Cyberscope

Audit Report

MASTERNODED

July 2024

SHA256 16565fa3689cedefb1b75bdad74450689d647c4a1b2443cf15bafb4b6de44a41

Audited by © cyberscope

Table of Contents

Review

Testing Deploy	https://testnet.bscscan.com/address/0xea4d221138e3ef4b536c930b13a5251f7661ea55

Audit Updates

Initial Audit	09 Jul 2024
Corrected Phase 2	24 Jul 2024

Source Files

Filename	SHA256
contracts/NodedStaking.sol	16565fa3689cedefb1b75bdad74450689d647c4a1b2443cf15bafb4b6de44a41

Overview

The contract, designed as a pool management system, interacts primarily with Balancer pools and manages various functionalities related to staking, unstaking, and pool administration. This smart contract leverages the Balancer protocol to enable users to stake assets into pools and earn returns based on predefined APRs and stake durations. The contract is also integrated with a specific token, referred to here as `nodedToken`, which adds an additional layer of functionality regarding interest calculations.

Scope Audit

The scope of the current audit is focused exclusively on the contract code provided. It is important to note that the functionality of this smart contract relies heavily on the correct and proper functioning of the Balancer contract, which falls outside the scope of this audit. As such, any operations or dependencies involving the Balancer contract should be carefully reviewed and validated independently to ensure that interactions with it are secure and function as intended. Users and stakeholders should be aware that any issues arising from the Balancer platform itself could directly impact the operations and security of the audited contract.

Stake Functionality

Users can stake assets into active pools specified by the `poolId`. During the staking process, users must provide asset arrays and corresponding amounts, which are then processed and transferred into the Balancer pool. The contract checks if the asset is a native currency (e.g., ETH) or a token, handling transfers accordingly. A `JoinPoolRequest` is created and executed, allowing assets to be pooled and managed within the Balancer ecosystem. This transaction records key stake details such as the amount, asset type, and the time when the stake started, which are crucial for calculating returns later.

Unstake Functionality

The unstake function allows users to withdraw their staked assets from the pool after the lockup period has ended. The function calculates the initial and new balances of assets, applies any specified fees, and then returns the assets minus fees to the user. If the pool contains the `nodedToken`, additional interest calculated based on the staked amount and the time staked is paid out in `nodedToken`. This function also manages the internal record-keeping by updating or deleting user stakes and pools as needed based on the remaining balances.

Stake Noded Functionality

The `stakeNoded` function in the contract allows users to stake their nodedTokens by specifying an amount and a lockup duration index. The function validates the input to ensure that the staked amount is greater than zero and that the lockup index is valid. Once validated, the function transfers the specified amount of nodedTokens from the user to the contract. The user's stake details, including the staked amount, start time, lockup duration, and the annual percentage rate (APR), are recorded in a struct and stored in the user's stake array. Additionally, the total amount of nodedTokens staked and the total number of users staking are updated accordingly. This process ensures that all staking transactions are securely handled and recorded for future reference.

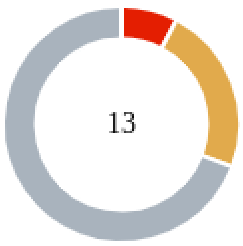
Unstake Noded Functionality

The `unstakeNoded` function enables users to withdraw their staked nodedTokens after the lockup period has ended. The function checks that the specified stake index is valid and that the lockup period has passed. It then calculates the interest earned on the staked amount based on the APR and lockup duration. After computing any applicable fees, the function transfers the staked amount plus interest minus the fee back to the user. The total amount of nodedTokens staked is updated to reflect the withdrawal. The function also removes the unstaked entry from the user's stake array, ensuring the internal record-keeping remains accurate. If the user has no remaining stakes, the total number of staking users is decremented, maintaining a precise count of active participants.

Owner Functionalities

The contract owner has extensive control over the management of pools. They can create new pools with specific parameters such as APR, fees, and active status. Owners can also update the APR for `nodedToken`, set and adjust pool-specific fees, and toggle the active status of a pool. Additionally, the owner has the power to delete pools from the contract. These functionalities emphasize the centralized control the owner possesses over the operational aspects of the contract, allowing for dynamic management of pool terms and conditions.

Findings Breakdown



●	Critical	1
●	Medium	3
●	Minor / Informative	9

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	0	1	0	0
●	Medium	0	3	0	0
●	Minor / Informative	9	0	0	0

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
🔴	IID	Inconsistent Interest Distribution	Acknowledged
🟠	ERDAP	Equivalent Rewards Distribution Across Pools	Acknowledged
🟠	UEC	Unchecked ETH Contribution	Acknowledged
🟠	UPAP	Unnecessary Parameter Arrays Passed	Acknowledged
⚪	CCR	Contract Centralization Risk	Unresolved
⚪	FRO	Function Reuse Optimization	Unresolved
⚪	ICBV	Inadequate Contract Balance Verification	Unresolved
⚪	MEE	Missing Events Emission	Unresolved
⚪	MU	Modifiers Usage	Unresolved
⚪	PBV	Percentage Boundaries Validation	Unresolved
⚪	PTAI	Potential Transfer Amount Inconsistency	Unresolved
⚪	TSI	Tokens Sufficiency Insurance	Unresolved
⚪	L04	Conformance to Solidity Naming Conventions	Unresolved

IID - Inconsistent Interest Distribution

Criticality	Critical
Location	contracts/NodedUpgradable.sol#L371,408
Status	Acknowledged

Description

The contract is designed to distribute the calculated `nodedInterest` from the `nodedToken` equally across the asset in the `unstake` function's assets array, without considering the specific amount or decimal usage of each individual asset. This method applies the same interest amount indiscriminately, which can lead to discrepancies in the distribution relative to the actual stake values and properties of each asset. As a result, the function does not accurately reflect the proportionate interests earned based on the different stake sizes and asset characteristics, potentially leading to financial inconsistencies and unfair distributions.

```
   function unstake(
        bytes32 poolId,
        uint256 stakeIndex,
        address[] memory assets,
        uint256[] memory amounts,
        bytes memory userData
    ) external nonReentrant {
        Stake storage stakex =
stakes[msg.sender][poolId][stakeIndex];
        require(block.timestamp >= stakex.startTime +
stakex.lockupDuration, "Lockup period is not over");
        require(stakex.amount > 0, "No amount found for this
stake");
        ...

        if (containsNoded(poolId)) {
            uint256 nodedInterest =
calculateProjectedReturns(stakex.amount, nodedApr,
stakex.lockupDuration);
            nodedToken.safeTransfer(msg.sender, nodedInterest);
        }
        ...
    }
```

Recommendation

It is recommended to adjust the interest distribution mechanism to account for the individual proportions and characteristics of each staked asset within the unstake function. Implementing a proportional distribution model, where the interest applied to each asset is scaled based on its specific stake amount and characteristics (such as decimals), will ensure a fair and accurate allocation of `nodedInterest`. This adjustment will enhance the precision of payouts and align the distribution with the underlying economic intentions of the staking model.

Team Update

The team has acknowledged that this is not a security issue and states: *If a pool contains noded token it will be distributed according to the percentage defined for noded token, it does not have to take into account all the other assets in the pool, which is defined in nodedApr.*

ERDAP - Equivalent Rewards Distribution Across Pools

Criticality	Medium
Location	contracts/NodedUpgradable.sol#L262
Status	Acknowledged

Description

The contract is currently structured to reward users with interest from staking only if they use a specific `poolId` associated with the noded token. This design decision results in a limitation where staking with any other `poolId` does not yield interest rewards, but users are still subject to fee deductions during the unstaking process. This imbalance could significantly reduce the incentive for users to participate in the staking process unless they are able to stake in the specific pool that contains the noded token. As a result, the contract may see lower overall participation and engagement from potential stakers.

```
if (containsNoded(poolId)) {
    uint256 nodedInterest =
calculateNodedInterest(stakex.amount, block.timestamp -
stakex.startTime);
    nodedToken.transfer(msg.sender, nodedInterest);
}
```

Recommendation

It is recommended to reconsider the application of interest during the unstaking process for all pools, not just those containing the noded token. Providing a more uniform interest distribution or reward system could enhance user engagement and fairness across different pools. Adjusting the reward structure to ensure all participants can receive some form of return, regardless of the pool in which they choose to stake, will likely encourage broader participation and contribute to the sustainability and attractiveness of the staking platform.

Team Update

The team has acknowledged that this is not a security issue and states: *If there is nodedToken in the pool, it will be calculated according to the nodedApr and sent separately, if not no matter how much time a user selects to stake for, it will be the same reward as the* *pool yields according to its dynamic apr, so how this works is, when user stakes, he gets a bptamount (balancer pool token), and when unstaking, this bptToken is converted into multiple assets after accumulating the interest (apr) this apr is decided by balancer itself and we just take the fees from it.*

UEC - Unchecked ETH Contribution

Criticality	Medium
Location	contracts/NodedUpgradable.sol#L253
Status	Acknowledged

Description

The contract is not adequately verifying the `msg.value` during the `stake` function when an `assetAddress` is specified that is not zero. This oversight allows users to potentially include ETH (native currency) as `msg.value` along with token transfers specified in the `assets` array. Consequently, the `joinPool` function processes both the specified token amount and the provided `msg.value`, which can lead to unexpected financial interactions in the pool, such as incorrect token balances or unintended liquidity provisions.

```
    function stake(
            bytes32 poolId,
            address[] calldata assets,
            uint256[] calldata amounts,
            uint256 lockupIndex,
            bytes calldata userData
        ) external payable nonReentrant {
            ...

            if (assetAddress == address(0)) {
                require(msg.value == assetAmount, "Native amount sent does not
match the required amount");
            } else {
                IERC20Upgradeable(assetAddress).transferFrom(msg.sender,
address(this), assetAmount);
            }

            JoinPoolRequest memory request = JoinPoolRequest({
                assets: assets,
                maxAmountsIn: amounts,
                userData: userData,
                fromInternalBalance: false
            });
            balancer.joinPool{ value: msg.value }(poolId, address(this),
address(this), request);
```

```
    ...
```

Recommendation

It is recommended to add an additional check in the `stake` function to ensure that if the `assets` provided are tokens (i.e., `assetAddress` is not zero), the `msg.value` should be zero. This precaution will prevent the inadvertent use of ETH in transactions intended only for token transfers, thereby aligning the stake contributions strictly with user intentions and the contract's requirements for asset management. This change will enhance security and clarity in transaction processing within the contract.

UPAP - Unnecessary Parameter Arrays Passed

Criticality	Medium
Location	contracts/NodedStaking.sol#L217
Status	Acknowledged

Description

The contract is currently designed to accept arrays of `assets` addresses and `amounts` when staking, despite the intended functionality being to stake only one token at a time. This design allows for multiple assets to be passed arbitrarily, which can lead to confusion and potential misuse. By requiring an array of addresses and amounts, the contract adds unnecessary complexity and the risk of errors. Instead, the contract should be designed to accept only a single asset address and amount as parameters, and then create the necessary arrays within the function to comply with the requirements of the Balancer contract. This approach would streamline the staking process and ensure that only one asset is staked at a time, aligning with the intended functionality.

```
   function stake(
        bytes32 poolId,
        address[] calldata assets,
        uint256[] calldata amounts,
        uint256 lockupIndex,
        bytes calldata userData
    ) external payable nonReentrant {
       ...

        address assetAddress = address(0);
        uint256 assetAmount = 0;

        for (uint256 i = 0; i < assets.length; i++) {
            if (amounts[i] > 0) {
                assetAddress = assets[i];
                assetAmount = amounts[i];
                break;
            }
        }

        JoinPoolRequest memory request = JoinPoolRequest({
            assets: assets,
            maxAmountsIn: amounts,
            userData: userData,
            fromInternalBalance: false
        });
        balancer.joinPool{ value: msg.value }(poolId,
address(this), address(this), request);
        ...
```

Recommendation

It is recommended to use single variables for the assets address and amount instead of arrays. The contract should then create the required arrays within the function to ensure that no more than one asset is passed arbitrarily to the Balancer contract. This change will simplify the function's interface, reduce the risk of errors, and ensure that the staking process is aligned with the intended design of staking only one token at a time.

Team Update

The team has acknowledged that this is not a security issue and states: *This is done because balancer wants the input that way, the functionality is handled by a UI so there is no chances is necessary.*

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L90,94,99,103,108,118,127,132
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

Specifically, the contract owner has the authority to change the address of the balancer contract, set the APR for node tokens (nodedApr), adjusting pool-specific fees and APRs, creating new pools, toggling the active status of pools, and even deleting pools. This centralized control extends to financial aspects that can directly affect the earnings and strategies of stakeholders involved in these pools. The ability to alter financial parameters and pool status at the discretion of a single entity can lead to potential biases, mismanagement, or abuse of power.

```
   function setNodedApr(uint256 _nodedApr) public onlyOwner {
        nodedApr = _nodedApr;
    }

    function setPoolFee(bytes32 poolId, uint256 feePercentage)
public onlyOwner {
        require(feePercentage <= 10000, "Fee percentage cannot
exceed 100%");
        pools[poolId].feePercentage = feePercentage;
    }

    function setPoolApr(bytes32 poolId, uint256 apr) public
onlyOwner {
        pools[poolId].apr = apr;
    }

    function getPoolDataFromBalancer(bytes32 poolId) public
view returns (address[] memory, uint256[] memory, uint256) {
        (address[] memory tokens, uint256[] memory balances,
uint256 lastChangeBlock) = balancer.getPoolTokens(poolId);
        return (tokens, balances, lastChangeBlock);
    }

    function createPool(
        ...

        pools[poolId] = Pool(poolId, assets, lockupDurations,
apr, poolToken, isActive, feePercentage);
        poolIds.push(poolId);
    }

    function togglePoolActive(bytes32 poolId, bool isActive)
public onlyOwner {
        require(poolId != bytes32(0), "Invalid poolId");
        pools[poolId].isActive = isActive;
    }

    function deletePool(bytes32 poolId) public onlyOwner {
        require(poolId != bytes32(0), "Invalid poolId");
        delete pools[poolId];
        for (uint256 i = 0; i < poolIds.length; i++) {
            if (poolIds[i] == poolId) {
                poolIds[i] = poolIds[poolIds.length - 1];
                poolIds.pop();
                break;
            }
        }
    }
   function setBalancerAddress(address _balancerAddress) public
onlyOwner {
        require(_balancerAddress != address(0), "Balancer
address cannot be zero");
        balancer = IBalancerPool(_balancerAddress);
    }
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

FRO - Function Reuse Optimization

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L435,449
Status	Unresolved

Description

The contract is currently implementing separate logic in the `calculateAPYForUser` function that essentially duplicates the functionality already available in the `calculateInterestForStake` function. Specifically, the computation of interest based on the amount staked, APR, and time staked is being recalculated in `calculateAPYForUser` for each stake, even though `calculateInterestForStake` is designed to perform this calculation efficiently for individual stakes. This redundancy not only leads to a larger and more complex codebase but also increases the potential for errors and reduces the maintainability of the contract.

```
   function calculateAPYForUser(address user, bytes32 poolId)
public view returns (uint256 totalInterest) {
        Stake[] memory userStakes = stakes[user][poolId];
        totalInterest = 0;
        for (uint256 i = 0; i < userStakes.length; i++) {
            uint256 timeStaked = block.timestamp -
userStakes[i].startTime;
            if (timeStaked > userStakes[i].lockupDuration) {
                timeStaked = userStakes[i].lockupDuration;
            }
            uint256 interest = userStakes[i].amount *
userStakes[i].apr * timeStaked / (365 days * 10000);
            totalInterest += interest;
        }
        return totalInterest;
    }

    function calculateInterestForStake(address user, bytes32
poolId, uint256 stakeIndex) public view returns (uint256
interest) {
        Stake memory stakex = stakes[user][poolId][stakeIndex];
        uint256 timeStaked = block.timestamp -
stakex.startTime;
        if (timeStaked > stakex.lockupDuration) {
            timeStaked = stakex.lockupDuration;
        }
        interest = stakex.amount * stakex.apr * timeStaked /
(365 days * 10000);
        return interest;
    }
```

Recommendation

It is recommended to utilize the `calculateInterestForStake` function within the `calculateAPYForUser` function to calculate the interest for each stake. By calling `calculateInterestForStake` directly, the contract can reduce code duplication, minimize potential bugs, and improve overall efficiency. This approach will streamline the code, making it easier to manage and audit while ensuring that all stake interest calculations are consistent across different functions.

ICBV - Inadequate Contract Balance Verification

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L228,255,385,399
Status	Unresolved

Description

The contract is designed to calculate the balances of pool tokens before and after joining or exiting a pool, but it fails to verify that the balance after these operations is indeed greater than the initial balance as expected. This oversight can lead to scenarios where, despite successful transaction indications, the actual token balances do not reflect anticipated increments due to potential discrepancies in the external pool's balance handling or errors in the execution logic. This gap in verification may expose users to financial discrepancies without immediate detection.

```
       uint256 initialBalance =
IERC20Upgradeable(pool.poolToken).balanceOf(address(this));
        ...
        balancer.joinPool{ value: msg.value }(poolId,
address(this), address(this), request);

        uint256 newBalance =
IERC20Upgradeable(pool.poolToken).balanceOf(address(this));
        ...

        uint256[] memory initialBalances = new
uint256[](assets.length);
        for (uint256 i = 0; i < assets.length; i++) {
            initialBalances[i] =
IERC20Upgradeable(assets[i]).balanceOf(address(this));
        }
       ...
        for (uint256 i = 0; i < assets.length; i++) {
            uint256 newBalance =
IERC20Upgradeable(assets[i]).balanceOf(address(this));
            uint256 difference = newBalance -
initialBalances[i];
            uint256 fee = difference *
pools[poolId].feePercentage / 10000;
            ...
```

Recommendation

It is recommended to add a check within the smart contract to ensure that the balance after the pool operations is greater than the initial balance. This validation should be implemented after each `joinPool` and `exitPool` operation to safeguard against unanticipated results, thereby enhancing the reliability and security of the transaction process. Such a measure would help guarantee that all operations have the intended effect on the contract's balance, providing a systematic approach to error detection and prevention.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L125,143,217,371
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
   function setNodedApr(uint256 _nodedApr) public onlyOwner {
        nodedApr = _nodedApr;
    }

    function setPoolFee(bytes32 poolId, uint256 feePercentage)
public onlyOwner {
        require(feePercentage <= 10000, "Fee percentage cannot
exceed 100%");
        pools[poolId].feePercentage = feePercentage;
    }

    function setPoolApr(bytes32 poolId, uint256 apr) public
onlyOwner {
        pools[poolId].apr = apr;
    }

    function createPool(
        bytes32 poolId,
        uint256[] memory lockupDurations,
        uint256 apr,
        bool isActive,
        uint256 feePercentage
    ) public onlyOwner {
    ...
    }

    function stake(
        bytes32 poolId,
        address[] calldata assets,
        uint256[] calldata amounts,
        uint256 lockupIndex,
        bytes calldata userData
    ) external payable nonReentrant {
    ...
    }

    function unstake(
        bytes32 poolId,
        uint256 stakeIndex,
        address[] memory assets,
        uint256[] memory amounts,
        bytes memory userData
    ) external nonReentrant {
    ...
    }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L174,179,197
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(poolId != bytes32(0), "Invalid poolId");
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

PBV - Percentage Boundaries Validation

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L125,129
Status	Unresolved

Description

The contract utilizes variables for percentage-based calculations that are required for its operations. These variables are involved in multiplication and division operations to determine proportions related to the contract's logic. If such variables are set to values beyond their logical or intended maximum limits, it could result in incorrect calculations. This misconfiguration has the potential to cause unintended behavior or financial discrepancies, affecting the contract's integrity and the accuracy of its calculations.

```
    function setNodedApr(uint256 _nodedApr) public onlyOwner {
        nodedApr = _nodedApr;
    }

     function setPoolApr(bytes32 poolId, uint256 apr) public
onlyOwner {
        pools[poolId].apr = apr;
    }
```

Recommendation

To mitigate risks associated with boundary violations, it is important to implement validation checks for variables used in percentage-based calculations. Ensure that these variables do not exceed their maximum logical values. This can be accomplished by incorporating `require` statements or similar validation mechanisms whenever such variables are assigned or modified. These safeguards will enforce correct operational boundaries, preserving the contract's intended functionality and preventing computational errors.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L303
Status	Unresolved

Description

The `safeTransferFrom` function aisre used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
nodedToken.safeTransferFrom(msg.sender, address(this), amount);

nodedStakes[msg.sender].push(NodedStake({
        amount: amount,
        startTime: block.timestamp,
        lockupDuration: nodedPool.lockupDurations[lockupIndex],
        apr: nodedPool.apys[lockupIndex]
    }));

    totalNodedStaked += amount;
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
Actual Transferred Amount = Balance After Transfer - Balance
Before Transfer
```

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L111,409
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

```
nodedToken = IERC20Upgradeable(_nodedToken);
...
nodedToken.transfer(msg.sender, nodedInterest);
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/NodedStaking.sol#L76,101,118,125,431,495
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
uint256 MAX_INT
address _nodedToken
address _balancerAddress
uint256 _nodedApr
address _user
uint256 _amount
address _tokenAddress
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.

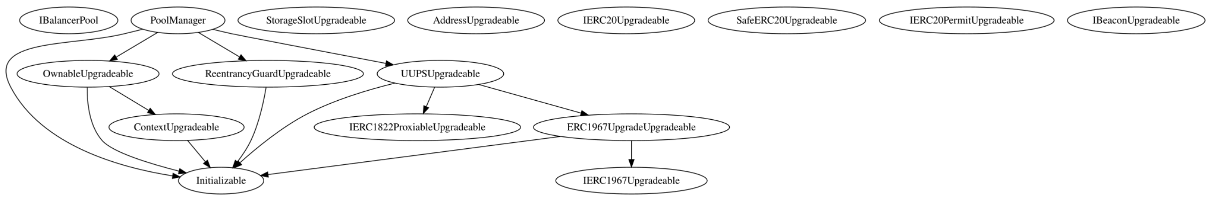
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

Functions Analysis

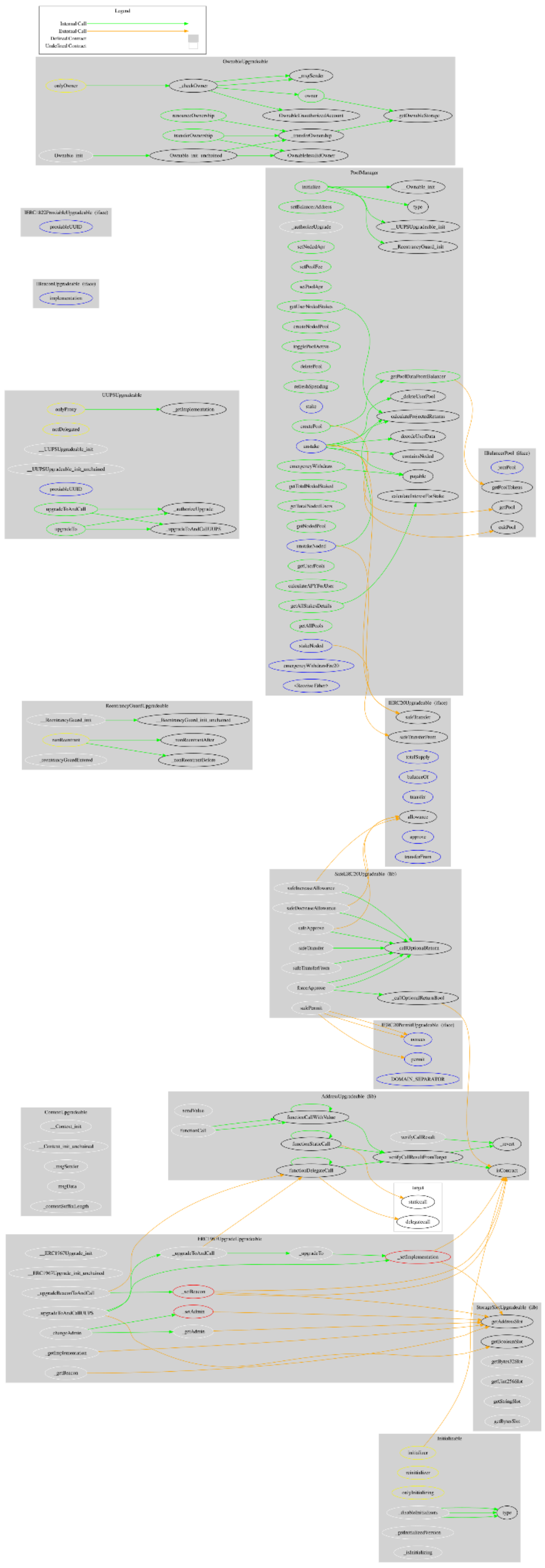
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
PoolManager	Implementation	Initializable, OwnableUpgradeable, ReentrancyGuardUpgradeable, UUPSUpgradeable		
	initialize	Public	✓	initializer
	setBalancerAddress	Public	✓	onlyOwner
	_authorizeUpgrade	Internal	✓	onlyOwner
	setNodedApr	Public	✓	onlyOwner
	setPoolFee	Public	✓	onlyOwner
	setPoolApr	Public	✓	onlyOwner
	getPoolDataFromBalancer	Public		-
	createPool	Public	✓	onlyOwner
	createNodedPool	Public	✓	onlyOwner
	togglePoolActive	Public	✓	onlyOwner
	deletePool	Public	✓	onlyOwner
	_deleteUserPool	Internal	✓	
	refreshSpending	Public	✓	onlyOwner
	stake	External	Payable	nonReentrant
	stakeNoded	External	✓	nonReentrant

	unstakeNoded	External	✓	nonReentrant
	getUserNodedStakes	Public		-
	getTotalNodedStaked	Public		-
	getTotalNodedUsers	Public		-
	getNodedPool	Public		-
	decodeUserData	Public		-
	unstake	External	✓	nonReentrant
	getUserPools	Public		-
	calculateAPYForUser	Public		-
	calculateInterestForStake	Public		-
	getAllStakesDetails	Public		-
	calculateProjectedReturns	Public		-
	getAllPools	Public		-
	containsNoded	Internal		
	emergencyWithdraw	Public	✓	onlyOwner
	emergencyWithdrawErc20	External	✓	onlyOwner
		External	Payable	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
OwnableUpgradeable
onlyOwner
_checkOwner
_msgSender
owner
OwnableUnauthorizedAccount
_getOwnableStorage
renounceOwnership
transferOwnership
_transferOwnership
OwnableInvalidOwner
__Ownable_init
__Ownable_init_unchained
IERC1822ProxiableUpgradeable (iface)
proxiableUUID
IBeaconUpgradeable (iface)
implementation
PoolManager
initialize
type
__UUPSUpgradeable_init
__ReentrancyGuard_init
setBalancerAddress
_authorizeUpgrade
setNodedApr
setPoolFee
setPoolApr
getUserNodedStakes
createNodedPool
togglePoolActive
deletePool
refreshSpending
stake
createPool
unstake
emergencyWithdraw
getTotalNodedStaked
getTotalNodedUsers
getNodedPool
unstakeNoded
getUserPools
calculateAPYForUser
getAllStakesDetails
getAllPools
stakeNoded
emergencyWithdrawErc20
<Receive Ether>
getPoolDataFromBalancer
_deleteUserPool
calculateProjectedReturns
decodeUserData
containsNoded
payable
calculateInterestForStake
IBalancerPool (iface)
joinPool
getPoolTokens
getPool
exitPool
UUPSUpgradeable
IERC20Upgradeable (iface)
safeTransfer
safeTransferFrom
totalSupply
balanceOf
ReentrancyGuardUpgradeable
SafeERC20Upgradeable (lib)
IERC20PermitUpgradeable (iface)
AddressUpgradeable (lib)
ContextUpgradeable
ERC1967UpgradeUpgradeable
StorageSlotUpgradeable (lib)
Initializable

Summary

MASTERNODED contract implements a staking mechanism. This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io