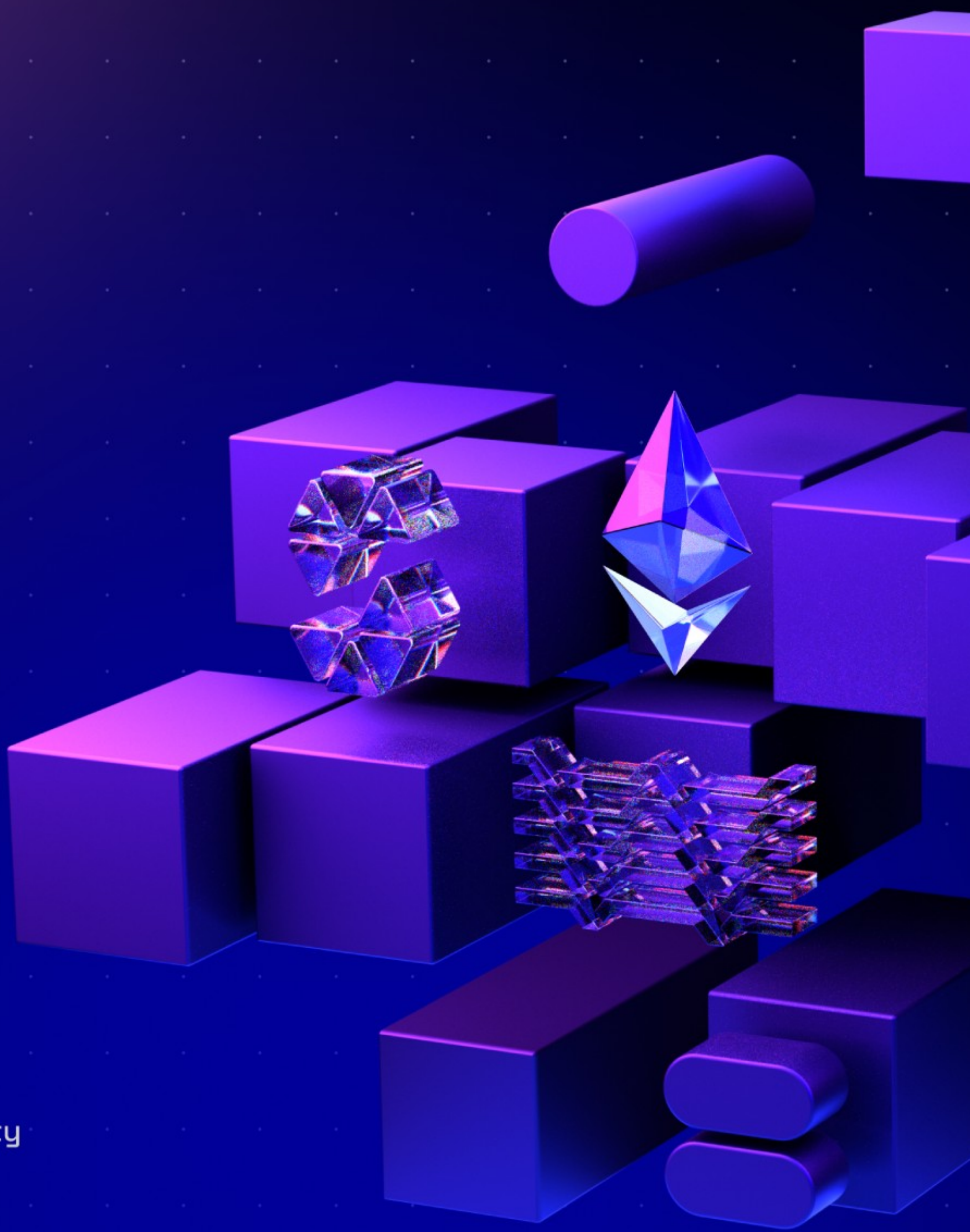
ackee
blockchain security

Lido

Staking Router V2

14.10.2024



Ackee Blockchain Security

Contents

1. Document Revisions

1.0-draft	Draft Report	23.08.2024
1.0	Final Report	05.09.2024
1.1	Fix Review	08.10.2024
1.1	Update fix review commit	08.10.2024
1.2	Deployment Verification	14.10.2024

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 for VS Code 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
Jan Kalivoda	Lead Auditor
Andrey Babushkin	Auditor
Naoki Yoshida	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 is a protocol that allows to stake ETH. Lido Staking Router V2 is a component that allows to utilize modular design with a support for permissionless staking modules.

