SECURITY AUDIT REPORT

Node Staking

DATE

29 Sep 2025

PREPARED BY

0xTeam.

WEB3 AUDITS

info@0xTeam.Space

0xTeamSpace

0xTeamSpace

Contents

Revision History & Version Control

Version	Date	Author's	Description
1.0	29 Sep 2025	Gurkirat, Manoj & Aditya	Initial Audit Report

0xTeam conducted a comprehensive Security Audit on the Node Staking to ensure the overall code quality, security, and correctness. The review focused on ensuring that the code functions as intended, identifying potential vulnerabilities, and safeguarding the integrity of Node Staking's operations against possible attacks.

Report Structure

The report is divided into two primary sections:

1. **Executive Summary** : Provides a high-level overview of the audit findings.
2. **Technical Analysis** : Offers a detailed examination of the Smart contracts code.

Note :
The analysis is static and manual, exclusively focused on the smart contract code. The information provided in this report should be used to assess the security, quality, and expected behavior of the code.

1.0 Disclaimer

This is a summary of our audit findings based on our analysis, following industry best practices as of the date of this report.However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. The audit focuses on Smart contracts coding practices and any issues found in the code, as detailed in this report. For a complete understanding of our analysis, you should read the full report. We have made every effort to conduct a thorough analysis, but it's important to note that you should not rely solely on this report and cannot make claims against us based on its contents. We strongly advise you to perform your own independent checks before making any decisions. Please read the disclaimer below for more information.

DISCLAIMER: By reading this report, you agree to the terms outlined in this disclaimer. If you do not agree, please stop reading immediately and delete any copies you have. This report is for informational purposes only and does not constitute investment advice. You should not rely on the report or its content, and 0xTeam and its affiliates (including all associated companies, employees, and representatives) are not responsible for any reliance on this report.The report is provided "as is" without any guarantees. 0xTeam excludes all warranties, conditions, or terms, including those implied by law, regarding quality, fitness for a purpose, and use of reasonable care. Except where prohibited by law, 0xTeam is not liable for any type of loss or damage, including direct, indirect, special, or consequential damages, arising from the use or inability to use this report. The findings are solely based on the Smart contracts code provided to us.

2.0 Executive Summary

2.1 Overview

0xTeam has meticulously audited the Node Staking Smart contracts project. The primary objective of this audit was to assess the security, functionality, and reliability of the Node Staking's before their deployment on the blockchain. The audit focused on identifying potential vulnerabilities, evaluating the contract's adherence to best practices, and providing recommendations to mitigate any identified risks. The comprehensive analysis conducted during this period ensures that the Node Staking is robust and secure, offering a reliable environment for its users.

2.2 Scope

The scope of this audit involved a thorough analysis of the Node Staking Smart contracts, focusing on evaluating its quality, rigorously assessing its security, and carefully verifying the correctness of the code to ensure it functions as intended without any vulnerabilities.

Files in Examination:

Language	Solidity
In-Scope	• contracts\BridgedERC721.sol • contracts\StudioChainNodeStaking.sol
Github	https://github.com/amgi-studios/Node-Staking
Fixed Commit Hash	980479f6268fa1af18cfbd9c8d485c8ceb4496c4

OUT-OF-SCOPE: External Smart contracts code, other imported code.

2.3 Audit Summary

Name	Verified	Audited	Vulnerabilities
Node Staking	Yes	Yes	Refer Section 5.0

2.4 Vulnerability Summary

High	Medium	Low	Informational
1	0	3	2

High Medium Low Informational

2.5 Recommendation Summary

Severity

Issues		High	Medium	Low	Informational	Total (Σ)
	Open	1	0	3	2	6
	Resolved					
	Acknowledged					
	Partially Resolved					
	Total (Σ)	1	0	3	2	**6**

- **Open**: Unresolved security vulnerabilities requiring resolution.
- **Resolved**: Previously identified vulnerabilities that have been fixed.
- **Acknowledged**: Identified vulnerabilities noted but not yet resolved.
- **Partially Resolved**: Risks mitigated but not fully resolved.

2.6 Summary of Findings

ID	Title	Severity	Fixed
H-01	False reward accrual when user adding to existing stake _addToExistingStake	High	
L-01	Missing input validation in constructor during daysInMonth	Low	
L-02	Missing input validation in setDaysInMonth function	Low	
L-03	Redundant _fillCompleteNodes function	Low	
I-01	Redundant zero-amount check in _calculateTotalLifetimeRewards	Info	
I-02	Inefficient and repeated state reads in initialActivation function	Info	

✔ - Fixed ● - Partially Fixed ✘ - Not Fixed 📝 - Acknowledged

3.0 Checked Vulnerabilities

We examined Smart contracts for widely recognized and specific vulnerabilities. Below are some of the common vulnerabilities considered.

Category	Check Items
Source Code Review	➔ Reentrancy Vulnerabilities ➔ Ownership Control ➔ Time-Based Dependencies ➔ Gas Usage in Loops ➔ Transaction Sequence Dependencies ➔ Style Guide Compliance ➔ EIP Standard Compliance ➔ External Call Verification ➔ Mathematical Checks ➔ Type Safety ➔ Visibility Settings ➔ Deployment Accuracy ➔ Repository Consistency
Functional Testing	➔ Business Logic Validation ➔ Feature Verification ➔ Access Control and Authorization ➔ Escrow Security ➔ Token Supply Management ➔ Asset Protection ➔ User Balance Integrity ➔ Data Reliability ➔ Emergency Shutdown Mechanism

4.0 Techniques , Methods & Tools Used

The following techniques, methods, and tools were used to review all the smart contracts

- **Structural Analysis:**
  This involves examining the overall design and architecture of the smart contract. We ensure that the contract is logically organised, scalable, and follows industry best practices. This step is crucial for identifying potential structural issues that could lead to vulnerabilities or maintenance challenges in the future.

- **Static Analysis:**
  Static analysis is conducted using automated tools to scan the contract's codebase for common vulnerabilities and security risks without executing the code. This process helps identify issues such as reentrancy, arithmetic errors, and potential denial-of-service (DOS) vulnerabilities early on, allowing for quick remediation.

- **Code Review / Manual Analysis:**
  A manual, in-depth review of the smart contract's code is performed to verify the logic and ensure it matches the intended functionality as described in the project's documentation. During this phase, we also confirm the findings from the static analysis and check for any additional issues that may not have been detected by automated tools.

- **Dynamic Analysis:**
  Dynamic analysis involves executing the smart contract in various controlled environments to observe its behaviour under different conditions. This step includes running comprehensive test cases, performing unit tests, and monitoring gas consumption to ensure the contract operates efficiently and securely in real-world scenarios.

- **Tools and Platforms Used for Audit:**
  Utilising tools such as Remix , Slither, Aderyn, Solhint for static analysis, and platforms like Hardhat and Foundry for dynamic testing and simulation.

Note: The following values for "Severity" mean:

- High: Direct and severe impact on the funds or the main functionality of the protocol.
- Medium: Indirect impact on the funds or the protocol's functionality.
- Low: Minimal impact on the funds or the protocol's main functionality.
- Informational: Suggestions related to good coding practices and gas efficiency.

5.0 Technical Analysis

High

[H-01] False reward accrual when user is adding to existing stake _addToExistingStake

Severity
High

Location	Functions
Contracts\StudioChainNodeStaking.sol	➔ `_updateRewardTimingAndCalculate(uint256 nodeId)` ➔ `calculateStakingRewards(address user, uint256 nodeId)`

Issue Description
When an existing staker who hasn't claimed rewards even once adds new funds to their stake using stakeInNode(uint256 nodeId) or initialActivation(), _addToExistingStake simply increases stake.amount without calculating the rewards of the initial stake. As a result, reward calculations use the original lastRewardCalculation together with the new (increased) stake.amount, causing newly added funds to earn rewards even though they have staked for less time. This allows a user to stake a tiny amount initially, wait, then add a large amount when he needs to claim or unstake, then claim rewards as if the large amount was present from the start.

Impact / Proof of Concept
- Can be exploited by small initial stake + large later stake → rewards computed on large total for the entire period.
- Affects both claim and unstake flows since both compute rewards from lastRewardCalculation and stake.amount.

- Below is the POC where we have explained that: When user1 and user2 stake 1000000 and 2500000 respectively initially and then user1 without claiming

any rewards stakes again for 1500000 after 11 months then after warping time by 1 more month (so rewards gets accured) both the users get the same rewards even user2 locked the more amount for longer period.

```
describe.only("Bug Unit Test", function (){
  it("Should calulate rewards according to the bug", async function () {
    // User1: stake small at the beginning
    await contract.connect(user1).initialActivation({
        value: ethers.utils.parseEther("1000000"), // small initial stake
    });

    // User2: stake full large amount at the beginning
    await contract.connect(user2).initialActivation({
        value: ethers.utils.parseEther("2500000"), // large upfront stake
    });

    const intialStakedValueOfUser1 = await contract.stakes(user1.address, 1);
    const intialStakedValueOfUser2 = await contract.stakes(user2.address, 1);

    console.log("Initial Staked value of user1", ethers.utils.formatEther(intialStakedValueOfUser1.amount));
    console.log("Initial Staked value of user2", ethers.utils.formatEther(intialStakedValueOfUser2.amount));

    console.log("⏳ Moving forward 11 months...");
    await increaseTime(86400 * 30 * 11);

    // User1: now stakes big amount late
    await contract.connect(user1).initialActivation({
      value: ethers.utils.parseEther("1500000"), // large late stake
    });

    console.log("⏳ Moving forward 1 more month...");
    await increaseTime(86400 * 30);

    // Calculate rewards for both
    const rewardsUser1 = await contract.calculateStakingRewards(user1.address, 1);
    const rewardsUser2 = await contract.calculateStakingRewards(user2.address, 1);

    console.log("This is global APY", (await contract.getGlobalAPY()));

    const currentTotalActivedNode = (await contract.getActiveNodesCount());

    console.log("Total activated nodes", Number(currentTotalActivedNode));

    console.log("-----------------------------------------------------");
    const finalStakedValueOfUser1 = await contract.stakes(user1.address, 1);
    const finalStakedValueOfUser2 = await contract.stakes(user2.address, 1);

    console.log("Final Staked value of user1", ethers.utils.formatEther(finalStakedValueOfUser1.amount));
    console.log("Final Staked value of user2", ethers.utils.formatEther(finalStakedValueOfUser2.amount));

    console.log("User1 Rewards (10 early + 15 late):", ethers.utils.formatEther(rewardsUser1));
    console.log("User2 Rewards (25 early only):   ", ethers.utils.formatEther(rewardsUser2));
    console.log("-----------------------------------------------------");

    // Bug: User1's rewards should be much lower than User2's,
    expect(rewardsUser1).to.be.closeTo(rewardsUser2, ethers.utils.parseEther("0.001"));
  });
```

```
it("claimStakingRewards: user1 (small early + large late) gets nearly same payout as user2 (large early) — demonstrates bug", async function () {
  // Setup: user1 small initial, user2 large initial
  await contract.connect(user1).initialActivation({ value: ethers.utils.parseEther("1000000") });
  await contract.connect(user2).initialActivation({ value: ethers.utils.parseEther("2500000") });

  // Advance ~11 months
  await increaseTime(86400 * 30 * 11);

  // user1 adds large amount late to fill the node
  await contract.connect(user1).initialActivation({ value: ethers.utils.parseEther("1500000") });

  // Advance 1 month so rewards accrue
  await increaseTime(86400 * 30);

  // Check globalAPY and sanity
  console.log("Global APY:", (await contract.getGlobalAPY()).toString());
  console.log("Active nodes:", (await contract.getActiveNodesCount()).toString());

  const initialBalanceBeforeClaimingOfUser1 = await  ethers.provider.getBalance(user1.address);
  const initialBalanceBeforeClaimingOfUser2 = await  ethers.provider.getBalance(user2.address);
  console.log("User 1 Initial wallet balance before claiming rewards", ethers.utils.formatEther(initialBalanceBeforeClaimingOfUser1));
  console.log("User 2 Initial wallet balance before claiming rewards", ethers.utils.formatEther(initialBalanceBeforeClaimingOfUser2));

  // Measure actual ETH received by calling claimStakingRewards (accounting for gas)
  await contract.connect(user1).claimStakingRewards(1);
  await contract.connect(user2).claimStakingRewards(1);


  const finalBalanceBeforeClaimingOfUser1 = await  ethers.provider.getBalance(user1.address);
  const finalBalanceBeforeClaimingOfUser2 = await  ethers.provider.getBalance(user2.address);
  console.log("User 1 Initial wallet balance before claiming rewards", ethers.utils.formatEther(finalBalanceBeforeClaimingOfUser1));
  console.log("User 2 Initial wallet balance before claiming rewards", ethers.utils.formatEther(finalBalanceBeforeClaimingOfUser2));

  const claimedAmtUser1 = finalBalanceBeforeClaimingOfUser1.sub(initialBalanceBeforeClaimingOfUser1);
  const claimedAmtUser2 = finalBalanceBeforeClaimingOfUser2.sub(initialBalanceBeforeClaimingOfUser2);

  console.log("claimed Value 1 ", ethers.utils.formatEther(claimedAmtUser1))
  console.log("claimed Value 2 ", ethers.utils.formatEther(claimedAmtUser2))

  expect(claimedAmtUser1).to.be.closeTo(claimedAmtUser2, ethers.utils.parseEther("0.001"));

});

});
```

```
gurkiratsingh@Gurkirats-MacBook-Air Node-Staking-main % npx hardhat test '/Users/gurkiratsingh/Desktop/Audit Codebases/Node-Staking-main/test/StudioChainStaking.test.js'


  StudioChainNodeStaking
    Bug Unit Test
Initial Staked value of user1 1000000.0
Initial Staked value of user2 2500000.0
⏳ Moving forward 11 months...
⏳ Moving forward 1 more month...
This is global APY BigNumber { value: "100" }
Total activated nodes 2
--------------------------------------------------
Final Staked value of user1 2500000.0
Final Staked value of user2 2500000.0
User1 Rewards (10 early + 15 late): 24657.535832064941653982
User2 Rewards (25 early only):    24657.535039320142059868
--------------------------------------------------
      ✔ Should calulate rewards according to the bug
Global APY: 100
Active nodes: 2
User 1 Initial wallet balance before claiming rewards 197499999.999541603195140657
User 2 Initial wallet balance before claiming rewards 197499999.999801947518928875
User 1 Initial wallet balance before claiming rewards 197524657.536099611173192246
User 2 Initial wallet balance before claiming rewards 197524657.53636061848895988
claimed Value 1  24657.536558007978051589
claimed Value 2  24657.536558670970031005
      ✔ claimStakingRewards: user1 (small early + large late) gets nearly same payout as user2 (large early) — demonstrates bug


  2 passing (494ms)
```

Recommendation

- Accrue pending rewards for the existing amount up to now(latest block.timestamp) and store them.
- Reset lastRewardCalculation = block.timestamp
- Liquity V2 has solved the similar problem in there codebase please go through it once "https://github.com/liquity/V2-gov/blob/main/src/Governance.sol"
- This bug will require some changes in the codebase so protocol could fix the issue accordingly.

Status - Open

Low

[L-01] Missing input validation in constructor during daysInMonth

Severity
LOW

Location	Functions/Variables
Contracts\StudioChainNodeStaking.sol	➔ `constructor(address _admin, uint256[12] memory _monthDays)`

Issue Description
The contract constructor accepts a uint256[12] array _monthDays to set the number of days in each month. Currently, the constructor does not validate the values of _monthDays beyond assigning them to daysInMonth[i].

```
constructor(address _admin, uint256[12] memory _monthDays) {

        currentMonth = 0;

        monthStartTime = block.timestamp;

        if (_admin == address(0)) revert InvalidAdmin();

        _grantRole(DEFAULT_ADMIN_ROLE, _admin);

        _grantRole(OPERATOR_ROLE, _admin);

        maxNodeId = 20;

        // Set days in each month from constructor argument

       // @audit CONFIRMED -- LOW shouldn't we add a check to see if the days that are
passed in array should be greater than 28 and less than = 31

        for (uint256 i = 0; i < 12; ++i) {

            daysInMonth[i] = _monthDays[i];

        }
```

Impact / Proof of Concept

- Only affects initialization correctness.
- A caller could pass invalid values (e.g.values > 0, values < 28, or values > 28) for months,which could break the logic in _calculateStakingDuration function as it is dependent on the daysInMonth state variable.

Recommendation

Validate the _monthDays array during construction add a require or if statement to check "_monthDays[i] >= 28 && _monthDays[i] <= 31" then it should revert.

Status - Opened

[L-02] Missing input validation in setDaysInMonth function

Severity
LOW

Location	Functions/Variables
Contracts\StudioChainNodeStaking.sol	➔ `setDaysInMonth(uint256 month, uint256 _days)`

Issue Description
Month index is unchecked:
The month argument can be any uint256. The function should validate that month is within [0, 11]. Without validation, invalid month indices may be stored, which can lead to misconfiguration or inconsistent calculations.

Insufficient days validation:
The current check allows _days values less than 28, which are invalid for real-world months. Functions relying on daysInMonth (e.g., _calculateStakingDuration) may calculate incorrect staking durations or rewards.

Impact / Proof of Concept
Incorrect month and day lengths can cause staking duration miscalculations in the function dependent on this in the contract.

Recommendation
- Add validation for both month and days:

```
function setDaysInMonth(uint256 month, uint256 _days) external
onlyRole(OPERATOR_ROLE) {

        // Validate month index (0-11)

        if (month > 11) revert InvalidMonth(month);



        // Validate realistic days (28-31)

        if (_days < 28 || _days > 31) revert InvalidDayCount(_days);
```

```
        daysInMonth[month] = _days;


        emit DaysInMonthUpdated(month, _days);
    }
```

Status - Opened

[L-03] Redundant _fillCompleteNodes function

Severity
LOW

Location	Functions/Variables
Contracts\StudioChainNodeStaking.sol	➔ `_fillCompleteNodes`

Issue Description
The _fillCompleteNodes function is defined to process complete node fills when staking an amount larger than FULL_NODE_THRESHOLD. However, this logic is already implemented directly inside the initialActivation function.

Despite being defined, _fillCompleteNodes is not called anywhere in the contract. As a result, it serves no functional purpose, duplicates existing logic, and increases contract bytecode size.

Impact / Proof of Concept
- No functional impact, since the function is never called.
- Slightly higher deployment cost due to unused bytecode.

Recommendation
- Remove the _fillCompleteNodes function if it is not intended to be used anywhere in the system.

Status - Opened

Info

[I-01] Redundant zero-amount check in _calculateTotalLifetimeRewards

Severity
INFO

Location	Functions/Variables
Contracts\StudioChainNodeStaking.sol	➔ `_calculateTotalLifetimeRewards`

Issue Description
The function _calculateTotalLifetimeRewards includes a defensive check:

```
if (stake.amount == 0) return 0;
```

However, this function is only invoked by unstake, which already enforces:

```
if (stake.amount == 0) revert NoActiveStake(nodeId);
```

Thus, the if (stake.amount == 0) check in _calculateTotalLifetimeRewards is unreachable and redundant

Impact / Proof of Concept
- No functional risk.
- Slightly increases bytecode size and gas consumption during deployment.

Recommendation
- Remove the redundant check from _calculateTotalLifetimeRewards to streamline logic

Status - Opened

[I-02] Inefficient and repeated state reads in initialActivation function

Severity
LOW

Location	Functions/Variables
Contracts\StudioChainNodeStaking.sol	➔ `initialActivation()`

Issue Description
The initialActivation function allows users to stake amount into a specific node. It performs multiple operations, including checking if the current node can accommodate the stake, potentially filling multiple nodes, and updating state variables.
issues were identified:
Repeated state reads for nodes[activeNodesCount].totalStaked

```
if (nodes[activeNodesCount].totalStaked + msg.value >= FULL_NODE_THRESHOLD) { ... }`
and`uint256 firstNodeAmount = FULL_NODE_THRESHOLD - nodes[activeNodesCount].totalStaked;
```

Impact / Proof of Concept
No direct impact on the smart contract just Gas inefficiency

Recommendation
- Cache repeated storage reads in local variables:

```
uint256 currentNodeTotalStaked = nodes[activeNodesCount].totalStaked;
uint256 fullNodeThreshold = FULL_NODE_THRESHOLD;

if (currentNodeTotalStaked + msg.value >= fullNodeThreshold) {
    uint256 firstNodeAmount = fullNodeThreshold -
currentNodeTotalStaked;
    ...
}
```

Status - Opened

6.0 Auditing Approach and Methodologies Applied

The Solidity smart contract was audited using a comprehensive approach to ensure the highest level of security and reliability. Careful attention was given to the following key areas to ensure the overall quality of the code:

- **Code quality and structure**: We conducted a detailed review of the codebase to identify any potential issues related to code structure, readability, and maintainability. This included analysing the overall architecture of the Solidity smart contract and reviewing the code to ensure it follows best practices and coding standards.
- **Security vulnerabilities**: Our team used manual techniques to identify any potential security vulnerabilities that could be exploited by attackers. This involved a thorough analysis of the code to identify any potential weaknesses, such as buffer overflows, injection vulnerabilities, signatures, and deprecated functions.
- **Documentation and comments**: Our team reviewed the code documentation and comments to ensure they accurately describe the code's intended behaviour and logic. This helps developers to better understand the codebase and make modifications without introducing new issues.
- **Compliance with best practices**: We checked that the code follows best practices and coding standards that are recommended by the Solidity community and industry experts. This ensures that the Solidity smart contract is secure, reliable, and efficient.

Our audit team followed OWASP and Ethereum (Solidity) community security guidelines for this audit. As a result, we were able to identify potential issues and provide recommendations to improve the smart contract's security and performance.

Throughout the audit of the smart contracts, our team placed great emphasis on ensuring the overall quality of the code and the use of industry best practices. We meticulously reviewed the codebase to ensure that it was thoroughly documented and that all comments and logic aligned with the intended behaviour. Our approach to the audit was comprehensive, methodical, and aimed at ensuring that the smart contract was secure, reliable, and optimised for performance.

6.1 Code Review / Manual Analysis

Our team conducted a manual analysis of the Solidity smart contracts to identify new vulnerabilities or to verify vulnerabilities found during static and manual analysis. We carefully analysed every line of code and made sure that all instructions provided during the onboarding phase were followed. Through our manual analysis, we were able to identify potential vulnerabilities that may have been missed by automated tools and ensure that the smart contract was secure and reliable.

6.2 Tools Used for Audit

In the course of our audit, we leveraged a suite of tools to bolster the security and performance of our program. While our team drew on their expertise and industry best practices, we also integrated various tools into our development environment. Noteworthy among them are Remix,Slither, Aderyn, Solhint for Static Analysis and Hardhat & Foundry for Dynamic Analysis. This holistic approach ensures a thorough analysis, uncovering potential issues that automated tools alone might overlook. 0xTeam takes pride in utilising these tools, which significantly contribute to the quality, security, and maintainability of our codebase

7.0 Limitations on Disclosure and Use of this Report

This report contains information concerning potential details of the Node Staking Project and methods for exploiting them. 0xTeam recommends that special precautions be taken to protect the confidentiality of both this document and the information contained herein. Security Assessment is an uncertain process, based on past experiences, currently available information, and known threats. All information security systems, which by their nature are dependent on human beings, are vulnerable to some degree. Therefore, while 0xTeam considers the major security vulnerabilities of the analysed systems to have been identified, there can be no assurance that any exercise of this nature will identify all possible vulnerabilities or propose exhaustive and operationally viable recommendations to mitigate those exposures. In addition, the analysis set forth herein is based on the technologies and known threats as of the date of this report. As technologies and risks change over time, the vulnerabilities associated with the operation of the Node Staking Smart contracts Code Base described in this report, as well as the actions necessary to reduce the exposure to such vulnerabilities, will also change. 0xTeam makes no undertaking to supplement or update this report based on changed circumstances or facts of which 0xTeam becomes aware after the date hereof, absent a specific written agreement to perform the supplemental or updated analysis. This report may recommend that 0xTeam use certain software or hardware products manufactured or maintained by other vendors. 0xTeam bases these recommendations upon its prior experience with the capabilities of those products. Nonetheless, 0xTeam does not and cannot warrant that a particular product will work as advertised by the vendor, nor that it will operate in the manner intended. This report was prepared by 0xTeam for the exclusive benefit of Node Staking and is proprietary information. The Non-Disclosure Agreement (NDA) in effect between 0xTeam and Node Staking governs the disclosure of this report to all other parties including product vendors and suppliers.