

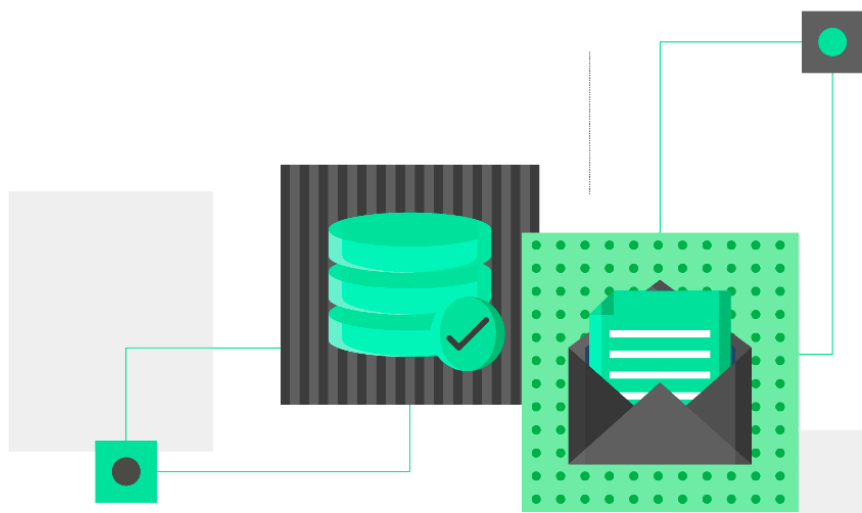
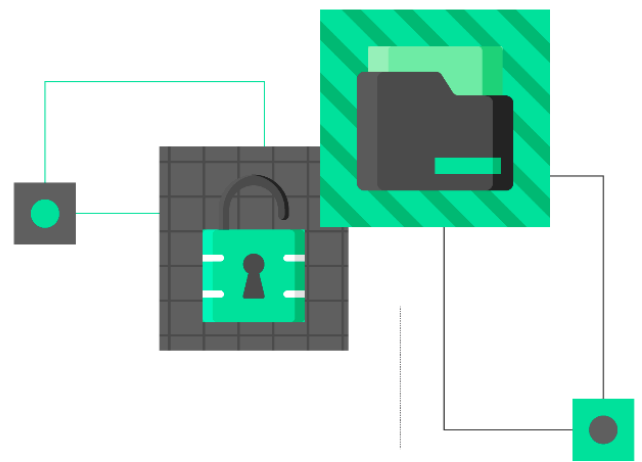


Mantiseclabs

Smart Contract Audit

NodeStaking.sol
Presearch

July 2025
Sep 2025



Contents

Disclaimer	3
Audit Process & Methodology	4
Audit Purpose	5
Contract Details	5
Security Level Reference	6
Findings	7
Additional Details	20
Concluding Remarks	21



Disclaimer

This disclaimer is to inform you that the report you are reading has been prepared by Mantiseclabs for informational purposes only and should not be considered as investment advice. It is important to conduct your own independent investigation before making any decisions based on the information contained in the report. The report is provided "as is" without any warranties or conditions of any kind and Mantiseclabs excludes all representations, warranties, conditions, and other terms. Additionally, Mantiseclabs assumes no liability or responsibility for any kind of loss or damage that may result from the use of this report.

It is important to note that the analysis in the report is limited to the security of the smart contracts only and no applications or operations were reviewed. The report contains proprietary information, and Mantiseclabs holds the copyright to the text, images, photographs, and other content. If you choose to share or use any part of the report, you must provide a direct link to the original document and credit Mantiseclabs as the author.

By reading this report, you are accepting the terms of this disclaimer. If you do not agree with these terms, it is advisable to discontinue reading the report and delete any copies in your possession.

Audit Process & Methodology

The MantiseC Labs team carried out a thorough audit for the project, starting with an in-depth analysis of code design patterns. This initial step ensured the smart contract's architecture was well-structured and securely integrated with third-party smart contracts and libraries. Also, our team conducted a thorough line-by-line inspection of the smart contract, seeking out potential issues such as Signature Replay Attacks, Unchecked External Calls, External Contract Referencing, Variable Shadowing, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, among others.

During the Unit testing phase, we assessed the functions authored by the developer to ascertain their precise functionality. Our Automated Testing procedures leveraged proprietary tools designed in-house to spot vulnerabilities and security flaws within the Smart Contract. The code was subjected to an in-depth audit administered by an independent team of auditors, encompassing the following critical aspects:

- Scrutiny of the smart contract's structural analysis to verify its integrity.
- Extensive automated testing of the contract
- A manual line-by-line Code review, undertaken with the aim of evaluating, analyzing, and identifying potential security risks.
- An evaluation of the contract's intended behavior, encompassing a review of provided documentation to ensure the contract conformed to expectations.
- Rigorous verification of storage layout in upgradeable contracts.
- An integral component of the audit procedure involved the identification and recommendation of enhanced gas optimization techniques for the contract

Audit Purpose

Mantisec Labs was hired by the Presearch team to review their smart contract. This audit was conducted in July & Sep 2025.

The main reasons for this review were:

- To find any possible security issues in the smart contract.
- To carefully check the logic behind the given smart contract.

This report provides valuable information for assessing the level of risk associated with this smart contract and offers suggestions on how to improve its security by addressing any identified issues.

Contract Details

Project Name	Presearch
Contract links	NodeStaking.sol
Language	Solidity
Type	ERC20

Security Level Reference

Each problem identified in this report has been categorized into one of the following severity levels:

- **Critical:** Vulnerabilities that present an immediate and serious threat to system or data integrity, demanding urgent action.
- **High:** Significant risks that have the potential to cause major security breaches or loss of functionality.
- **Medium:** Issues that moderately affect system performance or security and require timely resolution.
- **Low:** Low-risk concerns primarily related to optimization and code quality, with minimal direct impact on system security.
- **Informational:** Observations or recommendations that do not pose any direct risk but provide insights for potential improvements or best practices.

Severity	Score
Critical	4-5
High	3-4
Medium	2-3
Low	1-2
Informational	0-1

Findings Overview

Contract Names:

- `NodeStaking.sol`

Critical	High	Medium	Low	Informational
0	0	3	2	0

Issue	Severity	Fix Date
M01- Missing Emergency Pause Mechanism	Medium(2)	10-07-2025
M02- Signature Hash Not Fully Typed	Medium(2)	10-07-2025
M03- No Domain Separation in Signatures	Medium(2)	10-07-2025
L01- Unbounded batchId String Length	Low(1)	10-07-2025
L02- Unsafe Token Transfers Without SafeERC20	Low(1)	21-07-2025

Findings Details

[NodeStaking.sol](#)

M01- Missing Emergency Pause Mechanism

Severity: Medium

Impact: Inability to halt staking or migrations during a critical incident

Status: Patched

Issue:

The `NodeStaking` contract lacks a `pause()` mechanism to halt critical functionality such as staking, unstaking, and batch migrations in case of emergencies, exploits, or misconfigurations. This limits the admin's ability to respond to unforeseen issues and increases risk to user funds.

Affected Functions: `stake(...)`, `unstake(...)`, `migrateBatchNodeStakes(...)`

Recommendation:

Integrate the `Pausable` contract from OpenZeppelin and apply the `whenNotPaused` modifier to key state-changing functions. Additionally, expose `pause()` and `unpause()` functions restricted to the contract admin.

M02- Signature Hash Not Fully Typed (`abi.encodePacked` Used)

Severity: Medium

Impact: Risk of hash collisions or signature ambiguity

Status: Patched

Issue:

The contract uses `abi.encodePacked` when generating the message hash for signatures in `_verifyStakeSignature` and `_verifyBatchSignature`. This can lead to ambiguous encoding for dynamic types (like `string`) and may create collisions if not used carefully.

```
bytes32 messageHash = keccak256(  
    abi.encodePacked(msg.sender, amount, nodeId, batchId, deadline, nonce)  
);
```

If two different sets of parameters result in the same packed byte sequence (especially due to dynamic types like `string`), an attacker might forge a valid signature for unintended data.

Recommendation:

Use `abi.encode` instead of `abi.encodePacked` for unambiguous encoding.

M03- No Domain Separation in Signatures

Severity: Medium

Impact: Signature replay across networks or contracts

Status: Patched

Issue:

The contract constructs ECDSA message hashes without including a domain separator (e.g., `chainId`, `contract address`). As a result, a valid signature could potentially be replayed:

- on a different contract with the same message format
- on another chain with the same `msg.sender`

Impact Scenario:

If two chains or contracts share the same signer logic, an off-chain signature intended for one can be reused on the other, unless additional checks are in place.

Recommendation:

Add domain separation by including `address(this)` and `block.chainid` in the signed message.

L01- Unbounded `batchId` String Length

Severity: **Low** (Performance-focused)

Impact: Increased Gas Cost, Potential Denial-of-Service (DoS) Vector

Status: **Patched**

Issue:

The contract uses `string` for `batchId` in `NodeMigration` and related mappings (`_usedBatchIds`). This introduces unnecessary gas overhead due to dynamic string handling, especially in mappings and comparison operations (`_usedBatchIds[batchId]`).

Risk:

String comparisons are more expensive than fixed-size types like `bytes32`, and if exploited (e.g., by passing large or varied strings), could lead to gas exhaustion and DoS attacks on the migration process.

Recommendation:

Replace `string` with `bytes32` for `batchId` to reduce gas costs and simplify validation.

Benefit:

Using `bytes32` enables deterministic formatting (e.g., keccak256 hashes) and can improve indexing if events are emitted with batch IDs.

L02- Unsafe Token Transfers Without SafeERC20

Severity: Low

Location: `stake()`, `unstake()`, `migrateBatchNodeStakes()`, `recoverTokens()`

Status: Patched

Description:

Although `try/catch` blocks are used to handle ERC-20 transfer reverts, they do not protect against tokens that fail silently by returning `false`—a behavior observed in tokens like USDT and BNB. As a result, the contract may assume a successful transfer when it actually failed.

Recommendation:

Adopt OpenZeppelin's latest `SafeERC20` library for all token operations to enforce proper success checks. Optionally, `try/catch` can still be used around `SafeERC20` calls to guard against unexpected reverts (e.g., from underlying system calls), but `SafeERC20` must be the primary enforcement mechanism.

Additional Review – New Functions in Node Staking Contract

As part of the follow-up assessment, two newly introduced functions were reviewed in the Node Staking contract:

- `mainNodeStake`
- `mainNodeUnstake`

These functions utilize a separate mapping (`mainNodeStakes`) for tracking and include a flag mechanism to enable or disable the `mainNodeStake` method.



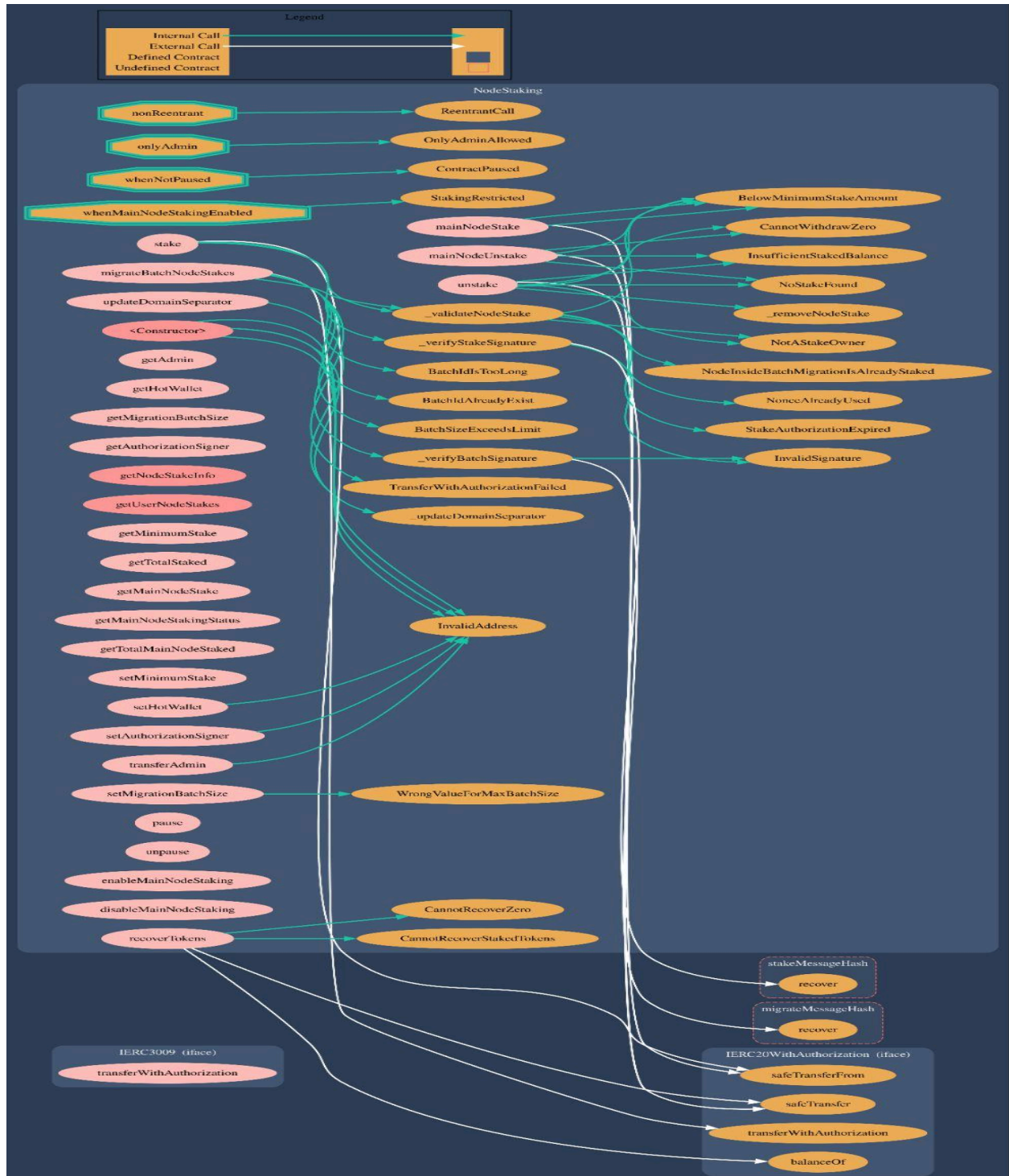
Findings:

- The logic for staking and unstaking was analyzed for correctness, security, and adherence to best practices.
- No vulnerabilities, inconsistencies, or deviations from expected staking behavior were identified.
- The flag-based control mechanism for `mainNodeStake` was verified as functioning as intended.



Mantisec Labs

Additional Details:



Concluding Remarks

To wrap it up, this audit has given us a good look at the contract's security and functionality.

Our auditors confirmed that all the issues are now resolved by the developers.