Revision 1.0

Lido team engaged Ackee Blockchain Security to perform a security review of the Lido protocol with a total time donation of 26 engineering days, where 6 days were dedicated to fuzzing, in a period between July 22 and August 23, 2024, with Jan Kalivoda as the lead auditor.

The audit was performed on the commit `fafa23`[1] and the scope was the following:

- contracts/0.4.24/nos/NodeOperatorsRegistry.sol
- contracts/0.8.9/DepositSecurityModule.sol
- contracts/0.8.9/StakingRouter.sol
- contracts/common/lib/MinFirstAllocationStrategy.sol
- contracts/0.8.9/oracle/AccountingOracle.sol
- contracts/0.8.9/sanity_checks/OracleReportSanityChecker.sol

We began our review using static analysis tools, including Wake. We then took a deep dive into the logic of the contracts. For testing and fuzzing, we have involved Wake testing framework. We've created a Python model of the Lido protocol with the new Curated Staking Module (NodeOperatorsRegistry) and Community Staking Module (CSM) as modules. All the in-scope contracts were deployed, including the CSM codebase, the remaining protocol architecture was forked from the mainnet. On the Python model, we have built a manually guided fuzzing campaign, with flows[2] implemented for each function in the contracts. Finally, we've defined several stateful invariants where most notable ones are:

- the Python state is the same as in the contracts (differential testing approach),
- invariants on key counts (e.g. deposited keys count is always less or equal to the vetted keys count),
- correct incrementation of nonces,

and more stateless checks, such as correct event emission, etc. Fuzz tests were updated during the review and ran for days to ensure the system behaves to our expectations. It helped us to discover some inconsistency scenarios, such as L2 issue. For a complete list of fuzzing invariants and flows, see Appendix B.1.

During the review, we paid special attention to:

- exploring a potential attack surface of the core contracts because of introducing permissionless staking modules,
- new unvetting and pausing mechanism in DepositSecurityModule,
- possible guardians misbehaving, including signature replays and correct nonces usage,
- multi-transactional third-phase reports from accounting oracle,
- permissionless reward distribution in NodeOperatorsRegistry,
- ensuring overall access controls are not too relaxed or too strict,
- and looking for common issues such as data validation.

Our review resulted in 7 findings, ranging from Info to Low severity. The codebase is very solid, well documented and the team was always responsive.

Ackee Blockchain Security recommends Lido:

- address all the reported issues.

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

Revision 1.1

The fix review was conducted on the given commit `1ffbb7`[3]. The scope were the fixes from the previous revision. All findings were fixed except W1 and W2, which were acknowledged. See updated findings for more details.

Revision 1.2

Lido engaged Ackee Blockchain Security to perform deployment verification of Staking Router V2 on the Ethereum mainnet. The verification was performed on the same commit as in the previous revision, `1ffbb7`[4].

The verification concluded successfully with an exact match for bytecode without metadata hash achieved for all scoped contracts at the following addresses on the Ethereum mainnet:

- MinFirstAllocationStrategy: 0x7e70De6D1877B3711b2bEDa7BA00013C7142d993
- StakingRouter: 0x89eDa99C0551d4320b56F82DDE8dF2f8D2eF81aA
- NodeOperatorsRegistry: 0x1770044a38402e3CfCa2Fcfa0C84a093c9B42135
- DepositSecurityModule: 0xfFA96D84dEF2EA035c7AB153D8B991128e3d72fD
- AccountingOracle: 0x0e65898527E77210fB0133D00dd4C0E86Dc29bC7
- OracleReportSanityChecker: 0x6232397ebac4f5772e53285B26c47914E9461E75

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

[1] full commit hash: `fafa232a7b3522fdee5600c345b5186b4bcb7ada`

[2] A test step/scenario during fuzzing (see https://ackee.xyz/wake/docs/latest/testing-framework/fuzzing/#flows)

[3] full commit hash: `1ffbb7e49e112fcac678f59bf63ba57a7e522874`

[4] full commit hash: `1ffbb7e49e112fcac678f59bf63ba57a7e522874`

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* and
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
0	0	0	3	2	2	7

