf83fCfF7eFf871Cc01D9
- NOAddresses: 0xF8E5de8bAf8Ad7C93DCB61D13d00eb3D57131C72

- QueueLib: 0xD19B40Cb5401f1413D014A56529f03b3452f70f9
- CSModule: 0x8daea53b17a629918cdfab785c5c74077c1d895b
- OssifiableProxy (for CSModule): 0xdA7dE2ECdDfccC6c3AF10108Db212ACBBf9EA83F
- CSAccounting: 0x71FCD2a6F38B644641B0F46c345Ea03Daabf2758
- OssifiableProxy (for CSAccounting): 0x4d72BFF1BeaC69925F8Bd12526a39BAAb069e5Da
- CSFeeOracle: 0x919ac5C6c62B6ef7B05cF05070080525a7B0381E
- OssifiableProxy (for CSFeeOracle): 0x4D4074628678Bd302921c20573EEa1ed38DdF7FB
- CSFeeDistributor: 0x17Fc610ecbbAc3f99751b3B2aAc1bA2b22E444f0
- OssifiableProxy (for CSFeeDistributor): 0xD99CC66fEC647E68294C6477B40fC7E0F6F618D0
- CSVerifier: 0x3Dfc50f22aCA652a0a6F28a0F892ab62074b5583
- CSEarlyAdoption: 0x3D5148ad93e2ae5DedD1f7A8B3C19E7F67F90c0E
- HashConsensus: 0x71093efF8D8599b5fA340D665Ad60fA7C80688e4
- CSMSettleElStealingPenalty: 0xF6B6E7997338C48Ea3a8BCfa4BB64a315fDa76f4

The deployment verification script is available at https://github.com/Ackee-Blockchain/tests-lido-csm.

The deployed contracts were tested through forking with fuzz tests prepared in the revision 1.0, with an exception of the `CSVerifier` contract, which would result in fuzzing undesirably slow. No issues were detected during the fuzz testing.

Revision 3.0

Lido engaged Ackee Blockchain Security to perform a review of removal of the validator slashing reporting functionality with a total time donation of 2 engineering days in a period between January 28 and January 30, 2025, with Michal Převrátil as the lead auditor.

The audit was performed on the commit `346991`[5] in the community-staking-module repository, with the scope being the changes made to the `CSVerifier.sol` file since the previous revision.

The audit began with a manual review of the `CSVerifier.sol` file and relevant parts of the `CSModule.sol` file.

The review focused on ensuring:

- the removal of slashing reporting functionality does not affect other functionalities of the contracts,
- the accounting of validator slashing remains correct,
- there are no weak spots in the contract update process and no additional risks are introduced with the removal of slashing reporting functionality.

Manually guided fuzz tests prepared in the previous revisions were updated and successfully run.

The security review concluded with no new findings discovered.

Additionally, deployment verification of the `CSVerifier` contract at address 0x0c345dFa318f9F4977cdd4f33d80F9D0ffA38e8B confirmed that the contract was deployed from the audited code with the correct Pectra hardfork parameters. The verification concluded successfully with an exact bytecode match and parameters matching the expected values defined in EIP-7600 and Lido DAO proposal[6].

The deployment verification script is available at https://github.com/Ackee-Blockchain/tests-lido-csm/tree/revision-3.0.

Ackee Blockchain Security recommends Lido:

- add comments to the `CSModule.submitInitialSlashing` function explaining that the function is no longer expected to be called,
- ensure the old `CSVerifier` contract is disabled upon deployment of the new `CSVerifier` contract.

See Report Revision 3.0 for the description of changes in this revision and updates to the fuzz tests.

[1] full commit hash: `8ce9441dce1001c93d75d065f051013ad5908976`

[2] full commit hash: `13f78f1ec44436abfb1b2a55f640e2178f79d029`

[3] full commit hash: `347496df916c3b987a7f3fe8b0bd85c9b62ad730`

[4] full commit hash: `347496df916c3b987a7f3fe8b0bd85c9b62ad730`

[5] full commit hash: `3469910c0d29a54b37d0c4de3cf527a3e7be2099`

[6] https://research.lido.fi/t/lip-27-ensuring-compatibility-with-ethereum-s-pectra-upgrade/9444

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	0	1	8	16	14	39

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
M1: Non-optimistic vetting & unbonded keys bad accounting	Medium	1.0	Fixed
L1: Check for `fastLane` member can be bypassed while consensus disabled	Low	1.0	Acknowledged
L2: Discard logic of report may never be used	Low	1.0	Acknowledged
L3: Single leaf rewards pulling	Low	1.0	Acknowledged

Finding title	Severity	Reported	Status
L4: Execution layer stealing settlement revert	Low	1.0	Fixed
L5: Strict `msg.value` check	Low	1.0	Fixed
L6: Execution layer rewards stealing settlement not resetting bond curve	Low	1.0	Fixed
L7: Bad `targetLimit` accounting in `getNodeOperatorSummary`	Low	1.0	Fixed
L8: Depositable keys count not updated in `normalizeQueue`	Low	1.0	Fixed
W1: `Versioned` contracts can be initialized to zero version	Warning	1.0	Fixed
W2: Inconsistencies in setting consensus version in `BaseOracle`	Warning	1.0	Partially fixed
W3: `targetLimitMode` loose check in `updateTargetValidatorsLimits`	Warning	1.0	Fixed
W4: Inconsistent `_onlyRecoverer` function implementation	Warning	1.0	Fixed
W5: Fixed withdrawal credentials in `CSVerifier`	Warning	1.0	Fixed
W6: Dirty memory bytes in `Validator` SSZ serialization	Warning	1.0	Fixed

Finding title	Severity	Reported	Status
W7: Permissionless queue clearing	Warning	1.0	Acknowledged
W8: Missing `curveId` check	Warning	1.0	Fixed
W9: Node operator may withdraw before execution layer stealing reported	Warning	1.0	Acknowledged
W10: `processOracleReport` check prevents fixing mistakes	Warning	1.0	Acknowledged
W11: `targetLimitMode` set to `0` not clearing `targetLimit`	Warning	1.0	Fixed
W12: Permissionless unpausable functions	Warning	1.0	Acknowledged
W13: Unchecked blocks	Warning	1.0	Fixed
W14: EIP-7002 mandatory for CSM	Warning	1.0	Acknowledged
W15: Event inconsistencies	Warning	1.0	Fixed
W16: `depositable > enqueued` blocking Staking Router	Warning	1.0	Fixed
I1: `HashConsensus` condition never met	Info	1.0	Acknowledged
I2: type `GIndex` should have `pow()` function available	Info	1.0	Fixed

Finding title	Severity	Reported	Status
I3: `CSBondCore._claimStETH` function unnecessarily calls the `_ethByShares` function when emitting event	Info	1.0	Fixed
I4: `AssetRecoverer` does not allow specifying the amount in the `recoverEther` function	Info	1.0	Acknowledged
I5: Interfaces outside of dedicated folder	Info	1.0	Fixed
I6: `HashConsensus` should inherit from `IConsensusContract`	Info	1.0	Fixed
I7: Redundant `whenPaused` check	Info	1.0	Fixed
I8: `pullFeeRewards` does not update depositable keys count	Info	1.0	Acknowledged
I9: `CSBondCore._getClaimableBondShares` should be unimplemented	Info	1.0	Acknowledged
I10: State variable read multiple times	Info	1.0	Fixed
I11: Inconsistent higher bits clearing in `QueueLib`	Info	1.0	Fixed
I12: `QueueLib.clean` return last index of cleared item	Info	1.0	Fixed

Finding title	Severity	Reported	Status
I13: Unused code	Info	1.0	Partially fixed
I14: Incorrect documentation & typos	Info	1.0	Fixed

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Michal Převrátil	Lead Auditor
Dmytro Khimchenko	Auditor
Lukáš Rajnoha	Auditor
Štěpán Šonský	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

Community Staking Module (CSM) is a first permissionless staking module in Lido. 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 always internally converted to stETH, so node operators receive rewards in form of stETH rebases. The required bond amount is non-linear with the number of validator keys, so the more keys an operator has, the lower is the required bond per key.

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 of 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. After the module launch, only operators selected for early adoption are allowed to join with limited number of validator keys. After the early adoption end, any operator can join the module. 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 setteled 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.

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.

Validator slashing and withdrawals may be reported permissionlessly using EIP-4788 proofs.

Until EIP-7002 is merged, Lido is having to trust node operators not to be

intentionally penalized, lowering the validator's balance provided by Lido. Node operator's bond does not fully cover all possible losses that may incur from validator penalties. Additionaly, it is not possible for Lido to enforce a validator withdrawal to return the deposit provided by Lido. However, Lido is allowed to mark a given validator as stuck, preventing other validator keys from being deposited by the same node operator and even stopping distribution of its rewards.

After the EIP-7002 is merged, Lido will be able to perform a full withdrawal of malicious and under-performing validators, further reducing the risk.

CSM is required to increment the module nonce on validator key changes to prevent deposits of invalid keys. Nonce changes are needed to require additional bond increases or to charge penalties to prevent denial of service attacks.

Fuzzing

A manually-guided differential stateful fuzz test was developed during the review to test the correctness and robustness of the system. The fuzz test employs fork testing technique to test the system with external contracts exactly as they are deployed in the deployment environment. This is crucial to detect any potential integration issues.

The differential fuzz test keeps its 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 Staking Router fuzz test prepared by Ackee Blockchain Security during a parallel audit to ensure compatibility and integration of the two systems.

Additional fuzz tests were prepared by Ackee Blockchain Security to isolate and focus on certain parts of the codebase. These tests covered:

- `HashConsensus` contract with reports voting, changing membership and configuring parameters,
- `QueueLib` helper library with adding, removing, cleaning and normalizing queue entries,
- various helper functions used for EIP-4788 proofs verification, including `GIndex` helpers, `SSZ.verifyProof` and `uint256.toLittleEndian`.

The fuzz tests simulate the whole system and make strict assertions about the behavior of the contracts. The most severe findings M1, L3, L4, L6, L7 and L8 were 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.

Findings

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

M1: Non-optimistic vetting & unbonded keys bad accounting

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	CSModule.sol	Type:	Logic error
Reported on:		August 6, 2024	

Description

Under normal circumstances, the `CSModule` contract performs optimistic vetting of validator keys. I.e., if there is no invalid deposit key and signature waiting to be processed, new validator keys are also considered valid (vetted).

In the case of non-optimistic vetting, the depositable keys count must be adjusted to reflect the number of vetted keys. At the same time, the depositable keys count may be influenced by unbonded keys, i.e., keys that are not covered by the deposited bond.

The following code is responsible for computing the new depositable keys count based on vetted and unbonded keys:

Listing 1. Excerpt from CSModule._updateDepositableValidatorsCount

```
1739 uint256 newCount = no.totalVettedKeys - no.totalDepositedKeys;
1740
1741 uint256 unbondedKeys = accounting.getUnbondedKeysCount(nodeOperatorId);
1742 if (unbondedKeys > newCount) {
1743     newCount = 0;
1744 } else {
1745     unchecked {
1746         newCount -= unbondedKeys;
1747     }
1748 }
```

However, unbonded keys computed with `CSAccounting.getUnbondedKeysCount` cannot be directly compared to `no.totalVettedKeys`, because the value of unbonded keys is computed from `no.totalAddedKeys`.

Listing 2. Excerpt from CSAccounting._getUnbondedKeysCount

```
610 uint256 nonWithdrawnKeys = CSM.getNodeOperatorNonWithdrawnKeys(
611     nodeOperatorId
612 );
```

Listing 3. Excerpt from CSModule

```
1392 function getNodeOperatorNonWithdrawnKeys(
1393     uint256 nodeOperatorId
1394 ) external view returns (uint256) {
1395     NodeOperator storage no = _nodeOperators[nodeOperatorId];
1396     unchecked {
1397         return no.totalAddedKeys - no.totalWithdrawnKeys;
1398     }
1399 }
```

The issue was discovered with fuzzing using the Wake testing framework. See Appendix B for more information on the fuzzing campaign performed during the audit.

Exploit scenario

A node operator has 5 deposited keys, 2 keys waiting to be processed, 6 vetted keys and 1 unbonded key. Out of 2 keys waiting to be processed, 1 key is invalid and the other is valid. However, due to the wrong accounting, the final count of depositable keys is 0.

Recommendation

Fix the accounting of unbonded keys in the case of non-optimistic vetting.

One possible implementation of the fix in the

`CSModule._updateDepositableValidatorsCount` function could be structured as follows:

```
uint256 newCount = no.totalVettedKeys - no.totalDepositedKeys;

uint256 unbondedKeys = accounting.getUnbondedKeysCount(nodeOperatorId);
if (unbondedKeys > no.totalAddedKeys - no.totalDepositedKeys) {
    newCount = 0;
} else {
    unchecked {
        newCount = Math.min(
            newCount,
            no.totalAddedKeys - no.totalDepositedKeys - unbondedKeys
        );
    }
}
```

Fix 2.0

The modified logic now correctly accounts for unbonded keys in the case of non-optimistic vetting.

Listing 4. Excerpt from CSModule._updateDepositableValidatorsCount

```
1736 uint256 newCount = no.totalVettedKeys - no.totalDepositedKeys;
1737 uint256 unbondedKeys = accounting.getUnbondedKeysCount(nodeOperatorId);
1738
1739 {
1740     uint256 nonDeposited = no.totalAddedKeys - no.totalDepositedKeys;
1741     if (unbondedKeys >= nonDeposited) {
1742         newCount = 0;
1743     } else if (unbondedKeys > no.totalAddedKeys - no.totalVettedKeys) {
1744         newCount = nonDeposited - unbondedKeys;
1745     }
1746 }
```

Go back to Findings Summary

L1: Check for `fastLane` member can be bypassed while consensus disabled

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	HashConsensus.sol	Type:	Logic error

Description

The `HashConsensus.sol` contract has the logic of a fast lane, which lets specific hash consensus members vote for the report earlier than any consensus member who is not in the fast lane. However, after a quorum is disabled or a quorum needs votes of all members for the report, every consensus member is approached as if it was in the fast lane.

Listing 5. Excerpt from HashConsensus/_submitReport

```
1016 if (
1017     currentSlot <= frame.refSlot + config.fastLaneLengthSlots &&
1018     !_isFastLaneMember(memberIndex, frame.index)
1019 ) {
```

Listing 6. Excerpt from HashConsensus

```
928 function _isFastLaneMember(
929     uint256 index,
930     uint256 frameIndex
931 ) internal view returns (bool) {
932     uint256 totalMembers = _memberStates.length;
933     (uint256 flLeft, uint256 flPastRight) = _getFastLaneSubset(
934         frameIndex,
935         totalMembers
936     );
```

Exploit scenario

Fast lane members chosen by the `HashConsensus` contract are voting. In one moment, quorum is disabled. After that, every consensus member submits its vote to `HashConsensus`. After the quorum is enabled, all votes are accepted as valid votes without considering that they were not in the fast lane before. It occurs because the quorum value is connected to validating if the consensus member is in the fast lane or not.

Listing 7. Excerpt from HashConsensus/_getFastLaneSubset

```
911 function _getFastLaneSubset(
912     uint256 frameIndex,
913     uint256 totalMembers
914 ) internal view returns (uint256 startIndex, uint256 pastEndIndex) {
915     uint256 quorum = _quorum;
916     if (quorum >= totalMembers) {
917         startIndex = 0;
918         pastEndIndex = totalMembers;
919     } else {
920         startIndex = frameIndex % totalMembers;
921         pastEndIndex = startIndex + quorum;
922     }
```

Recommendation

Check if the consensus member is in the fast lane without considering the current quorum's value.

Acknowledgment 2.0

The finding was acknowledged by Lido with the following comment:

> `fastLane` *feature was introduced in the Lido V2 release to ensure that all Oracle members are active by assigning a new batch of members to each report to report first. This feature was never meant to be restrictive. Hence, the issue mentioned*

> *is not an issue, given the initial purpose of the* `fastLane`.
>
> — Lido

Go back to Findings Summary

L2: Discard logic of report may never be used

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	HashConsensus.sol, BaseOracle.sol	Type:	Logic error

Description

The project introduces logic of discarding a report that has already been chosen by consensus members but has lost the needed number of votes by some means. Using the logic, consensus members can re-elect correct reports. A typical call trace of reaching the consensus is provided below:

Listing 8. Excerpt from HashConsensus._submitReport

```
1087 if (support >= _quorum) {
1088     _consensusReached(frame, report, varIndex, support);
1089 } else if (prevConsensusLost) {
1090     _consensusNotReached(frame);
1091 }
```

Listing 9. Excerpt from HashConsensus

```
1094 function _consensusReached(
1095     ConsensusFrame memory frame,
1096     bytes32 report,
1097     uint256 variantIndex,
1098     uint256 support
1099 ) internal {
1100     if (
1101         _reportingState.lastConsensusRefSlot != frame.refSlot ||
1102         _reportingState.lastConsensusVariantIndex != variantIndex
1103     ) {
1104         _reportingState.lastConsensusRefSlot = uint64(frame.refSlot);
1105         _reportingState.lastConsensusVariantIndex = uint64(variantIndex);
1106         emit ConsensusReached(frame.refSlot, report, support);
1107         _submitReportForProcessing(frame, report);
```

```
1108     }
1109 }
```

Listing 10. Excerpt from HashConsensus

```
1253 function _submitReportForProcessing(
1254     ConsensusFrame memory frame,
1255     bytes32 report
1256 ) internal {
1257     IReportAsyncProcessor(_reportProcessor).submitConsensusReport(
1258         report,
1259         frame.refSlot,
1260         _computeTimestampAtSlot(frame.reportProcessingDeadlineSlot)
1261     );
1262 }
```

Listing 11. Excerpt from BaseOracle.submitConsensusReport

```
252 _storageConsensusReport().value = report;
```

After all validations are met, the report value is stored in the `ConsensusReport` struct.

However, consensus members can immediately send the chosen report for processing, preventing the use of discard logic. This can be achieved by calling the `submitReportData` function in the `CSFeeOracle` contract. This function can be called by ANY consensus member from the `HashConsensus` contract to start processing the report.

Listing 12. Excerpt from CSFeeOracle

```
122 function submitReportData(
123     ReportData calldata data,
124     uint256 contractVersion
125 ) external whenResumed {
126     _checkMsgSenderIsAllowedToSubmitData();
127     _checkContractVersion(contractVersion);
128     _checkConsensusData(
129         data.refSlot,
```

```
130         data.consensusVersion,
131         // it's a waste of gas to copy the whole calldata into mem but seems
    there's no way around
132         keccak256(abi.encode(data))
133     );
134     _startProcessing();
135     _handleConsensusReportData(data);
136 }
```

During this processing, the value of the ref slot is stored as `LAST_PROCESSING_REF_SLOT_POSITION`.

Listing 13. Excerpt from BaseOracle._startProcessing

```
394 _checkProcessingDeadline(report.processingDeadlineTime);
395
396 uint256 prevProcessingRefSlot = LAST_PROCESSING_REF_SLOT_POSITION
397     .getStorageUint256();
398 if (prevProcessingRefSlot == report.refSlot) {
399     revert RefSlotAlreadyProcessing();
400 }
401
402 LAST_PROCESSING_REF_SLOT_POSITION.setStorageUint256(report.refSlot);
```

While the deadline for submitting the report does not end, consensus members might vote for other reports. However, if the report has already been processed, it cannot be discarded because `LAST_PROCESSING_REF_SLOT_POSITION` is set.

The flow of submitting the new report and discarding the previous one involves the following steps:

Listing 14. Excerpt from HashConsensus._submitReport

```
1023 if (slot <= _getLastProcessingRefSlot()) {
```

Listing 15. Excerpt from HashConsensus

```
1248 function _getLastProcessingRefSlot() internal view returns (uint256) {
```

```
1249     return
1250         IReportAsyncProcessor(_reportProcessor).getLastProcessingRefSlot();
1251 }
```

Listing 16. Excerpt from BaseOracle

```
297 /// @notice Returns the last reference slot for which processing of the
    report was started.
298 ///
299 function getLastProcessingRefSlot() external view returns (uint256) {
300     return LAST_PROCESSING_REF_SLOT_POSITION.getStorageUint256();
301 }
```

Exploit scenario

After submitting a report and noticing that consensus is reached, a hash consensus member calls the `submitReportData` function to start processing the previously submitted report, avoiding the chance that anyone discards the provided report.

Recommendation

There is no straightforward solution to this problem as fixing it requires design changes. One possible remediation is to disallow the report to be processed for a given period of time (while ensuring the report will be processed eventually before the deadline) after the report is accepted by the consensus members.

Acknowledgment 2.0

The finding was acknowledged by Lido with the following comment:

> *The discard feature was introduced for re-voting for the alternative report in case the existing one can not make it through the sanity checks (might happen due to a software bug in the oracle software). The described issue refers to the*

> *other use case that was never considered during development. Even though the issue is valid for the case described, it does not affect the intended functionality of the re-voting feature.*
>
> — Lido

Go back to Findings Summary

L3: Single leaf rewards pulling

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	CSAccounting.sol	Type:	Data validation

Description

Node operators may claim their rewards and excessive bond using one of the `claimRewardsStETH`, `claimRewardsWstETH`, `claimRewardsUnstETH` functions in the `CSAccounting` contract. As an optional step, it is also possible to pull the rewards from the `CSFeeDistributor` contract by providing a non-empty `rewardsProof` merkle proof.

However, the condition checking if the rewards proof is empty prevents pulling rewards in the case of a single leaf merkle tree.

Listing 17. Excerpt from CSAccounting.claimRewardsStETH

```
287 if (rewardsProof.length != 0) {
288     _pullFeeRewards(nodeOperatorId, cumulativeFeeShares, rewardsProof);
289 }
290 CSBondCore._claimStETH(nodeOperatorId, stETHAmount, rewardAddress);
```

The issue was discovered with fuzzing using the Wake testing framework. See Appendix B for more information on the fuzzing campaign performed during the audit.

Exploit scenario

Due to the logic in the `claimRewards*` functions, it is not possible to pull rewards in the same function call in the case when the merkle tree contains only one leaf. The proof is empty in single leaf merkle trees, thus the pull logic is not executed.

Recommendation

Ensure that the rewards tree will always have more than one leaf or that users are informed that they need to pull rewards using a dedicated `pullFeeRewards` function in the opposite case.

Alternatively, add an additional boolean parameter to the `claimRewards*` functions which will indicate that the rewards should be pulled from the `CSFeeDistributor` contract.

Note that the same issue is also present when creating a new node operator and submitting an early adoption merkle proof. However, the case with a single leaf is not expected to happen in practice as the merkle tree will be generated only once with more than one address (leaf).

Acknowledgment 2.0

The finding was acknowledged by Lido with the following comment:

> *The team discovered this issue even before the audits started, but it was never mentioned in the on-chain codebase. The solution here is to add a 'stone' (dummy leaf) into the rewards distribution tree to ensure at least two leaves in the tree. This feature is already implemented in the off-chain Oracle code. Note added to the code base in f5b5361.*
>
> — Lido

Go back to Findings Summary

L4: Execution layer stealing settlement revert

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	CSBondCore.sol	Type:	Denial of service
Reported on:		July 22, 2024	

Description

Node operators are responsible for setting up validator hardware and software. According to the Lido's Community Staking Module policy, node operators are required to configure execution layer and MEV rewards recipient address to the Lido's rewards vault.

Off-chain implementation is expected to observe the behavior of validators and report the case when execution layer rewards are sent to a different address than Lido's rewards vault. The reporting is performed using the `CSModule.reportELRewardsStealingPenalty` function. When a report is submitted, a corresponding part of the bond is locked. The penalty then can be cancelled by Lido, compensated by the node operator or settled by Lido.

In the latter case, the `CSModule.settleELRewardsStealingPenalty` function is called with the list of all node operator IDs to settle the penalty. The internal logic loops over all node operator IDs and calls the `CSAccounting.settleLockedBondETH` function.

Listing 18. Excerpt from CSModule.settleELRewardsStealingPenalty

```
1062 for (uint256 i; i < nodeOperatorIds.length; ++i) {
1063     uint256 nodeOperatorId = nodeOperatorIds[i];
1064     _onlyExistingNodeOperator(nodeOperatorId);
1065     uint256 settled = accounting.settleLockedBondETH(nodeOperatorId);
```

The `CSAccounting.settleLockedBondETH` function then burns the locked part of

the bond (if any) and removes the lock.

Listing 19. Excerpt from CSAccounting

```
375 function settleLockedBondETH(
376     uint256 nodeOperatorId
377 ) external onlyCSM returns (uint256 settledAmount) {
378     uint256 lockedAmount = CSBondLock.getActualLockedBond(nodeOperatorId);
379     if (lockedAmount > 0) {
380         settledAmount = CSBondCore._burn(nodeOperatorId, lockedAmount);
381     }
382     // reduce all locked bond even if bond isn't covered lock fully
383     CSBondLock._remove(nodeOperatorId);
384 }
```

The burning is implemented as a request to the Lido's Burner contract to burn `min(locked_shares, available_shares)` shares.

Listing 20. Excerpt from CSBondCore

```
240 function _burn(
241     uint256 nodeOperatorId,
242     uint256 amount
243 ) internal returns (uint256 burned) {
244     uint256 toBurnShares = _sharesByEth(amount);
245     uint256 burnedShares = _reduceBond(nodeOperatorId, toBurnShares);
246     IBurner(LIDO_LOCATOR.burner()).requestBurnShares(
247         address(this),
248         burnedShares
249     );
```

However, it is not checked that `min(locked_shares, available_shares)`, represented by `burnedShares` in the code snippet, is non-zero.

The issue was discovered with fuzzing using the Wake testing framework. See Appendix B for more information on the fuzzing campaign performed during the audit.

Exploit scenario

Lido calls the `CSModule.settleELRewardsStealingPenalty` function with a list of node operator IDs to settle the penalties. One of the node operators has the zero amount of bond shares, which results in `IBurner.requestBurnShares` being called with zero shares as the input parameter.

Calling the `IBurner.requestBurnShares` function with zero shares reverts the execution with the `ZeroBurnAmount` error. The `CSModule.settleELRewardsStealingPenalty` function then must be called again without the node operator ID causing the execution to revert.

Recommendation

Do not call the `IBurner.requestBurnShares` function if the amount of shares to burn is zero.

Fix 2.0

Fixed by performing an early return if the amount of shares to burn is zero.

Listing 21. Excerpt from CSBondCore

```
234 function _burn(uint256 nodeOperatorId, uint256 amount) internal {
235     uint256 toBurnShares = _sharesByEth(amount);
236     uint256 burnedShares = _reduceBond(nodeOperatorId, toBurnShares);
237     // If no bond already
238     if (burnedShares == 0) return;
```

Go back to Findings Summary

L5: Strict `msg.value` check

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	CSModule.sol	Type:	Denial of service
Reported on:		July 18, 2024	

Description

The `CSModule` contract allows deposits of new validators keys through 3 different methods using ETH, stETH and wstETH as bond (security collateral). The functions accepting ETH require strict `msg.value` amount to succeed.

Listing 22. Excerpt from CSModule.addNodeOperatorETH

```
270 if (
271     msg.value !=
272     accounting.getBondAmountByKeysCount(
273         keysCount,
274         accounting.getBondCurve(nodeOperatorId)
275     )
276 ) {
277     revert InvalidAmount();
278 }
```

Listing 23. Excerpt from CSModule.addValidatorKeysETH

```
392 if (
393     msg.value !=
394     accounting.getRequiredBondForNextKeys(nodeOperatorId, keysCount)
395 ) {
396     revert InvalidAmount();
397 }
```

Exploit scenario

A node operator is expected to call `CSAccounting.getBondAmountByKeysCount`

and `CSAccounting.getRequiredBondForNextKeys` functions to estimate the amount of ETH needed for the keys deposit.

However, in between the call and the actual deposit transaction, the required amount of ETH might change. The reasons for this could be:

- malicious actor increasing the node operator's bond with a small amount using one of the `depositETH`, `depositStETH` or `depositWstETH` functions in the `CSModule` contract,
- change of conversion rates between ETH and stETH.

As a consequence, the deposit transaction will fail as the `msg.value` sent will not match the deposit amount.

Recommendation

Allow higher amounts of ETH in the `addNodeOperatorETH` and `addValidatorKeysETH` functions than the required minimum.

Fix 2.0

Fixed by accepting even higher amounts of ETH than the required minimum in both `addNodeOperatorETH` and `addValidatorKeysETH` functions.

Listing 24. Excerpt from CSModule.addNodeOperatorETH

```
275 if (
276     msg.value < accounting.getBondAmountByKeysCount(keysCount, curveId)
277 ) {
278     revert InvalidAmount();
279 }
```

Listing 25. Excerpt from CSModule.addValidatorKeysETH

```
392 if (
393     msg.value <
394     accounting.getRequiredBondForNextKeys(nodeOperatorId, keysCount)
```

```
395 ) {
396     revert InvalidAmount();
397 }
```

Go back to Findings Summary

L6: Execution layer rewards stealing settlement not resetting bond curve

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	CSModule.sol	Type:	Logic error
Reported on:		July 23, 2024	

Description

Node operators are responsible for setting up validator hardware and software. According to the Lido's Community Staking Module policy, node operators are required to configure execution layer and MEV rewards recipient address to the Lido's rewards vault.

Off-chain implementation is expected to observe the behavior of validators and report the case when execution layer rewards are sent to a different address than Lido's rewards vault. After the report is submitted, an adequate part of node operator's bond is locked, and Lido is allowed to settle the penalty by calling the `CSModule.settleELRewardsStealingPenalty` function.

Listing 26. Excerpt from CSModule.settleELRewardsStealingPenalty

```
1062 for (uint256 i; i < nodeOperatorIds.length; ++i) {
1063     uint256 nodeOperatorId = nodeOperatorIds[i];
1064     _onlyExistingNodeOperator(nodeOperatorId);
1065     uint256 settled = accounting.settleLockedBondETH(nodeOperatorId);
1066     if (settled > 0) {
1067         // Bond curve should be reset to default in case of confirmed MEV
     stealing. See https://hackmd.io/@lido/SygBLW5ja
1068         accounting.resetBondCurve(nodeOperatorId);
1069         // Nonce should be updated if depositableValidators change
1070         // No need to normalize queue due to only decrease in depositable
     possible
1071         _updateDepositableValidatorsCount({
1072             nodeOperatorId: nodeOperatorId,
```

```
1073             incrementNonceIfUpdated: true,
1074             normalizeQueueIfUpdated: false
1075         });
1076     }
1077     emit ELRewardsStealingPenaltySettled(nodeOperatorId);
1078 }
```

The logic assumes that `setteled` will only be zero when no bond is locked. However, this is not true in the case when there is non-zero reported stealing but the node operator has the zero bond so there is nothing to settle.

Exploit scenario

A node operator is picked for the early adoption program with a more beneficial bond curve. Later, execution layer rewards stealing is reported for the node operator. Additionally, the bond of node operator reaches zero due to a full withdrawal of a penalized validator. When performing the settlement of the reported stealing, the beneficial early adoption bond curve is mistakenly not reset to the default one.

Recommendation

Change the logic to distinguish between the states when there is no stealing reported for a node operator and when there is a stealing reported but the node operator has no bond to settle the penalty. Reset the bond curve when there is non-zero reported stealing even if the node operator has the zero bond.

Fix 2.0

Fixed by checking the locked bond amount before performing the settlement and resetting the bond curve if the locked bond was non-zero.

Listing 27. Excerpt from CSModule.settleELRewardsStealingPenalty

```
1064 uint256 lockedBondBefore = _accounting.getActualLockedBond(
```

```
1065     nodeOperatorId
1066 );
1067
1068 _accounting.settleLockedBondETH(nodeOperatorId);
1069
1070 // settled amount might be zero either if the lock expired, or the bond is
     zero
1071 // so we need to check actual locked bond before to determine if the
     penalty was settled
1072 if (lockedBondBefore > 0) {
1073     // Bond curve should be reset to default in case of confirmed MEV
     stealing. See https://hackmd.io/@lido/SygBLW5ja
1074     _accounting.resetBondCurve(nodeOperatorId);
```

Go back to Findings Summary

L7: Bad `targetLimit` accounting in `getNodeOperatorSummary`

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	CSModule.sol	Type:	Logic error

Description

The `CSModule.getNodeOperatorSummary` function is called by Lido's Staking Router to get the node operator summary information.

A part of the information returned by the function is the `targetLimit` value along with `targetLimitMode`, specifying if the node operator has the maximum validator limit set and with what mode it is enforced. The following modes are possible:

- `0`: target limit disabled,
- `1`: soft target limit with smooth exit mode,
- `2`: hard target limit with boosted exit mode.

The following logic is responsible for calculating the `targetLimit` and `targetLimitMode` values:

Listing 28. Excerpt from CSModule.getNodeOperatorSummary

```
1436 uint256 totalUnbondedKeys = accounting.getUnbondedKeysCountToEject(
1437     nodeOperatorId
1438 );
1439 // Force mode enabled and unbonded
1440 if (
1441     totalUnbondedKeys > 0 &&
1442     no.targetLimitMode == FORCED_TARGET_LIMIT_MODE_ID
1443 ) {
1444     targetLimitMode = FORCED_TARGET_LIMIT_MODE_ID;
1445     unchecked {
```

```
1446         targetValidatorsCount = Math.min(
1447             no.targetLimit,
1448             no.totalAddedKeys -
1449                 no.totalWithdrawnKeys -
1450                 totalUnbondedKeys
1451         );
1452     }
1453     // No force mode enabled but unbonded
1454 } else if (totalUnbondedKeys > 0) {
1455     targetLimitMode = FORCED_TARGET_LIMIT_MODE_ID;
1456     unchecked {
1457         targetValidatorsCount =
1458             no.totalAddedKeys -
1459             no.totalWithdrawnKeys -
1460             totalUnbondedKeys;
1461     }
1462 } else {
1463     targetLimitMode = no.targetLimitMode;
1464     targetValidatorsCount = no.targetLimit;
1465 }
```

The logic prefers hard limit mode `2` based on the unbonded keys count over a possibly higher soft limit mode `1`.

The issue was discovered internally by Lido and through fuzzing by Ackee Blockchain Security using the Wake testing framework. See Appendix B for more information on the fuzzing campaign performed during the audit.

Exploit scenario

Due to the incorrect logic, a node operator may bypass a higher soft limit targeting already deposited validator keys over a lower hard limit based on the unbonded keys count that only targets validator keys that are not yet deposited.

Recommendation

Ensure the logic takes into account unbonded keys only if some of the deposited validator keys are unbonded.

Fix 2.0

Fixed by always enforcing the hard limit mode 2 if there are unbonded deposited validator keys.

Listing 29. Excerpt from CSModule.getNodeOperatorSummary

```
1433 uint256 totalUnbondedKeys = accounting.getUnbondedKeysCountToEject(
1434     nodeOperatorId
1435 );
1436 uint256 totalNonDepositedKeys = no.totalAddedKeys -
1437     no.totalDepositedKeys;
1438 // Force mode enabled and unbonded deposited keys
1439 if (
1440     totalUnbondedKeys > totalNonDepositedKeys &&
1441     no.targetLimitMode == FORCED_TARGET_LIMIT_MODE_ID
1442 ) {
1443     targetLimitMode = FORCED_TARGET_LIMIT_MODE_ID;
1444     unchecked {
1445         targetValidatorsCount = Math.min(
1446             no.targetLimit,
1447             no.totalAddedKeys -
1448                 no.totalWithdrawnKeys -
1449                 totalUnbondedKeys
1450         );
1451     }
1452     // No force mode enabled but unbonded deposited keys
1453 } else if (totalUnbondedKeys > totalNonDepositedKeys) {
1454     targetLimitMode = FORCED_TARGET_LIMIT_MODE_ID;
1455     unchecked {
1456         targetValidatorsCount =
1457             no.totalAddedKeys -
1458             no.totalWithdrawnKeys -
1459             totalUnbondedKeys;
1460     }
```

Go back to Findings Summary

L8: Depositable keys count not updated in `normalizeQueue`

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	CSModule.sol	Type:	Logic error
Reported on:		July 25, 2024	

Description

The `CSModule.normalizeQueue` function may be called to re-enter the Community Staking Module queue in the case when a given node operator has the number of depositable keys greater than the number of keys in the queue.

Listing 30. Excerpt from CSModule

```
999 function normalizeQueue(uint256 nodeOperatorId) external {
1000     _onlyNodeOperatorManager(nodeOperatorId);
1001     depositQueue.normalize(_nodeOperators, nodeOperatorId);
1002 }
```

Under certain conditions, the depositable keys count value stored in the `CSModule` contract may not be up-to-date and may need to be recalculated. The function `CSModule.normalizeQueue` should be responsible for updating the depositable keys count before re-entering the queue, but this functionality is missing.

The issue was discovered with fuzzing using the Wake testing framework. See Appendix B for more information on the fuzzing campaign performed during the audit.

Exploit scenario

A part of node operator's bond is locked due to reported execution layer rewards stealing. As a consequence, the depositable keys count for the node operator is decreased.

The bond lock retention period ends and the bond is unlocked. However, the node operator's depositable keys count is not updated. The node operator then has to call one of the state-changing functions of the `CSModule` contract that were designed for a different purpose but update the depositable keys count as a side effect.

Recommendation

Update the depositable keys count in the `CSModule.normalizeQueue` function before re-entering (normalizing) the queue.

Fix 2.0

Fixed by explicitly calling the `CSModule._updateDepositableValidatorsCount` function in the `CSModule.normalizeQueue` function.

Listing 31. Excerpt from CSModule.normalizeQueue

```
997 _updateDepositableValidatorsCount({
998     nodeOperatorId: nodeOperatorId,
999     incrementNonceIfUpdated: true
1000 });
```

Go back to Findings Summary

W1: `Versioned` contracts can be initialized to zero version

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

Description

The `Versioned` contract handles versioning of inheriting contracts.

Contracts that inherit from the `Versioned` contract are expected to be initialized to a given version during their initialization phase by calling the `_initializeContractVersionTo` function.

Listing 32. Excerpt from Versioned

```
38 function _initializeContractVersionTo(uint256 version) internal {
39     if (getContractVersion() != 0) revert NonZeroContractVersionOnInit();
40     _setContractVersion(version);
41 }
```

As the documentation states, a zero version should only be allowed before initialization.

Listing 33. Excerpt from Versioned

```
7  contract Versioned {
8      using UnstructuredStorage for bytes32;
9
10     /// @dev Storage slot: uint256 version
11     /// Version of the initialized contract storage.
12     /// The version stored in CONTRACT_VERSION_POSITION equals to:
13     /// - 0 right after the deployment, before an initializer is invoked (and
   only at that moment);
14     /// - N after calling initialize(), where N is the initially deployed
   contract version;
15     /// - N after upgrading contract by calling finalizeUpgrade_vN().
```

However, the `_initializeContractVersionTo` function allows the version to be set to zero. Acidentally setting the version to zero would pose a great risk since contracts inheriting the `Versioned` contract do not need to rely on Open Zepellin's `Initializable` contract for securing the initialize function. Instead, they can rely on the `Versioned` contract, which fails when trying to initialize the contract version again after it has been already set (i.e., when the version is not zero).

In the audit scope, the `BaseOracle` contract inherits from the `Versioned` contract. Thus, if the `CSFeeOracle` (which inherits the `BaseOracle` contract) is initialized to a zero version by accident, it could be re-initialized by anyone.

Recommendation

Consider disallowing the `Versioned` contract to be set to a zero version once it has been initialized.

Fix 2.0

Fixed by reverting the execution of the `_initializeContractVersionTo` function if the version is zero.

Go back to Findings Summary

W2: Inconsistencies in setting consensus version in `BaseOracle`

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

Description

When the `BaseOracle` contract's `initialize` function is called, it sets the consensus version to a value passed as an argument via the `_setConsensusVersion` function.

Listing 34. Excerpt from BaseOracle

```
435 function _setConsensusVersion(uint256 version) internal {
436     uint256 prevVersion = CONSENSUS_VERSION_POSITION.getStorageUint256();
437     if (version == prevVersion) revert VersionCannotBeSame();
438     CONSENSUS_VERSION_POSITION.setStorageUint256(version);
439     emit ConsensusVersionSet(version, prevVersion);
440 }
```

When trying to set the consensus version to zero during the initialization, the `_setConsensusVersion` will revert with the `VersionCannotBeSame` error (as the `prevVersion` will be 0 in this case). Thus, only non-zero values can be set as the consensus version during the initialization.

Only afterwards, when the consensus version is already set to a non-zero value, is it allowed to set it to zero via the `setConsensusVersion` function. This poses an inconsistency in what values are allowed to be set as the consensus version during and outside of the initialization phase.

Additionally, variables that save arbitrary version data are generally required across the codebase to be greater than their previous value when a new value is being set (e.g., the contract version from the `Versioned` contract). Such restriction is not applied in the case of the consensus version, though, which

can be set to any value, including a lower one.

Recommendation

Consider restricting the consensus version to only allow setting values greater than the previous value, unless there is a specific reason to allow the values to decrease.

Partial solution 2.0

It is no longer possible to set the consensus version to zero.

The ability to set the consensus version to a lower value was kept in the `BaseOracle` contract with the following comment:

> *Consensus version is a flag for off-chain tooling, indicating what code to execute to reach consensus. It can be moved back and forth, and off-chain tooling will determine supported combinations of an oracle contract's and a consensus' versions. The only version value restricted to use is* `0`.
>
> — Lido

Go back to Findings Summary

W3: `targetLimitMode` loose check in `updateTargetValidatorsLimits`

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

Description

In the `CSModule` contract, the `updateTargetValidatorsLimits` function accepts a `targetLimitMode` parameter to specify a target limit mode to be set for a given Node Operator. The `targetLimitMode` parameter is expected to be one of the target limit modes defined in the documentation:

- `0`: target limit disabled,
- `1`: soft target limit with smooth exit mode,
- `2`: hard target limit with boosted exit mode.

When validating the `targetLimitMode` parameter, the function throws when the `targetLimitMode` value is greater than `uint8.max`, thus allowing a greater value than the current maximum target limit mode value (currently 2).

This can result in unexpected behavior if the `targetLimitMode` is set to an unexpected target limit mode value.

Recommendation

Although it is common practice to allow additional values for parameters to support possible future changes, in this case, this should not be necessary as the `CSModule` is expected to be upgradeable, thus making it possible to update the `updateTargetValidatorsLimits` function in the future. Only allow setting the `targetLimitMode` value to modes defined in the documentation instead of checking against the `uint8.max` value.

Fix 2.0

The target limit mode being set is now validated not to be greater than `FORCED_TARGET_LIMIT_MODE_ID` (currently 2).

Go back to Findings Summary

W4: Inconsistent `_onlyRecoverer` function implementation

Impact:	Warning	Likelihood:	N/A
Target:	CSFeeOracle.sol	Type:	Code quality

Description

The `AssetRecoverer` contract contains functions for recovering assets from the inheriting contracts. Access control for these functions is implemented via the `onlyRecoverer` modifier, which is expected to be overridden by the inheriting contracts.

There is an inconsistency in the implementation of the `_onlyRecoverer` function across contracts that inherit from the `AssetRecoverer` contract. Most contracts override the function as in the following example from the `CSAccounting` contract:

Listing 35. Excerpt from CSAccounting

```
638 function _onlyRecoverer() internal view override {
639     _checkRole(RECOVERER_ROLE);
640 }
```

However, the `CSFeeOracle` contract implements it as:

Listing 36. Excerpt from CSFeeOracle

```
207 function _onlyRecoverer() internal view override {
208     _checkRole(RECOVERER_ROLE, msg.sender);
209 }
```

If no second argument is passed to the `_checkRole` function, it uses the `_msgSender` function to retrieve the message sender. The `_msgSender` function returns `msg.sender` for regular transactions, but for meta transactions it can

be used to return the end user, rather than the relayer.

Recommendation

Although this inconsistency should not pose any risk in the current codebase, consider implementing the `_onlyRecoverer` function consistently across all contracts that inherit from `AssetRecoverer`.

Fix 2.0

The `CSFeeOracle._onlyRecoverer` function is now implemented consistently with the other contracts that inherit from the `AssetRecoverer` contract.

Go back to Findings Summary

W5: Fixed withdrawal credentials in `CSVerifier`

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

Description

The `CSVerifier` contract is responsible for verification of permissionless validator withdrawal and slashing proofs against the EIP-4788 beacon block root.

During the validator withdrawal verification, the `CSVerifier` contract checks the withdrawal credentials of the validator against the current address of Lido's withdrawal vault.

Listing 37. Excerpt from CSVerifier._processWithdrawalProof

```
306 // WC to address
307 address withdrawalAddress = address(
308     uint160(uint256(witness.withdrawalCredentials))
309 );
310 if (withdrawalAddress != LOCATOR.withdrawalVault()) {
311     revert InvalidWithdrawalAddress();
312 }
```

Although correct, the check will prevent reporting validator withdrawals in the case of the withdrawal vault address change.

Recommendation

Either ensure that a new EIP-4788 verifier can be deployed and attached to the `CSModule` contract timely, or remove the check.

Note that the check is redundant as validator keys must pass comprehensive off-chain verification before they are deposited and the public key in the proof of the withdrawn validator is checked against the public key saved in

the `CSModule` contract.

Fix 2.0

The `CSVerifier` contract now stores the withdrawal vault address which is independent of changes to the LidoLocator contract.

Listing 38. Excerpt from CSVerifier._processWithdrawalProof

```
308 if (withdrawalAddress != WITHDRAWAL_ADDRESS) {
309     revert InvalidWithdrawalAddress();
310 }
```

Go back to Findings Summary

W6: Dirty memory bytes in `Validator` SSZ serialization

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

Description

The `SSZ` library is used for SSZ serialization of beacon chain objects.

The `Validator` struct contains a dynamic `bytes pubkey` field that must be serialized separately. The public key is always 48 bytes, zero-padded from the right to 64 bytes and hashed with SHA-256 to produce the public key root.

Listing 39. Excerpt from SSZ.hashTreeRoot

```
87 bytes32 pubkeyRoot;
88
89 assembly {
90     // Dynamic data types such as bytes are stored at the specified offset.
91     let offset := mload(validator)
92     // Call sha256 precompile with the pubkey pointer
93     let result := staticcall(
94         gas(),
95         0x02,
96         add(offset, 32),
97         0x40,
98         0x00,
99         0x20
100     )
101
102     if iszero(result) {
103         // Precompiles returns no data on OutOfGas error.
104         revert(0, 0)
105     }
106
107     pubkeyRoot := mload(0x00)
108 }
```

The code listing assumes that the 16 bytes after the 48 bytes long public key in memory will always be zeroed out. However, this is not guaranteed by the Solidity compiler and may lead to subtle bugs.

Recommendation

Copy the public key into the memory scratch space and zero-out the 16 bytes after it.

Fix 2.0

Resolved by implementing the recommendation.

Go back to Findings Summary

W7: Permissionless queue clearing

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

Description

The `QueueLib.clean` function is responsible for removing queue items for node operators that have more keys enqueued than is the current amount of depositable keys. The function may be called by anyone through the `CSModule.cleanDepositQueue` function.

Exploit scenario

The clean queue functionality may be exploited by malicious actors waiting for node operators to have their depositable key counts decreased.

The depositable count may be decreased due to penalizations applied to node operator's bond, but also due to (even unjustified) reporting of execution layer rewards stealing or due to the node operator not reacting promptly to validator exit requests.

Losing positions in the queue for a given node operator results in having to re-enter the queue again once the depositable count increases and having to wait for a longer time.

Recommendation

Consider making the `CSModule.cleanDepositQueue` function privileged to prevent other users abusing temporary fluctuations in depositable counts.

Acknowledgment 2.0

The finding was acknowledged by Lido with the following comment:

> *The fact that keys might still be present in the deposit queue once they are no longer depositable results from a technical limitation. Simply speaking, there is no feasible way to delete batches with non-depositable keys from the queue, given the unpredictable size of the queue. Hence, the fact mentioned in the finding is acceptable according to the design and should not be fixed.*
>
> — Lido

Go back to Findings Summary

W8: Missing `curveId` check

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

Description

The `CSBondCurve.getBondAmountByKeysCount` and `CSBondCurve.getKeysCountByBondAmount` functions return the amount of bond needed to cover the given number of keys and vice versa.

Listing 40. Excerpt from CSBondCurve

```
96 function getBondAmountByKeysCount(
97     uint256 keys,
98     uint256 curveId
99 ) public view returns (uint256) {
100     return getBondAmountByKeysCount(keys, getCurveInfo(curveId));
101 }
```

Listing 41. Excerpt from CSBondCurve

```
107 function getKeysCountByBondAmount(
108     uint256 amount,
109     uint256 curveId
110 ) public view returns (uint256) {
111     return getKeysCountByBondAmount(amount, getCurveInfo(curveId));
112 }
```

Both functions accept `curveId` as a parameter, but it is not being checked if the provided `curveId` is valid. Due to the internal logic of both functions, the execution reverts with the `INDEX_ACCESS_OUT_OF_BOUNDS` error. The error does not provide any information about the invalid `curveId` parameter, which may cause confusion.

Recommendation

Revert the execution with the `InvalidBondCurveId` error if an invalid `curveId` value is supplied to provide a clear error message to users.

Fix 2.0

The function `getCurveInfo` now reverts with the `InvalidBondCurveId` error if an invalid `curveId` value is supplied. The error is propagated to both `getBondAmountByKeysCount` and `getKeysCountByBondAmount` functions.

Go back to Findings Summary

W9: Node operator may withdraw before execution layer stealing reported

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

Description

Node operators are responsible for setting up validator hardware and software. According to the Lido's Community Staking Module policy, node operators are required to configure execution layer and MEV rewards recipient address to the Lido's rewards vault.

Off-chain implementation is expected to observe the behavior of validators and report the case when execution layer rewards are sent to a different address than Lido's rewards vault. The reporting is performed using the `CSModule.reportELRewardsStealingPenalty` function.

However, if execution layer rewards stealing is not reported timely, node operators may exit their validators and withdraw all of their funds so that Lido is not able to settle the stolen rewards. This is due to the fact the validator bond is unlocked when the withdrawal of the validator is permissionlessly reported and not when the validator is marked as exited, which can only be done by Lido.

Recommendation

Either ensure is not possible to withdraw all validators before the execution layer rewards stealing is reported or unlock the bond when the validator is marked as exited instead of when marked as withdrawn.

Acknowledgment 2.0

The finding was acknowledged by Lido with the following comment:

> *The mechanism of reporting exited validators lies outside the control of CSM and is used for distinct purposes within the core Lido protocol. At the same time, asking for additional confirmation when reporting the validator's withdrawal will not be ideal from the UX perspective. It is proposed to report the facts of EL rewards stealing promptly and acknowledge that delays in the reporting might result in the bond withdrawals before the penalty reporting.*
>
> — Lido

Go back to Findings Summary

W10: `processOracleReport` check prevents fixing mistakes

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

Description

The function `processOracleReport` in the `CSFeeDistributor` contract is expected to be called by the `CSFeeOracle` contract once consensus is reached on rewards distribution.

The function changes the rewards distribution merkle tree root and CID only if new reward shares are distributed.

Listing 42. Excerpt from CSFeeDistributor.processOracleReport

```
120 if (
121     totalClaimableShares + distributed > STETH.sharesOf(address(this))
122 ) {
123     revert InvalidShares();
124 }
125
126 if (distributed > 0) {
127     if (bytes(_treeCid).length == 0) revert InvalidTreeCID();
128     if (_treeRoot == bytes32(0)) revert InvalidTreeRoot();
129     if (_treeRoot == treeRoot) revert InvalidTreeRoot();
130
131     // Doesn't overflow because of the very first check.
132     unchecked {
133         totalClaimableShares += distributed;
134     }
135
136     treeRoot = _treeRoot;
137     treeCid = _treeCid;
138
139     emit DistributionDataUpdated(
140         totalClaimableShares,
141         _treeRoot,
142         _treeCid
```

```
143     );
144 }
```

Although the non-zero `distributed` check is correct, it prevents fixing the rewards distribution merkle tree in case of a mistake and no new rewards are distributed.

Recommendation

Consider removing the `distributed > 0` check to support the scenario when it is necessary to fix rewards distribution in case of a mistake.

Acknowledgment 2.0

The finding was acknowledged by Lido with the following comment:

> *It is assumed that once settled, a report can not be changed. In case of a severe mistake in the report, several operational actions will be required (changing the report frame and delivering a new report). Since the operations mentioned can not be done immediately, by the time the fixed report arrives, there will be some new shares to be distributed, or some shares can be transferred to the* `CSFeeDistributor` *manually. Allowing for new* `treeRoot` *and* `treeCid` *with no new distributed shares makes contracts more vulnerable to errors in the off-chain code. Hence, the code was left as is.*
>
> — Lido

Go back to Findings Summary

W11: `targetLimitMode` set to `0` not clearing `targetLimit`

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

Description

`CSModule.updateTargetValidatorsLimits` is a privileged function called by the Lido's Staking Router contract to update the target validator limits and modes for a given node operator.

3 different modes are supported:

- `0`: target limit disabled,
- `1`: soft target limit with smooth exit mode,
- `2`: hard target limit with boosted exit mode.

When the mode is set to `0`, the `targetLimit` parameter is stored in the `CSModule` contract as the value passed to the function. However, it is reasonable to always set `targetLimit` to `0` when the mode is set to `0`. The current behavior in `CSModule` is inconsitent with the Node Operators Registry module that always sets `targetLimit` to `0` when the mode is set to `0`.

Recommendation

Always set `targetLimit` to `0` when the `targetLimitMode` is being set to `0`.

Fix 2.0

Fixed by always clearing `targetLimit` when `targetLimitMode` is set to `0`.

Go back to Findings Summary

W12: Permissionless unpausable functions

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

Description

The codebase contains multiple functions callable by any user or node operator that are not pausable. The functions that are not pausable are:

- `CSModule.removeKeys`,
- `CSModule.normalizeQueue`,
- `CSModule.compensateELRewardsStealingPenalty`,
- `CSModule.submitWithdrawal` through `CSVerifier.processWithdrawalProof` and `CSVerifier.processHistoricalWithdrawalProof`,
- `CSModule.submitInitialSlashing` through `CSVerifier.processSlashingProof`,
- `CSModule.cleanDepositQueue`,
- `CSAccounting.pullFeeRewards`.

Although these functions do not allow claiming of funds from the protocol, any issue discovered in them may still have serious consequences.

Recommendation

Reconsider making the mentioned functions pausable.

Acknowledgment 2.0

The finding was acknowledged by Lido with the following comment:

> *The main goal of the pause function is to support expected*

> *operational scenarios like module sunset or temporary pause. There are some unpausable methods. However, none of them allows for the creation of new validators or claiming rewards. Hence, any call of the unpausable methods will not result in the token transfers outside the module contracts or the creation of new validators.*
>
> — Lido

Go back to Findings Summary

W13: Unchecked blocks

Impact:	Warning	Likelihood:	N/A
Target:	CSBondCore.sol	Type:	Overflow/Underflow

Description

The codebase contains several unchecked blocks to optimize the gas usage. Some of them assume certain behavior of external contracts to work correctly. Even though the current behavior of the external contracts hold the assumptions true, this cannot be taken for granted as the behavior of the external contracts might change in the future. Even off-by-one error may lead to critical vulnerabilities then.

1. Unwrapping wstETH

Listing 43. Excerpt from CSBondCore._depositWstETH

```
152 uint256 sharesBefore = LIDO.sharesOf(address(this));
153 WSTETH.unwrap(amount);
154 uint256 sharesAfter = LIDO.sharesOf(address(this));
155 unchecked {
156     _increaseBond(nodeOperatorId, sharesAfter - sharesBefore);
157 }
```

The code assumes that `WSTETH.unwrap` will never decrease the amount of stETH shares held by the current contract. Breaking the assumption would lead to a critical vulnerability.

2. Wrapping stETH

Listing 44. Excerpt from CSBondCore._claimWstETH

```
227 uint256 sharesBefore = LIDO.sharesOf(address(this));
228 uint256 amount = WSTETH.wrap(_ethByShares(sharesToClaim));
229 uint256 sharesAfter = LIDO.sharesOf(address(this));
230 unchecked {
```

```
231     _unsafeReduceBond(nodeOperatorId, sharesBefore - sharesAfter);
232 }
```

The `WSTETH.wrap` call is not checked to never increase the amount of stETH shares held by the current contract. Additionally, the change in the amount of stETH shares must not exceed the amount of shares held by the node operator, otherwise the `_unsafeReduceBond` call will lead to underflow.

3. Requesting withdrawal through unstETH

Listing 45. Excerpt from CSBondCore._claimUnstETH

```
177 uint256 claimableShares = _getClaimableBondShares(nodeOperatorId);
178 uint256 sharesToClaim = requestedAmountToClaim <
179     _ethByShares(claimableShares)
180     ? _sharesByEth(requestedAmountToClaim)
181     : claimableShares;
182 if (sharesToClaim == 0) revert NothingToClaim();
183
184 uint256[] memory amounts = new uint256[](1);
185 amounts[0] = _ethByShares(sharesToClaim);
186 uint256 sharesBefore = LIDO.sharesOf(address(this));
187 uint256[] memory requestIds = WITHDRAWAL_QUEUE.requestWithdrawals(
188     amounts,
189     to
190 );
191 uint256 sharesAfter = LIDO.sharesOf(address(this));
192 unchecked {
193     _unsafeReduceBond(nodeOperatorId, sharesBefore - sharesAfter);
194 }
```

The code expects unstETH withdrawal request not to increase the amount of stETH shares held by the current contract. Additionally, the withdrawn shares must never be higher than the current amount of shares held by the node operator performing the withdrawal. Otherwise, the `_unsafeReduceBond` call will lead to underflow.

Recommendation

It is recommended to remove the unchecked blocks in the described cases, including the one in `_unsafeReduceBond` to remove the dependency on the correct behavior of external contracts.

Fix 2.0

Unchecked blocks from all the aforementioned cases were removed, including the block in `_unsafeReduceBond`.

Go back to Findings Summary

W14: EIP-7002 mandatory for CSM

Impact:	Warning	Likelihood:	N/A
Target:	N/A	Type:	N/A

Description

EIP-7002 is mandatory for the correct and low-risk operation of the Community Staking Module. Without the EIP merged, the protocol is open to higher risk of malicious actors having validators penalized, causing losses to deposits made by Lido. Although node operators deposit bonds to cover the penalties, the bonds are significantly lower than the damages caused by the penalties in the worst-case scenario.

Lido is unable to exit maliciously behaving and under-performing validators, only to use the means provided by CSM to stop depositing new validator keys and distributing new rewards.

Recommendation

Ensure the risk is acceptable for the protocol or wait for EIP-7002 to be merged before the deployment.

Acknowledgment 2.0

The finding was acknowledged by Lido with the following comment:

> *The team is aware of the necessity of the EIP-7002. The fact that EIP-7002 would most likely not be live on the mainnet by the time of CSM release is reflected in the bond sizes proposed for the mainnet and limited module share (see "Stake Allocation strategy") at the early stages.*
>
> — Lido

Go back to Findings Summary

W15: Event inconsistencies

Impact:	Warning	Likelihood:	N/A
Target:	*.sol	Type:	Code quality

Description

The codebase contains multiple inconsistencies in emitted events.

1. `BondLockRemoved` and `ELRewardsStealingPenaltySettled` emitted even if nothing locked

The `CSBondLock.BondLockRemoved` and `CSModule.ELRewardsStealingPenaltySettled` events are emitted even if a given node operator has not locked any part of the bond when settling execution layer rewards stealing.

Listing 46. Excerpt from CSAccounting.settleLockedBondETH

```
378 uint256 lockedAmount = CSBondLock.getActualLockedBond(nodeOperatorId);
379 if (lockedAmount > 0) {
380     settledAmount = CSBondCore._burn(nodeOperatorId, lockedAmount);
381 }
382 // reduce all locked bond even if bond isn't covered lock fully
383 CSBondLock._remove(nodeOperatorId);
```

Listing 47. Excerpt from CSBondLock

```
139 function _remove(uint256 nodeOperatorId) internal {
140     delete _getCSBondLockStorage().bondLock[nodeOperatorId];
141     emit BondLockRemoved(nodeOperatorId);
142 }
```

Listing 48. Excerpt from CSModule.settleELRewardsStealingPenalty

```
1065 uint256 settled = accounting.settleLockedBondETH(nodeOperatorId);
1066 if (settled > 0) {
1067     // Bond curve should be reset to default in case of confirmed MEV
     stealing. See https://hackmd.io/@lido/SygBLW5ja
1068     accounting.resetBondCurve(nodeOperatorId);
```

```
1069     // Nonce should be updated if depositableValidators change
1070     // No need to normalize queue due to only decrease in depositable
     possible
1071     _updateDepositableValidatorsCount({
1072         nodeOperatorId: nodeOperatorId,
1073         incrementNonceIfUpdated: true,
1074         normalizeQueueIfUpdated: false
1075     });
1076 }
1077 emit ELRewardsStealingPenaltySettled(nodeOperatorId);
```

2. Two events in `CSAccounting` but no in `CSModule` on execution layer rewards stealing compensation

Two events are emitted in the `CSAccounting` contract when calling the `CSModule.compensateELRewardsStealingPenalty`, the first being either `CSBondLock.BondLockChanged` or `CSBondLock.BondLockRemoved` and the second being `CSAccounting.BondLockCompensated`. No event is emitted when calling `CSModule.settleELRewardsStealingPenalty`.

This is an inconsistency compared to other functions working with execution layer rewards stealing penalties, when one event is emitted in `CSAccounting` and one in `CSModule`.

Recommendation

Fix the event inconsistencies in the codebase. More specifically:

1. Emit the `CSBondLock.BondLockRemoved` and `CSModule.ELRewardsStealingPenaltySettled` events only if the node operator has locked a non-zero part of the bond.
2. Consider emitting the `BondLockCompensated` from the `CSModule` contract instead of the `CSAccounting` contract.

Fix 2.0

1. The `CSBondLock.BondLockRemoved` and `CSModule.ELRewardsStealingPenaltySettled` events are no longer emitted if

the node operator has not locked any part of the bond.

2. A new event `ELRewardsStealingPenaltyCompensated` was introduced to the `CSModule` contract to be emitted when calling the `CSModule.compensateELRewardsStealingPenalty` function.

Go back to Findings Summary

W16: `depositable > enqueued` blocking Staking Router

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

Description

Under edge case scenarios, the number of depositable keys of a given node operator may be greater than the number of enqueued keys in the Community Staking Module queue. The depositable keys count is reported to the Staking Router contract responsible for allocation of stakes. Due to the number of depositable keys being greater than the number of keys waiting in the queue, the logic responsible for depositing keys may revert the execution with the `NotEnoughKeys` error. This may temporarily block the process of depositing new keys from all modules in Lido.

Recommendation

Ensure that `NotEnoughKeys` errors caused by the depositable keys count being greater than the enqueued keys count cannot block the Staking Router contract from making deposits with keys from other modules. Expect the describe scenario may happen in the future and be ready to normalize depositable validator counts for affected node operators.

Fix 2.0

Queue normalization is now always performed when the depositable keys count is changed to prevent the described scenario from occurring.

Go back to Findings Summary

I1: `HashConsensus` condition never met

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

Description

The following condition in the code is never met:

Listing 49. Excerpt from HashConsensus._submitReport

```
1013 if (currentSlot > frame.reportProcessingDeadlineSlot)
1014     revert StaleReport();
```

`currentSlot` and `frame` are calculated in realtime. `frame` is calculated in this way:

Listing 50. Excerpt from HashConsensus

```
753 ConsensusFrame({
754     index: frameIndex,
755     refSlot: uint64(frameStartSlot - 1),
756     reportProcessingDeadlineSlot: uint64(
757         nextFrameStartSlot - 1 - DEADLINE_SLOT_OFFSET
758     )
759 });
```

As `DEADLINE_SLOT_OFFSET` is set to 0, the calculation of `frame.reportProcessingDeadlineSlot` can be considered as:

```
1 reportProcessingDeadlineSlot: uint64(
2     nextFrameStartSlot - 1
3 )
```

`nextFrameStartSlot` is always greater than `currentSlot`. As the result, the condition is never met.

Recommendation

Remove this condition for gas optimization.

Acknowledgment 2.0

The finding was acknowledged by Lido with the following comment:

> `DEADLINE_SLOT_OFFSET` *is reserved for future use, and can be changed.*
>
> — Lido

Go back to Findings Summary

I2: type `GIndex` should have `pow()` function available

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

Description

There is missing `pow()` function, which should be also allowed for `GIndex` type.

Listing 51. Excerpt from GIndex

```
using {
    isRoot,
    isParentOf,
    index,
    width,
    shr,
    shl,
    concat,
    unwrap
} for GIndex global;
```

Recommendation

Add the `pow` function to the using-for directive.

Fix 2.0

The `pow` function was included into the using-for directive.

Go back to Findings Summary

I3: `CSBondCore._claimStETH` function unnecessarily calls the `_ethByShares` function when emitting event

Impact:	Info	Likelihood:	N/A
Target:	CSBondCore.sol	Type:	Gas optimization

Description

The `CSBondCore` contract is an abstract contract used for the accounting of Node Operators bonds.

`CSBondCore._claimStETH` calculates claimed shares in ether when emitting the `BondClaimedStETH` event via the `_ethByShares` function. The amount of claimed shares is, however, already returned from the `LIDO.transferShares(to, sharesToClaim)` call. It is, thus, sufficient to catch this returned value and use it in the event emission instead of calling `_ethByShares` again.

Listing 52. Excerpt from CSBondCore

```
199 function _claimStETH(
200     uint256 nodeOperatorId,
201     uint256 requestedAmountToClaim,
202     address to
203 ) internal {
204     uint256 claimableShares = _getClaimableBondShares(nodeOperatorId);
205     uint256 sharesToClaim = requestedAmountToClaim <
206         _ethByShares(claimableShares)
207         ? _sharesByEth(requestedAmountToClaim)
208         : claimableShares;
209     if (sharesToClaim == 0) revert NothingToClaim();
210     _unsafeReduceBond(nodeOperatorId, sharesToClaim);
211
212     LIDO.transferShares(to, sharesToClaim);
213     emit BondClaimedStETH(nodeOperatorId, to, _ethByShares(sharesToClaim));
214 }
```

Recommendation

Consider removing the `_ethByShares` function call to calculate claimed shares in favor of using the value returned from the `LIDO.transferShares(to, sharesToClaim)` call to reduce gas consumption.

Fix 2.0

The return value from `LIDO.transferShares` is now directly used instead of the `_ethByShares` computation in the `_claimStETH` function and also in the `CSBondCore._charge` function.

Go back to Findings Summary

I4: `AssetRecoverer` does not allow specifying the amount in the `recoverEther` function

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

Description

The `AssetRecoverer` is an abstract contract that is used to recover ether, ERC20, ERC721 and ERC1155 assets from the account.

The `AssetRecoverer` contains an inconsistency where its `recoverERC20` function requires to specify the amount of tokens to be recovered from the account, while the `recoverEther` function does not and uses `address(this).balance` as the amount to be recovered instead.

Recommendation

Consider adding an `amount` parameter to the `recoverEther` function, specifying the amount of ether to be recovered from the account.

Acknowledgment 2.0

The finding was acknowledged by Lido with the following comment:

> *None of the CSM contracts assumes the presence of ETH on the balance during normal operation. The amount is introduced in the recoverERC20 methods due to possible peculiarities in the ERC20 token implementations (like the fee on transfer: https://github.com/lyrx/fot).*
>
> — Lido

Go back to Findings Summary

I5: Interfaces outside of dedicated folder

Impact:	Info	Likelihood:	N/A
Target:	HashConsensus.sol, BaseOracle.sol	Type:	Code quality

Description

Currently, important interfaces are defined within implementation files, which can lead to reduced code clarity and potential reusability issues. Specifically:

- `IReportAsyncProcessor` interface in `HashConsensus.sol`,
- `IConsensusContract` interface in `BaseOracle.sol`.

Recommendation

Extract these interfaces into separate files within the `interfaces` folder.

Fix 2.0

Both interfaces were moved to the dedicated `interfaces` folder.

Go back to Findings Summary

I6: `HashConsensus` should inherit from `IConsensusContract`

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

Description

The `HashConsensus` contract implements all functions from `IConsensusContract` interface and is externally called through the interface, but does not inherit from it.

Recommendation

Make the `HashConsensus` contract inherit from the `IConsensusContract` interface.

Fix 2.0

The `HashConsensus` contract now inherits from the `IConsensusContract` interface.

Go back to Findings Summary

I7: Redundant `whenPaused` check

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

Description

`CSFeeOracle.resume` is a privileged function used to resume the contract to normal operation after it has been paused.

Listing 53. Excerpt from CSFeeOracle

```
139 function resume() external whenPaused onlyRole(RESUME_ROLE) {
140     _resume();
141 }
```

The `whenPaused` modifier is applied to the function to ensure that it can only be called when the contract is paused. However, the same check is already performed in the internally called `_resume` function.

Recommendation

Remove the redundant `whenPaused` modifier from the `resume` function.

Fix 2.0

The redundant `whenPaused` modifier was removed from the codebase.

Go back to Findings Summary

I8: `pullFeeRewards` does not update depositable keys count

Impact:	Info	Likelihood:	N/A
Target:	CSAccounting.sol	Type:	Logic error

Description

Node operators are allowed to pull their rewards without claiming them to increase their bond and prevent the possibility of having unbonded keys in case of penalizations. The permissionless function `CSAccounting.pullFeeRewards` is responsible for pulling the rewards from the `CSFeeDistributor` contract to increase the node operator's bond.

Listing 54. Excerpt from CSAccounting

```
410 function pullFeeRewards(
411     uint256 nodeOperatorId,
412     uint256 cumulativeFeeShares,
413     bytes32[] calldata rewardsProof
414 ) external {
415     _onlyExistingNodeOperator(nodeOperatorId);
416     _pullFeeRewards(nodeOperatorId, cumulativeFeeShares, rewardsProof);
417 }
```

However, the function does not trigger recalculation of the depositable keys count of the node operator. This forces users to call another function to update the state in the case when pulling rewards could help cover currently unbonded validator keys.

Recommendation

Consider updating the depositable keys count after pulling the rewards to ensure the correct internal accounting.

Acknowledgment 2.0

The finding was acknowledged by Lido with the following comment:

> *The method can only increase bond balance and is not meant to be called by the Node operators. This method is introduced to allow pulling rewards before penalty application to ensure proper penalization.*
>
> — Lido

Go back to Findings Summary

I9: `CSBondCore._getClaimableBondShares` should be unimplemented

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

Description

The function `CSBondCore._getClaimableBondShares` is a base implementation marked as `virtual` that is overridden by the `CSAccounting` contract. The base implementation is never referenced in the codebase, only the overridden implementation is used.

Recommendation

For improved clarity and to avoid confusion, it is recommended to remove the body of the `CSBondCore._getClaimableBondShares` function making it unimplemented.

Acknowledgment 2.0

The finding was acknowledged by Lido with the following comment:

> *The original implementation of `CSBondCore._getClaimableBondShares` is preserved for the cases of partial CSM code reuse. One may consider creating an alternative version of `CSAccounting.sol` that would not include `CSBondCurve.sol` or `CSBondLock.sol`. Hence, the original implementation is kept for consistency.*
>
> — Lido

Go back to Findings Summary

I10: State variable read multiple times

Impact:	Info	Likelihood:	N/A
Target:	CSModule.sol	Type:	Gas optimization

Description

The function `CSModule.settleELRewardsStealingPenalty` loops over the provided node operator IDs and settles execution layer rewards stealing through the `CSAccounting` contract.

Listing 55. Excerpt from CSModule.settleELRewardsStealingPenalty

```
1062 for (uint256 i; i < nodeOperatorIds.length; ++i) {
1063     uint256 nodeOperatorId = nodeOperatorIds[i];
1064     _onlyExistingNodeOperator(nodeOperatorId);
1065     uint256 settled = accounting.settleLockedBondETH(nodeOperatorId);
1066     if (settled > 0) {
1067         // Bond curve should be reset to default in case of confirmed MEV
     stealing. See https://hackmd.io/@lido/SygBLW5ja
1068         accounting.resetBondCurve(nodeOperatorId);
```

The `accounting` state variable is guaranteed to be the same in the loop so it is not necessary to read it in each iteration, but the compiler is unable to optimize this.

Recommendation

Read the `accounting` variable once before the loop and use a local variable in the loop to reduce the number of SLOADs.

Fix 2.0

The `accounting` state variable is now cached in a local variable before the loop for better gas efficiency.

Go back to Findings Summary

I11: Inconsistent higher bits clearing in `QueueLib`

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

Description

`keys` and `next` are helper functions for the `QueueLib` library used to extract data stored in a packed user-defined value type `Batch` aliasing `uint256`.

Listing 56. Excerpt from QueueLib.sol

```
31 function keys(Batch self) pure returns (uint64 n) {
32     assembly {
33         n := shl(64, self)
34         n := shr(192, n)
35     }
36 }
37
38 function next(Batch self) pure returns (uint128 n) {
39     assembly {
40         n := self // uint128(self)
41     }
42 }
```

The `keys` function performs clearing of the higher bits of the `Batch` type while the `next` function does not. This is an inconsistency.

Recommendation

Consider unifying the behavior of the `keys` and `next` functions by either clearing the higher bits in both or leaving them as is.

Note that Solidity automatically clears the higher bits when casting an unsigned integer up to a larger type. However, this is not true for inline assembly blocks.

Fix 2.0

Higher bits are now also cleared in the `next` function.

Listing 57. Excerpt from QueueLib.sol

```
38 function next(Batch self) pure returns (uint128 n) {
39     assembly {
40         n := shl(128, self)
41         n := shr(128, n)
42     }
```

Go back to Findings Summary

I12: `QueueLib.clean` return last index of cleared item

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

Description

The `QueueLib.clean` function is responsible for removing queue items for node operators that have more keys enqueued than is the current amount of depositable keys.

The function accepts the `maxItems` parameter to limit the amount of items processed and returns `toRemove` as the number of items actually removed. The function caller can then perform a non-state-changing call to the function to get the estimated number of removed items.

It may be beneficial for the caller to also have the last index of a removed item returned by the function. This way, the caller may use the most efficient value of `maxItems` to process the queue.

Recommendation

Return the last index of a removed item from the `QueueLib.clean` function.

Fix 2.0

The `QueueLib.clean` function now returns the last index of a removed item. The value is propagated in the `CSModule.cleanDepositQueue` function to the caller.

Go back to Findings Summary

I13: Unused code

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

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 2.0

The interface files were kept in the codebase as they are used in tests and deploy scripts. The `_updateContractVersion` function was kept for consistency with the Lido protocol core codebase. The rest of the unused code occurrences was removed.

Go back to Findings Summary

I14: Incorrect documentation & typos

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

Description

There are several typos and incorrect statements in the documentation.

1. `HashConsensus` list item numbers

Listing 58. Excerpt from HashConsensus

```
46 ///   1. there previously was a consensus report; AND
47 ///   1. processing of the consensus report hasn't started yet; AND
48 ///   2. report processing deadline is not expired yet; AND
49 ///   3. there's no consensus report now (otherwise, `submitConsensusReport`
   is called instead).
```

The item numbers in the list should be 1, 2, 3, 4.

2. Left over `forceCall_` reference in `OssifiableProxy`

Listing 59. Excerpt from OssifiableProxy

```
80 /// @param setupCalldata_ Data for the setup call. The call is skipped if
   setupCalldata_ is
81 ///     empty and forceCall_ is false
82 // solhint-disable-next-line func-name-mixedcase
83 function proxy__upgradeToAndCall(
84     address newImplementation_,
85     bytes calldata setupCalldata_
86 ) external onlyAdmin {
87     ERC1967Utils.upgradeToAndCall(newImplementation_, setupCalldata_);
88 }
```

There is not such parameter as `forceCall_` in the `proxy__upgradeToAndCall` function.

3. Incorrect statement about `totalExitedKeys`

Listing 60. Excerpt from ICSModule

```
22 /* 2 */ uint32 totalExitedKeys; // @dev only increased
```

Although usually correct, the exited keys count may unsafely decrease in the `CSModule.unsafeUpdateValidatorsCount` function.

4. Inconsistent comments spacing in `SigningKeys`

Listing 61. Excerpt from SigningKeys

```
60 let _ofs := add(pubkeys.offset, mul(i, 48)) //PUBKEY_LENGTH = 48
61 let _part1 := calldataload(_ofs) // bytes 0..31
62 let _part2 := calldataload(add(_ofs, 0x10)) // bytes 16..47
```

Listing 62. Excerpt from SigningKeys

```
74 sstore(curOffset, mload(add(tmpKey, 0x20))) // store bytes 0..31
75 sstore(add(curOffset, 1), shl(128, mload(add(tmpKey, 0x30)))) // store bytes
   32..47
76 // store signature
77 let _ofs := add(signatures.offset, mul(i, 96)) //SIGNATURE_LENGTH = 96
```

One space should be used between `//` and the comment text for consistency.

Recommendation

Fix the typos and incorrect statements in the documentation.

Fix 2.0

All the typos and incorrect statements in the documentation were fixed.

Go back to Findings Summary

Report Revision 2.0

Revision Team

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

System Overview

Except for changes corresponding to the fixes of the previously reported findings, revision 2.0 contains the following additional changes:

- `_sharesByEth` and `_ethByShares` helper functions now early return 0 when the input is 0,
- resetting the bond curve was optimized if the default bond curve is already assigned,
- bond locking mechanism no longer uses an unchecked block to update the locked bond amount,
- `WSTETH.getWstETHByStET` calls were optimized in favor of `_sharesByEth`,
- ditribution log CID was introduced as a new parameter to the `CSFeeDistributor` contract,
- maximum key removal charge parameter was added to the `CSModule` contract,
- `pubkey` parameter was added to the `WithdrawalSubmitted` and `InitialSlashingSubmitted` events,
- it is no longer possible to report a zero execulation layer stealing amount,
- execution layer stealing compensation can only be performed by node managers,

- `CSModule.recoverStETHShares` was removed as it was redundant to the `recoverERC20` function.

Fuzzing

Fuzz tests created in revision 1.0 were updated to reflect changes in the codebase. Additionally, a new invariant IV20 was added.

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

Report Revision 3.0

Revision Team

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

System Overview

The validator initial slashing reporting functionality was removed from the `CSVerifier` contract in preparation for the upcoming Pectra hardfork. With this codebase update, penalization of validators for initial slashing is postponed until the validator withdrawal.

No additional changes were introduced.

Fuzzing

The fuzz tests created in the previous revisions were updated to reflect the removal of the initial slashing reporting feature. The tests are available at https://github.com/Ackee-Blockchain/tests-lido-csm/tree/revision-3.0.

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

Appendix A: How to cite

Please cite this document as:

> Ackee Blockchain Security, Lido: Community Staking Module, 8.4.2025.

Appendix B: Wake Findings

This section lists the outputs from the Wake tool used for fuzz 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	1.0
F2	Addition of new validator keys to existing node operators with all supported token types	1.0
F3	Deposits of additional tokens into node operator bonds	1.0
F4	Removal of validator keys from node operators	1.0
F5	Addition of new consensus members responsible for rewards distribution report voting	1.0
F6	Removal of consensus members	1.0
F7	Submission and voting on rewards distribution reports	1.0
F8	Pulling of rewards by node operators	1.0
F9	Claiming of rewards by node operators with all supported token types	1.0
F10	Reporting of execution layer rewards stealing	1.0
F11	Cancellation of execution layer rewards stealing reports	1.0
F12	Settlement of execution layer rewards stealing reports	1.0
F13	Compensation of stolen execution layer rewards	1.0

ID	Flow	Added
F14	Obtaining of validator keys to be deposited by Staking Router	1.0
F15	Normalization of the Community Staking Module queue	1.0
F16	Cleaning of the queue	1.0
F17	Updating of target validators limits and modes	1.0
F18	Updating of stuck validators count	1.0
F19	Updating of exited validators count	1.0
F20	Decrease of vetted signing keys count	1.0
F21	Unsafe change of exited and stuck validators count	1.0
F22	Processing of (possibly historical) validator withdrawal proofs	1.0
F23	Processing of validator slashing proofs	1.0

Table 4. Wake fuzzing flows

The following table lists the invariants checked after each flow.

ID	Invariant	Added	Status
IV1	All important account balances are matching expected values	1.0	Success
IV2	All stETH share balances are matching expected values	1.0	Success
IV3	Difference between expected amount of tokens needed for new validator keys and actual amount is within 10 wei	1.0	Success
IV4	Bond information (including locked bonds) is matching expected values	1.0	Success

ID	Invariant	Added	Status
IV5	Bond curve is reset on confirmed slashing and execution layer rewards stealing	1.0	**Fail** (L6)
IV6	Node operator signing keys and signatures stored in the `CSModule` contract match expected values	1.0	Success
IV7	Numbers of node operator added keys, withdrawn keys, stuck keys, deposited keys, exited keys, vetted keys, enqueued keys match expected values	1.0	Success
IV8	Normalized node operator depositable key counts match expected values	1.0	**Fail** (M1, L8)
IV9	Node operator target validator limits and modes match expected values	1.0	**Fail** (L7)
IV10	Node operator rewards addresses and manager addresses match expected values	1.0	Success
IV11	Node operator slashing and withdrawal status is correct	1.0	Success
IV12	Community Staking Module queue follows expected structure	1.0	Success
IV13	Module nonce is incrementing correctly	1.0	Success
IV14	Withdrawal of all active node operators does not cause accounting issues	1.0	Success
IV15	Contracts emit expected events with correct parameters	1.0	**Fail** (W15)
IV16	Application for early adoption program works correctly	1.0	Success

ID	Invariant	Added	Status
IV17	Claiming of rewards by node operators works correctly	1.0	**Fail** (L3)
IV18	Transactions do not revert except where explicitly expected	1.0	**Fail** (L4, W8)
IV19	Validator keys and signatures obtained for deposits are in correct order and match expected values	1.0	Success
IV20	`cleanDepositQueue` returns correct values of removed items count and last removed item index	2.0	Success

Table 5. Wake fuzzing invariants

B.2. Detectors

This section contains vulnerability and code quality detections from the Wake tool.

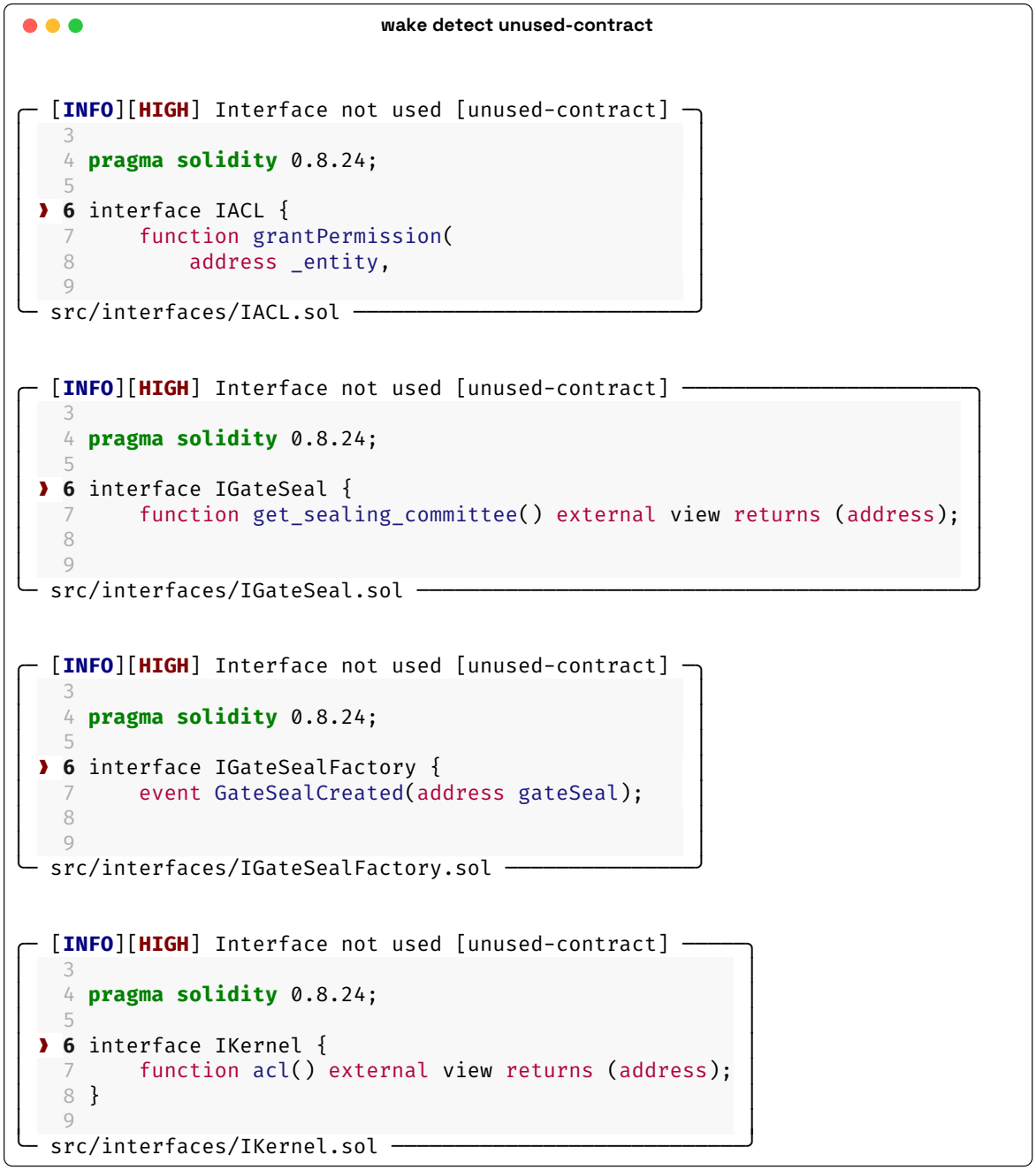


Figure 1. Unused interfaces

```
wake detect unused-contract

[INFO][HIGH] Interface not used [unused-contract]
  3
  4 pragma solidity 0.8.24;
  5
❱ 6 interface IStakingRouter {
  7     event ContractVersionSet(uint256 version);
  8     event ExitedAndStuckValidatorsCountsUpdateFailed(
  9
src/interfaces/IStakingRouter.sol

[INFO][HIGH] Interface not used [unused-contract]
  3
  4 pragma solidity 0.8.24;
  5
❱ 6 interface IVEBO {
  7     function SUBMIT_DATA_ROLE() external view returns (bytes32);
  8
  9
src/interfaces/IVEBO.sol
```

Figure 2. Unused interfaces (continued)

```
wake detect unused-error

[INFO][HIGH] Unused error [unused-error]
 152     error SigningKeysInvalidOffset();
 153
 154     error AlreadySubmitted();
❱ 155     error AlreadyWithdrawn();
 156
 157     error AlreadyActivated();
 158
src/CSModule.sol
```

Figure 3. Unused error

```
wake detect unused-using-for

[WARNING][LOW] Unused contract in using-for directive [unused-using-for]
 30     AccessControlEnumerableUpgradeable,
 31     AssetRecoverer
 32 {
❱ 33     using SafeERC20 for IERC20;
 34
 35     bytes32 public constant PAUSE_ROLE = keccak256("PAUSE_ROLE");
 36
src/CSAccounting.sol
```

Figure 4. Unused using-for directive

Figure 5. Unused functions

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz