BEOSIN
Blockchain Security

# NodeDAO Protocol

Smart Contract Security Audit

V1.1

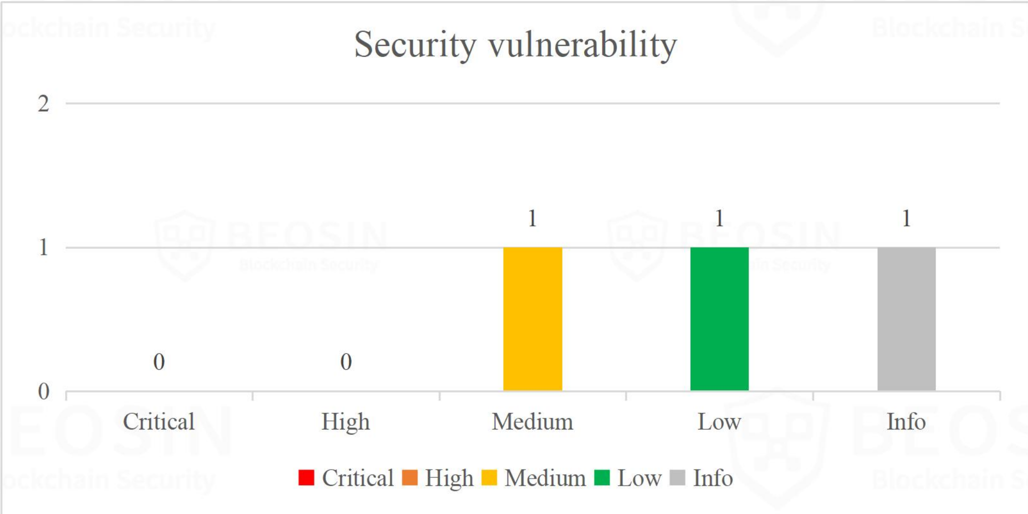
No. 202302161759

Feb 16th, 2023

# Contents

# Summary of Audit Results

**After auditing, 1 Medium-risk, 1 Low-risk and 1 Info-risk items were identified in the NodeDAO Protocol project.** Specific audit details will be presented in the **Findings** section. Users should pay attention to the following aspects when interacting with this project:




*Notes:

- **Risk Description:**

1. Since the project is deployed in proxy mode, the audit is only for the implementation of the contract audit, the audit code details can be found in the following project fact sheet.

- **Project Description:**

1. **Business overview**

NodeDAO Protocol is a smart contract of liquid staking derivatives. This audit includes: Oracle module, Registry module, Vault module, TimelockController module and staking module. There are two staking modes. The first staking mode is for users who stake less than 32 ETH. Users will get the corresponding NETH tokens and pay the corresponding fee when staking. The second staking model is for users who stake more than 32 ETH and they will receive the corresponding NFT tokens without paying any fee when staking. Users can enjoy the rewards by staking ETH to the contract and operator sends these amount to ETH 2.0. TimelockController module can permit controllers to delay the execution of transactions. The Vault module is used to receive and settle user rewards. The Registry module provides a request to register an operator. Oracles module is used to obtain the latest data of the Beacon chain: the number of validators and ETH balance. The project administrator can add an address to Oracle Member, and Oracle Member can submit the latest data of the Beacon chain every day (the default is submitted once a day), when oracle member submits the same data more than 2/3, the latest data of beacon chain will be updated in this contract.

This is the second audit version of NodeDAO Protocol, which is from commit hash 356941569ff5e29763e5c639c5cf914a102fe437 to 0c98571bd2ab009d96e3ba5bc91dbe5f93f37031. More details of first audit version please see : https://beosin.com/audits/NodeDAO-Protocol_202302011759.pdf.

# 1 Overview

## 1.1 Project Overview

| | |
|---|---|
| **Project Name** | NodeDAO Protocol |
| **Platform** | Ethereum |
| **Audit scope** | https://github.com/King-Hash-Org/NodeDAO-Protocol |
| **Commit Hash** | 356941569ff5e29763e5c639c5cf914a102fe437(Initial)<br>cdc94e75bb3f4bf367de19c76979b01999f8f448<br>0c98571bd2ab009d96e3ba5bc91dbe5f93f37031(Finally) |

## 1.2 Audit Overview

Audit work duration: Jan 13, 2023 – Feb 16, 2023

Audit methods: Formal Verification, Static Analysis, Typical Case Testing and Manual Review.

Audit team: Beosin Security Team.

# 2 Findings

| Index | Risk description | Severity level | Status |
|---|---|---|---|
| NodeOperatorRegistry-1 | Related functions do not verify operator quit condition | Medium | Fixed |
| NodeOperatorRegistry-2 | Slash function is not called | Info | Fixed |
| LiquidStaking-1 | Centralization risk | Low | Acknowledged |

**Status Notes:**

LiquidStaking-1 is unfixed and will has centralization risk.

# Finding Details:

## [NodeOperatorRegistry-1]Related functions do not verify operator quit condition

| | |
|---|---|
| **Severity Level** | Medium |
| **Type** | Business Security |
| **Lines** | src\registries\NodeOperatorRegistry.sol #L481-515 |
| **Description** | In the NodeOperatorRegistry contract, the quit operator is still trusted. In *isTrustedOperator* and *isTrustedOperatorOfControllerAddress* functions do not have the relevant check condition for operator quit, which will cause the *isTrustedOperatorOfControllerAddress* function to bypass the verification and continue to receive rewards and update operations in the LiquidStaking contract. |

```
478      * @notice Returns whether an operator is trusted
479      * @param _id operator id
480      */
481     function isTrustedOperator(uint256 _id) external view operatorExists(_id) returns (bool) {
482         if (blacklistOperators[_id]) {
483             return false;
484         }
485 
486         if (permissionlessBlockNumber != 0 && block.number >= permissionlessBlockNumber) {
487             return true;
488         }
489 
490         return operators[_id].trusted;
491     }
492 
```

Figure 1 Source code of *isTrustedOperator* function (Unfixed)

```
502      * @notice Returns whether an operator is trusted
503      * @param _controllerAddress controller address
504      */
505     function isTrustedOperatorOfControllerAddress(address _controllerAddress) external view returns (uint256) {
506         uint256 _id = controllerAddress[_controllerAddress];
507         if (blacklistOperators[_id]) {
508             return 0;
509         }
510 
511         if (permissionlessBlockNumber != 0 && block.number >= permissionlessBlockNumber) {
512             return _id;
513         }
514 
515         return trustedControllerAddress[_controllerAddress];
516     }
```

Figure 2 Source code of isTrustedOperatorOfControllerAddress function (Unfixed)

```
function registerValidator(
    bytes[] calldata _pubkeys,
    bytes[] calldata _signatures,
    bytes32[] calldata _depositDataRoots
) external nonReentrant whenNotPaused {
    require(
        _pubkeys.length == _signatures.length && _pubkeys.length == _depositDataRoots.length,
        "parameter must have the same length"
    );

    // must be a trusted operator
    uint256 operatorId = nodeOperatorRegistryContract.isTrustedOperatorOfControllerAddress(msg.sender);
    require(operatorId != 0, "The sender must be controlAddress of the trusted operator");
    require(operatorPoolBalances[operatorId] / DEPOSIT_SIZE >= _pubkeys.length, "Insufficient balance");

    reinvestRewardsOfOperator(operatorId);

    for (uint256 i = 0; i < _pubkeys.length; ++i) {
        _stakeAndMint(operatorId, _pubkeys[i], _signatures[i], _depositDataRoots[i]);
    }

    uint256 stakeAmount = DEPOSIT_SIZE * _pubkeys.length;
    operatorPoolBalances[operatorId] -= stakeAmount;
    operatorPoolBalancesSum -= stakeAmount;
    beaconOracleContract.addPendingBalances(stakeAmount);
}
```

Figure 3 Source code of *registerValidator* function

| | |
|---|---|
| **Recommendations** | It is recommended to add the verify condition if the operator is quit in the relevant function. |
| **Status** | Fixed. |

```
 * @notice Returns whether an operator is trusted
 * @param _id operator id
 */
function isTrustedOperator(uint256 _id) external view operatorExists(_id) returns (bool) {
    if (blacklistOperators[_id]) {
        return false;
    }

    if (operators[_id].isQuit) {
        return false;
    }

    if (permissionlessBlockNumber != 0 && block.number >= permissionlessBlockNumber) {
        return true;
    }

    return operators[_id].trusted;
}
```

Figure 4 Source code of *isTrustedOperator* function (fixed)

```
function isTrustedOperatorOfControllerAddress(address _controllerAddress) external view returns (uint256) {
    uint256 _id = controllerAddress[_controllerAddress];
    if (blacklistOperators[_id]) {
        return 0;
    }

    if (operators[_id].isQuit) {
        return 0;
    }

    if (permissionlessBlockNumber != 0 && block.number >= permissionlessBlockNumber) {
        return _id;
    }

    return trustedControllerAddress[_controllerAddress];
}
```

Figure 5 Source code of *isTrustedOperatorOfControllerAddress* function (fixed)

## [NodeOperatorRegistry-2] Slash function is not called

| | |
|---|---|
| **Severity Level** | Info |
| **Type** | Business Security |
| **Lines** | src\registries\NodeOperatorRegistry.sol #L536-540 |
| **Description** | In the NodeOperatorRegistry contract, the *Slash* function can only be called by the LiquidStaking contract, but there is no *Slash* function called in LiquidStaking contract. |

```
531  /**
532   * @notice When a validator run by an operator goes seriously offline, it will be slashed
533   * @param _operatorId operator id
534   * @param _amount slash amount
535   */
536  function slash(uint256 _amount, uint256 _operatorId) external nonReentrant onlyLiquidStaking {
537      require(operatorPledgeVaultBalances[_operatorId] >= _amount, "Insufficient funds");
538      operatorPledgeVaultBalances[_operatorId] -= _amount;
539      liquidStakingContract.slashReceive{value: _amount}(_amount);
540      emit Slashed(_amount, _operatorId);
541  }
542
```

Figure 6 Source code of *Slash* function

| | |
|---|---|
| **Recommendations** | It is recommended to add relevant code or delete the slash function. |
| **Status** | Fixed. |

```
/**
 * @notice slash operator
 * @param _operatorId operator id
 * @param _amount slash amount
 */
function slashOperator(uint256 _operatorId, uint256 _amount) external onlyOwner {
    nodeOperatorRegistryContract.slash(_amount, _operatorId);
    operatorPoolBalances[_operatorId] += _amount;

    emit OperatorSlashed(_operatorId, _amount);
}
```

Figure 7 Source code of *slashOperator* function

## [LiquidStaking-1] Centralization risk

| | |
|---|---|
| **Severity Level** | Low |
| **Type** | Business Security |
| **Lines** | src\LiquidStaking.sol |
| **Description** | onlyOwner has a certain centralization risk. Figure 8 can modify the dao address, the onlyDao modifier can modify some key contract address and parameters. In figure 9 and figure 10, onlyOwner modifier can assign operatorIds and amounts of operatorPoolBalances. |

```
    /**
     * @notice set dao address
     * @param _dao new dao address
     */
    function setDaoAddress(address _dao) external onlyOwner {
        require(_dao != address(0), "Dao address invalid");
        emit DaoAddressChanged(dao, _dao);
        dao = _dao;
    }
```

Figure 8 Source code of *setDaoAddress* function

```
    function assignBlacklistOrQuitOperator(
        uint256 _assignOperatorId,
        uint256[] calldata _operatorIds,
        uint256[] calldata _amounts
    ) external onlyOwner {
        // assignOperatorId must be a blacklist operator
        require(
            !nodeOperatorRegistryContract.isTrustedOperator(_assignOperatorId)
                || nodeOperatorRegistryContract.isQuitOperator(_assignOperatorId),
            "This operator is trusted"
        );
        require(_operatorIds.length == _amounts.length, "Invalid length");

        // Update operator available funds
        uint256 totalAmount = 0;
        for (uint256 i = 0; i < _operatorIds.length; ++i) {
            uint256 operatorId = _operatorIds[i];
            uint256 amount = _amounts[i];
            totalAmount += amount;
            operatorPoolBalances[operatorId] += amount;
        }
```

Figure 9 Source code of *assignBlacklistOrQuitOperator* function

```
    /**
     * @notice slash operator
     * @param _operatorId operator id
     * @param _amount slash amount
     */
    function slashOperator(uint256 _operatorId, uint256 _amount) external onlyOwner {
        nodeOperatorRegistryContract.slash(_amount, _operatorId);
        operatorPoolBalances[_operatorId] += _amount;

        emit OperatorSlashed(_operatorId, _amount);
    }
```

Figure 10 Source code of *slashOperator* function

| | |
|---|---|
| **Recommendations** | It is recommended to modify the management authority to DAO. |
| **Status** | Acknowledged. Project team respond that all owner will transfer ownership to the time lock contract and the DAO will control the time lock. |

# Appendix

## 3.1 Vulnerability Assessment Metrics and Status in Smart Contracts

### 3.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1 (Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

| Impact / Likelihood | Severe | High | Medium | Low |
|---|---|---|---|---|
| Probable | Critical | High | Medium | Low |
| Possible | High | High | Medium | Low |
| Unlikely | Medium | Medium | Low | Info |
| Rare | Low | Low | Info | Info |

### 3.1.2 Degree of impact

- **Severe**

Severe impact generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

### 3.1.4 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

### 3.1.5 Fix Results Status

| **Status** | **Description** |
| --- | --- |
| **Fixed** | The project party fully fixes a vulnerability. |
| **Partially Fixed** | The project party did not fully fix the issue, but only mitigated the issue. |
| **Acknowledged** | The project party confirms and chooses to ignore the issue. |

## 3.2 Audit Categories

| No. | Categories | Subitems |
|---|---|---|
| 1 | Coding Conventions | Compiler Version Security |
| | | Deprecated Items |
| | | Redundant Code |
| | | require/assert Usage |
| | | Gas Consumption |
| 2 | General Vulnerability | Integer Overflow/Underflow |
| | | Reentrancy |
| | | Pseudo-random Number Generator (PRNG) |
| | | Transaction-Ordering Dependence |
| | | DoS (Denial of Service) |
| | | Function Call Permissions |
| | | call/delegatecall Security |
| | | Returned Value Security |
| | | tx.origin Usage |
| | | Replay Attack |
| | | Overriding Variables |
| | | Third-party Protocol Interface Consistency |
| 3 | Business Security | Business Logics |
| | | Business Implementations |
| | | Manipulable Token Price |
| | | Centralized Asset Control |
| | | Asset Tradability |
| | | Arbitrage Attack |

Beosin classified the security issues of smart contracts into three categories: Coding Conventions, General Vulnerability, Business Security. Their specific definitions are as follows:

- **Coding Conventions**

Audit whether smart contracts follow recommended language security coding practices. For example, smart contracts developed in Solidity language should fix the compiler version and do not use deprecated keywords.

- **General Vulnerability**

General Vulnerability include some common vulnerabilities that may appear in smart contract projects. These vulnerabilities are mainly related to the characteristics of the smart contract itself, such as integer overflow/underflow and denial of service attacks.

- **Business Security**

Business security is mainly related to some issues related to the business realized by each project, and has a relatively strong pertinence. For example, whether the lock-up plan in the code match the white paper, or the flash loan attack caused by the incorrect setting of the price acquisition oracle.

*Note that the project may suffer stake losses due to the integrated third-party protocol. This is not something Beosin can control. Business security requires the participation of the project party. The project party and users need to stay vigilant at all times.

## 3.3 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in blockchain.

## 3.4 About Beosin

Beosin is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions. Beosin has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, Beosin has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

**Official Website**

https://www.beosin.com

**Telegram**

https://t.me/+dD8Bnqd133RmNWNl

**Twitter**

https://twitter.com/Beosin_com

**Email**

Contact@beosin.com