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
L1: Overflow on type casting	Low	1.0	Fixed
L2: Potential revert on underflow	Low	1.0	Fixed
L3: The `clearNodeOperatorPenalty` returns always `false`	Low	1.0	Fixed
W1: Pausing deposits is susceptible to replay	Warning	1.0	Acknowledged
W2: Refunding validators can cause future failed exits unpenalized	Warning	1.0	Acknowledged

Finding title	Severity	Reported	Status
I1: Missing event on `clearNodeOperatorPenalty`	Info	1.0	Fixed
I2: Typos	Info	1.0	Fixed

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Jan Kalivoda	Lead Auditor
Andrey Babushkin	Auditor
Naoki Yoshida	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

The scope included the core contracts that required adjustments to support the Community Staking Module (CSM). Significant changes were made to the DepositSecurityModule (DSM), which can now unvet keys for any module or pause deposits for all of them. Additionally, the NodeOperatorsRegistry (NOR) now supports permissionless reward distribution and different target limits for exit orders. Moreover, the third-phase report from the AccountingOracle (AO) is multi-transactional, enabling it to handle an increasing number of node operators due to the growing user base associated with CSM. Lastly, the StakingRouter (SR) has been updated to reflect the new design decisions.

Trust Model

The core of the protocol stays permissioned and defines numerous roles. In general, users have to trust Lido and Lido DAO to set all the parameters correctly. For example, guardians (in terms of DSM) can potentially harm the protocol, since they can unvet all the keys or pause deposits for all modules.

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.1.

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

Findings

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

L1: Overflow on type casting

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	StakingRouter.sol	Type:	Overflow/Underflow

Description

The `_updateStakingModule` function is susceptible to overflow on type downcasting for the `_maxDepositsPerBlock` and `_minDepositBlockDistance` parameters.

Listing 1. Excerpt from [StakingRouter]()

```
stakingModule.maxDepositsPerBlock = uint64(_maxDepositsPerBlock);
stakingModule.minDepositBlockDistance = uint64(_minDepositBlockDistance);
```

Exploit scenario

The `updateStakingModule` function is called with `_minDepositBlockDistance` set to 2**64. As a result, the `_minDepositBlockDistance` is set to 0 even though it shouldn't be according the following requirement. Additionaly, the changed value is emitted as an event in `uint256` so it can be left undetected.

Listing 2. Excerpt from [StakingRouter]()

```
if (_minDepositBlockDistance == 0) revert InvalidMinDepositBlockDistance();
```

Recommendation

Add a data validation for required range to prevent overflow.

Fix 1.1

The check against overflow is added.

[Go back to Findings Summary]

L2: Potential revert on underflow

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	StakingRouter.sol	Type:	Overflow/Underflow

Description

The `unsafeSetExitedValidatorsCount` function can be called in rare cases. However, it is possible to set higher value of exited keys than deposited and thus break the invariant. As a result, unexpected behavior can be observed. Such as, integer underflow revert in the following occurrences.

Listing 3. Excerpt from [StakingRouter]()

```
activeValidatorsCount = totalDepositedValidators - Math256.max(
    stakingModule.exitedValidatorsCount, totalExitedValidators
);
```

Listing 4. Excerpt from [StakingRouter]()

```
cacheItem.activeValidatorsCount =
    totalDepositedValidators -
    Math256.max(totalExitedValidators,
stakingModuleData.exitedValidatorsCount);
```

Exploit scenario

The `unsafeSetExitedValidatorsCount` function is called, making the count of exited keys higher than deposited. As a result, deposits for the specific staking module are blocked by revert on underflow.

Recommendation

Add a data validation to the `unsafeSetExitedValidatorsCount` function to hold the invariants (exited <= deposited, stucked <= deposited - exited). Even though there is an unsafe function to correct the state, the state should be always corrected to hold the pre-defined invariants.

Fix 1.1

The data validation to hold the invariant is added.

[Go back to Findings Summary]

L3: The `clearNodeOperatorPenalty` returns always `false`

Low severity issue

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

Description

The `clearNodeOperatorPenalty` function defines boolean return value but never uses it. It can be expected to return `true` if the penalty is cleared and `false` otherwise. However, it always returns `false`.

Listing 5. Excerpt from [NodeOperatorsRegistry]()

```
function clearNodeOperatorPenalty(uint256 _nodeOperatorId) external returns
(bool) {
    Packed64x4.Packed memory stuckPenaltyStats =
_loadOperatorStuckPenaltyStats(_nodeOperatorId);
    require(
        !_isOperatorPenalized(stuckPenaltyStats) &&
stuckPenaltyStats.get(STUCK_PENALTY_END_TIMESTAMP_OFFSET) != 0,
        "CANT_CLEAR_PENALTY"
    );
    stuckPenaltyStats.set(STUCK_PENALTY_END_TIMESTAMP_OFFSET, 0);
    _saveOperatorStuckPenaltyStats(_nodeOperatorId, stuckPenaltyStats);
    _updateSummaryMaxValidatorsCount(_nodeOperatorId);
    _increaseValidatorsKeysNonce();
}
```

Exploit scenario

The `clearNodeOperatorPenalty` function is called, succeeds, and returns `false`.

Recommendation

Implement the return values to match the expected behavior.

Fix 1.1

The return value is implemented and the function returns `true` on clearing the penalty.

[Go back to Findings Summary]

W1: Pausing deposits is susceptible to replay

Impact:	Warning	Likelihood:	N/A
Target:	DepositSecurityModule.sol	Type:	Replay attack

Description

The `pauseDeposits` function is susceptible to a signature replay attack. Once a guardian exposes the signature to pause the protocol, it is possible to replay the same signature until the pause intent expires.

Only the owner (Lido DAO) can unpause deposits. If the owner reacts promptly and the pause intent has not yet expired, anyone can pause the protocol again. As a result, when the pause signature is exposed, it can be guaranteed that the protocol will remain paused for the entire duration of the pause intent, because only the pause intent block number serves as an invalidation element.

Listing 6. Excerpt from [DepositSecurityModule]()

```
function pauseDeposits(uint256 blockNumber, Signature memory sig) external {
    /// @dev In case of an emergency function `pauseDeposits` is supposed to be called
    /// by all guardians. Thus only the first call will do the actual change. But
    /// the other calls would be OK operations from the point of view of protocol’s logic.
    /// Thus we prefer not to use “error” semantics which is implied by `require`.
    if (isDepositsPaused) return;

    address guardianAddr = msg.sender;
    int256 guardianIndex = _getGuardianIndex(msg.sender);

    if (guardianIndex == -1) {
        bytes32 msgHash = keccak256(abi.encodePacked(PAUSE_MESSAGE_PREFIX, blockNumber));
        guardianAddr = ECDSA.recover(msgHash, sig.r, sig.vs);
        guardianIndex = _getGuardianIndex(guardianAddr);
```

```
        if (guardianIndex == -1) revert InvalidSignature();
    }

    if (block.number - blockNumber > pauseIntentValidityPeriodBlocks) revert
PauseIntentExpired();

    isDepositsPaused = true;
    emit DepositsPaused(guardianAddr);
}
```

Recommendation

Each signature should be invalidated after use; for example, by implementing nonce logic.

Acknowledgment 1.1

> *This is a known protocol behaviour. The current design of the DSM assumes that unpausing deposits can only be done through on-chain DAO voting.*
>
> *The voting duration is significantly longer than the expiration time of any pause intention. By the time the DAO has voted to unpause, all previously signed pause intentions would have expired, thereby mitigating the risk of replay attacks.*
>
> *Therefore, no changes to the current implementation are necessary.*
>
> — Lido Team

Go back to Findings Summary

W2: Refunding validators can cause future failed exits unpenalized

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

Description

If the refunded validator counts remains at a high value (after first use), then even with changes in the count of stuck validators, such node operators are not penalized.

Listing 7. Excerpt from [NodeOperatorsRegistry]()

```
function _isOperatorPenalized(Packed64x4.Packed memory stuckPenaltyStats)
internal view returns (bool) {
    return stuckPenaltyStats.get(REFUNDED_VALIDATORS_COUNT_OFFSET) <
stuckPenaltyStats.get(STUCK_VALIDATORS_COUNT_OFFSET)
        || block.timestamp <=
stuckPenaltyStats.get(STUCK_PENALTY_END_TIMESTAMP_OFFSET);
}
```

As discussed with the team, the likelihood of this issue is low because the condition is difficult to satisfy, also given that the behavior of the node operator is detectable through off-chain monitoring.

Exploit scenario

The following steps could occur:

1. The node operator is requested to exit 10 validators.
2. The off-chain element attempts to exit 10 validators, successfully exits 6, and the remaining 4 validators become stuck.
3. The Accounting Oracle submits a report stating that the node operator has 4 stuck validators.

4. The node operator is forced to exit these 4 validators.
5. The count of refunded validators increases as the stuck validators are exited.
6. The count of refunded validators now matches the count of stuck validators at 4, and it sets the penalty timestamp with a delay.
7. After the penalty timestamp has passed, the node operator is not penalized.
8. The node operator is requested to exit 9 validators.
9. The node operator exits 6 of the validators, and the remaining 3 validators become stuck.
10. The Accounting Oracle submits a report stating that the node operator has 3 stuck validators.
11. This node operator should now be penalized, but the refunded validator count remains at 4, and the penalty timestamp has already passed, so the node operator is not penalized.
12. Reward distribution occurs, and the node operator receives the full amount of rewards for all deposited and unexited validators, including those that are stuck.

This situation is more likely to occur when the stuck penalty delay is set to a low value.

Recommendation

Be aware of this behavior, that the refunded validator counts are not handled automatically and has to be treated manually or implement the logic to reset the value of refunded validator counts.

Acknowledgment 1.1

> *It's true that if a validator becomes stuck, gets refunded, and later exits successfully, a positive difference between refunded and stuck validators might occur. However, refunded validators are handled operationally by the DAO, and there is no issue here— the values are simply updated as needed.*
>
> *There is no negative impact on the protocol. In fact, the outcome is positive: the operator has paid the refund, and the validator ultimately exited. Should a validator become stuck again, we can address the issue manually through DAO voting.*
>
> *Therefore, no changes are required to the current implementation, and the protocol is functioning as intended.*
>
> — Lido Team

[Go back to Findings Summary]

I1: Missing event on `clearNodeOperatorPenalty`

Impact:	Info	Likelihood:	N/A
Target:	NodeOperatorsRegistry.sol	Type:	Logging

Description

The `clearNodeOperatorPenalty` function does not emit an event even though it changes important state. As a result, for logging purposes it is only observable that the staking module's nonce was incremented.

Listing 8. Excerpt from [NodeOperatorsRegistry]()

```
function clearNodeOperatorPenalty(uint256 _nodeOperatorId) external returns (bool) {
    Packed64x4.Packed memory stuckPenaltyStats = _loadOperatorStuckPenaltyStats(_nodeOperatorId);
    require(
        !_isOperatorPenalized(stuckPenaltyStats) && stuckPenaltyStats.get(STUCK_PENALTY_END_TIMESTAMP_OFFSET) != 0,
        "CANT_CLEAR_PENALTY"
    );
    stuckPenaltyStats.set(STUCK_PENALTY_END_TIMESTAMP_OFFSET, 0);
    _saveOperatorStuckPenaltyStats(_nodeOperatorId, stuckPenaltyStats);
    _updateSummaryMaxValidatorsCount(_nodeOperatorId);
    _increaseValidatorsKeysNonce();
}
```

Recommendation

Add an event to the `clearNodeOperatorPenalty` function.

Fix 1.1

The event is added.

[Go back to Findings Summary]()

I2: Typos

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

Description

The codebase contains typos. There were identified the following occurrences:

- `_updateVettedSingingKeysCount` should be `_updateVettedSigningKeysCount`

Listing 9. Excerpt from [NodeOperatorsRegistry]()

```
function _updateVettedSingingKeysCount(
    uint256 _nodeOperatorId,
    uint256 _vettedSigningKeysCount,
    bool _allowIncrease
) internal {
```

Recommendation

Fix the typos.

Fix 1.1

The typos are fixed.

[Go back to Findings Summary]()

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, Lido: Staking Router V2, 14.10.2024.

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	Setting of NodeOperatorRegistry to StakingRouter	1.0
F2	Updating of StakingModule configuration	1.0
F3	Updating of TargetValidatorsLimits in StakingRouter	1.0
F4	Updating of StakingRouter configuration	1.0
F5	Updating of RefundedValidatorsCount in StakingRouter	1.0
F6	Updating of Exited and Stucked validators count through unsafe function	1.0
F7	Setting of StakingModule status from StakingRouter	1.0
F8	Setting of withdrawal credential	1.0
F9	Updating of PauseIntentValidityPeriodBlocks value	1.0
F10	Updating of MaxOperatorsPerUnvetting value	1.0
F11	Updating of the guardian quorum in DepositSecurityModule	1.0
F12	Addition of guardian in DepositSecurityModule	1.0
F13	Addition of multiple guardians in DepositSecurityModule	1.0
F14	Removal of guardian in DepositSecurityModule	1.0
F15	Pausing of deposit from DepositSecurityModule	1.0

ID	Flow	Added
F16	Unpausing of deposit from DepositSecurityModule	1.0
F17	Depositing of Buffered Ether from DepositSecurityModule	1.0
F18	Unvetting of signing key from DepositSecurityModule	1.0
F19	Addition of the node operator to NodeOperatorRegistry	1.0
F20	Activation of the node operator in NodeOperatorRegistry	1.0
F21	Deactivation of the node operator in NodeOperatorRegistry	1.0
F22	Updating of the node operator name in NodeOperatorRegistry	1.0
F23	Updating of the node operator reward address in NodeOperatorRegistry	1.0
F24	Updating of the node operator staking limit in NodeOperatorRegistry	1.0
F25	Addition of the signing key to NodeOperatorRegistry	1.0
F26	Removal of the signing key from NodeOperatorRegistry	1.0
F27	Clearing of the node operator penalty to NodeOperatorRegistry	1.0
F28	Updating of stuck penalty delay of the node operator in NodeOperatorRegistry	1.0
F29	Distribution of reward in NodeOperatorRegistry	1.0
F30	Obtaining of stETH by submitting ETH to LIDO	1.0
F31	Request withdraw to WithdrawalQueueERC712	1.0
F32	Claiming of the withdrawal by withdrawal id from WithdrawalQueueERC712	1.0
F33	Submission of the report to ValidatorsExitBusOracle	1.0

ID	Flow	Added
F34	Submission of the report to AccountingOracle	1.0
F35	Submission of the extra report data report to AccountingOracle	1.0

Table 4. Wake fuzzing flows

The following table lists the invariants checked after each flow.

ID	Invariant	Added	Status
IV1	All node operators key counts hold the state	1.0	Success
IV2	All configuration parameters of the node operators are matching expected values	1.0	Success
IV3	All key are matching expected values	1.0	Success
IV4	All key state is matching expected values between the counts	1.0	Success
IV5	All node operators count are matching expected values	1.0	Success
IV6	All shares of reward accounts in NodeOperatorRegistry are matching expected values	1.0	Success
IV7	All node operators reward account are matching expected values	1.0	Success
IV8	Total exited and stuck validator counts in NodeOperatorRegistry are matching expected values after extra report data submission	1.0	Success
IV9	Lido beacon state values are matching expected values	1.0	Success

ID	Invariant	Added	Status
IV10	All user's stETH share are matching expected values	1.0	Success
IV11	Guardian quorum of the DepositSecurityModule is matching expected value	1.0	Success
IV12	All staking modules parameters in the StakingRouter are matching expected values	1.0	Success
IV13	Staking modules count in the StakingRouter are matching expected values	1.0	Success
IV14	All staking module nonce values are matching expected value	1.0	Success
IV15	Transactions do not revert except where explicitly expected	1.0	Success
IV16	Contracts emit expected events with correct parameters	1.0	Success

Table 5. Wake fuzzing invariants

B.2. Detectors

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

```
wake detect missing-return

[WARNING][MEDIUM] Not all execution paths have assigned return values [missing-return]
        return !_isOperatorPenalized(stuckPenaltyStats) && stuckPenaltyStats.get(STUCK_PENALTY_
    }

    function clearNodeOperatorPenalty(uint256 _nodeOperatorId) external returns (bool) {
        Packed64x4.Packed memory stuckPenaltyStats = _loadOperatorStuckPenaltyStats(_nodeOperat
        console.log("isOperatorPenalized", _isOperatorPenalized(stuckPenaltyStats));

tests/migrated_contracts/NodeOperatorsRegistryMigrated.sol

[WARNING][MEDIUM] Not all execution paths have assigned return values [missing-return]

    /// @notice distributes rewards among node operators
    // @return the amount of stETH shares distributed among node operators
    function _distributeRewards() internal returns (uint256 distributed) {
        IStETH stETH = IStETH(getLocator().lido());


tests/migrated_contracts/NodeOperatorsRegistryMigrated.sol
```

Figure 1. Missing returns

We have migrated NodeOperatorRegistry to a newer Solidity version to be able to run a static analysis. This helped us to find L3 issue. The other detection for the same detector is valid but not an issue.

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz