Cyberscope

Audit Report

MASTERNODED

July 2024

SHA256 3d2c3831dac3c88ee8c790383151cdd9d8cd1ce67021ea171f3d64e9690e66a3

Audited by © cyberscope

Table of Contents

Review

Testing Deploy	https://testnet.bscscan.com/address/0x6707d4054746695e1fe4133fcf6155e97523f6e5

Audit Updates

Initial Audit	09 Jul 2024

Source Files

Filename	SHA256
contracts/NodedUpgradable.sol	3d2c3831dac3c88ee8c790383151cdd9d8cd1ce67021ea171f3d64e9690e66a3

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.

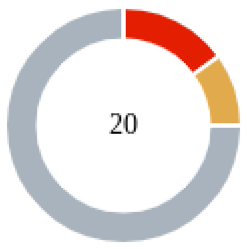
Unstake Functionality

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.

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	3
🟧	Medium	2
⬜	Minor / Informative	15

	Severity	Unresolved	Acknowledged	Resolved	Other
🟥	Critical	0	3	0	0
🟧	Medium	0	2	0	0
⬜	Minor / Informative	15	0	0	0

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
🔴	IID	Inconsistent Interest Distribution	Acknowledged
🔴	SLD	Staking Logic Discrepancy	Acknowledged
🔴	UFW	Unauthorized Fund Withdrawals	Acknowledged
🟠	ERDAP	Equivalent Rewards Distribution Across Pools	Acknowledged
🟠	UEC	Unchecked ETH Contribution	Acknowledged
⚪	CCR	Contract Centralization Risk	Unresolved
⚪	FRO	Function Reuse Optimization	Unresolved
⚪	ICBV	Inadequate Contract Balance Verification	Unresolved
⚪	MEE	Missing Events Emission	Unresolved
⚪	MPIV	Missing Pool ID Validation	Unresolved
⚪	MU	Modifiers Usage	Unresolved
⚪	PBV	Percentage Boundaries Validation	Unresolved
⚪	RAI	Redundant Address Initialization	Unresolved
⚪	TSI	Tokens Sufficiency Insurance	Unresolved

●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

IID - Inconsistent Interest Distribution

Criticality	Critical
Location	contracts/NodedUpgradable.sol#L229,262
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 is 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 {
        ...
        ExitPoolRequest memory request = ExitPoolRequest({
            assets: assets,
            minAmountsOut: amounts,
            userData: userData,
            toInternalBalance: false
        });

        balancer.exitPool(poolId, address(this),
payable(address(this)), request);

        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;
            uint256 amountAfterFee = difference - fee;
            IERC20Upgradeable(assets[i]).transfer(msg.sender,
amountAfterFee);
        }

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

SLD - Staking Logic Discrepancy

Criticality	Critical
Location	contracts/NodedUpgradable.sol#L159,229
Status	Acknowledged

Description

The contract is currently designed such that its `stake` function only considers the first asset with a non-zero amount from an array for staking, while the corresponding `unstake` function processes all the assets and amounts passed to it. This creates an inconsistency in how assets are managed, as the system records only a single asset as staked but allows the unstaking of multiple assets. This inconsistency can lead to errors in balance calculations, transaction validations, and might confuse users or lead to unexpected financial results.

```
   function stake(
        bytes32 poolId,
        address[] calldata assets,
        uint256[] calldata amounts,
        uint256 lockupIndex,
        bytes calldata userData
    ) external payable nonReentrant {
        Pool storage pool = pools[poolId];
        ...
        address assetAddress;
        uint256 assetAmount;

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

        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);

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

        stakes[msg.sender][poolId].push(
            Stake({
                amount: assetAmount,
                bptAmount: newBalance - initialBalance,
                assetAddress: assetAddress,
                startTime: block.timestamp,
                lockupDuration:
pool.lockupDurations[lockupIndex],
                apr: pool.apr,
                assets: assets,
                amounts: amounts
            })
        );

        ...
        }
    }

    function unstake(
        bytes32 poolId,
        uint256 stakeIndex,
        address[] memory assets,
        uint256[] memory amounts,
        bytes memory userData
    ) external nonReentrant {
        ...
        ExitPoolRequest memory request = ExitPoolRequest({
            assets: assets,
            minAmountsOut: amounts,
            userData: userData,
            toInternalBalance: false
        });

        balancer.exitPool(poolId, address(this),
payable(address(this)), request);

        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;
            uint256 amountAfterFee = difference - fee;
            IERC20Upgradeable(assets[i]).transfer(msg.sender,
amountAfterFee);
        }
        ...
    }
```

Recommendation

It is recommended to align the staking and unstaking functions either by modifying the `stake` function to handle all provided assets and amounts similarly to how the `unstake` function operates, or by restricting the `unstake` function to only process the asset(s) that were actually staked. This adjustment will ensure consistency across the contract's functions, reducing the potential for errors and improving the predictability and reliability of its operations.

Team Update

The team has acknowledged that this is not a security issue and states: *user will only ever stake 1 currency and according to balancer functionality it will be automatically divided into weightage of multiple currencies according to the pool weightage and the unstake function takes multiple currencies, should be clear now as balancer automatically adjusts 1 currency into multiple weightage.*

UFW - Unauthorized Fund Withdrawals

Criticality	Critical
Location	contracts/NodedUpgradable.sol#L229
Status	Acknowledged

Description

The contract is configured to allow users to unstake assets through an `unstake` function, which accepts arrays of `assets` and `amounts` as parameters. However, this function does not include checks to verify whether the specified amounts and assets actually match what the user originally staked. This oversight creates a significant security vulnerability, as users can exploit the contract by specifying any asset and amount, leading to unauthorized withdrawals. Such behavior could result in substantial financial losses for other stakeholders and undermine the integrity of the contract.

```
    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");

        uint256[] memory initialBalances = new
uint256[](assets.length);
        for (uint256 i = 0; i < assets.length; i++) {
            initialBalances[i] =
IERC20Upgradeable(assets[i]).balanceOf(address(this));
        }

        ExitPoolRequest memory request = ExitPoolRequest({
            assets: assets,
            minAmountsOut: amounts,
            userData: userData,
            toInternalBalance: false
        });

        balancer.exitPool(poolId, address(this),
payable(address(this)), request);

        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;
            uint256 amountAfterFee = difference - fee;
            IERC20Upgradeable(assets[i]).transfer(msg.sender,
amountAfterFee);
        }
    ...
    }
```

Recommendation

It is recommended to implement robust verification within the `unstake` function to confirm that the assets and amounts requested for withdrawal correspond precisely to those that were originally staked by the user. This could involve matching the unstake requests against records of staked assets and their respective amounts. Enhancing this verification process will safeguard against unauthorized withdrawals, ensuring that only legitimate, previously staked assets can be unstaked and protecting the contract from potential exploits.

Team Update

The team has acknowledged that this is not a security issue and states: *This check is basically overridden by userData where everything goes in encoded form in the balancer protocol. This user data is calculated on the backend and passed to the transaction directly according to the user output balances (which are derived by users share in the pool).*

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#L195
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.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L90,94,99,103,108,127,132
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 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;
            }
        }
    }
```

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#L290,304
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#L197,170,240,254
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#L90,94,99,108,159,229
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.

MPIV - Missing Pool ID Validation

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L108
Status	Unresolved

Description

The contract is currently deploying the `createPool` function without a crucial check to ensure that the `poolId` parameter is not equivalent to `bytes32(0)`. This omission can lead to the creation of pools with a default or invalid identifier, which may result in unintended interactions or the allocation of resources to non-existent pools. The absence of this check could also allow the execution of further code that depends on the validity of the `poolId`, potentially leading to errors or security vulnerabilities.

```
    function createPool(
        bytes32 poolId,
        uint256[] memory lockupDurations,
        uint256 apr,
        bool isActive,
        uint256 feePercentage
    ) public onlyOwner {
        require(feePercentage <= 10000, "Fee percentage cannot
exceed 100%");
        (address[] memory assets,,) =
getPoolDataFromBalancer(poolId);
        (address poolToken,) = balancer.getPool(poolId);
    ...
    }
```

Recommendation

It is recommended to add a validation step within the `createPool` function to verify that `poolId` is not equal to `bytes32(0)`. This check should be implemented as a requirement condition at the beginning of the function to prevent the function's execution if the condition is not met. Ensuring that all inputs are validated before use will help maintain the integrity of the contract's operations and enhance overall security.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L128,133,145
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#L90,99
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.

RAI - Redundant Address Initialization

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L83
Status	Unresolved

Description

The contract is initializing two different variables, `balancer` and `balancerAddress`, with the same address value. This redundancy indicates a potential oversight in the design of the contract's state management, leading to unnecessary bloating of the contract's code and possible confusion in its logical flow. Such practices can complicate future updates or integrations and may lead to inefficiencies in contract execution.

```
balancer = IBalancerPool(_balancerAddress);
balancerAddress = _balancerAddress;
```

Recommendation

It is recommended to utilize only one variable to hold the address if the address's purpose is to interact with a single contract interface. This approach simplifies the codebase, reduces the potential for errors, enhances readability, and optimizes gas costs during contract execution. Reviewing and consolidating variable assignments during the contract development phase can prevent such issues.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L85,265
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/NodedUpgradable.sol#L73,79,90,286,355
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.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L91
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
nodedApr = _nodedApr
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L172,173
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
address assetAddress
uint256 assetAmount
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L84
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
balancerAddress = _balancerAddress
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L2
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.0;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	contracts/NodedUpgradable.sol#L186,259,264,357
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
IERC20Upgradeable(assetAddress).transferFrom(msg.sender,
address(this), assetAmount)
IERC20Upgradeable(assets[i]).transfer(msg.sender,
amountAfterFee)
nodedToken.transfer(msg.sender, nodedInterest)
IERC20Upgradeable(_tokenAddress).transfer(msg.sender, _amount)
```

Recommendation

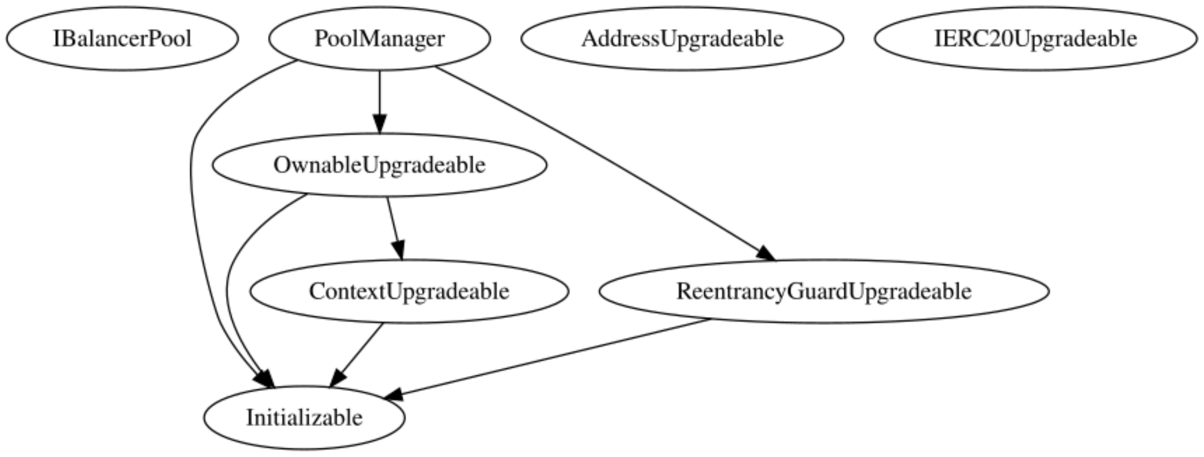
The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

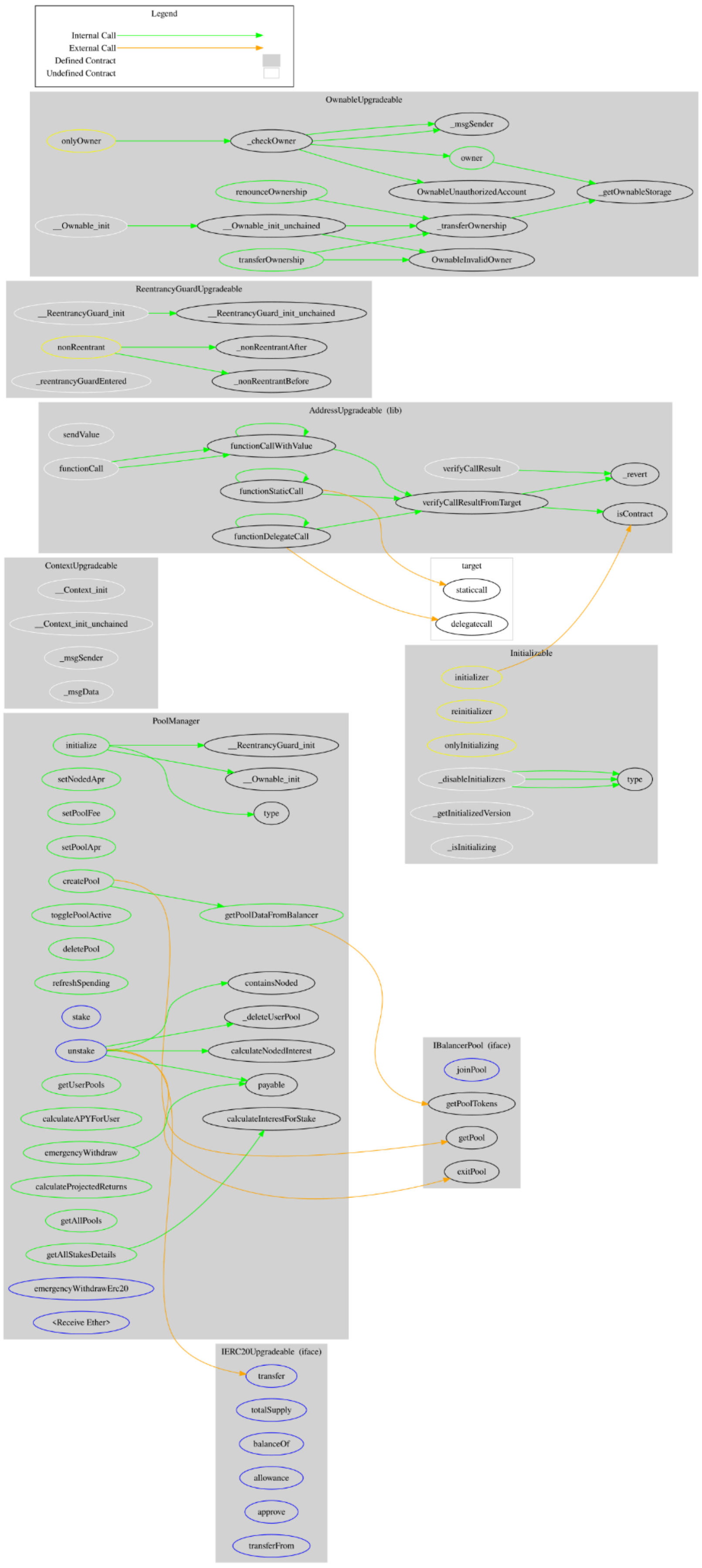
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IBalancerPool	Interface			
	joinPool	External	Payable	-
	exitPool	External	✓	-
	getPoolTokens	External		-
	getPool	External		-
PoolManager	Implementation	Initializable, OwnableUpgradeable, ReentrancyGuardUpgradeable		
	initialize	Public	✓	initializer
	setNodedApr	Public	✓	onlyOwner
	setPoolFee	Public	✓	onlyOwner
	setPoolApr	Public	✓	onlyOwner
	getPoolDataFromBalancer	Public		-
	createPool	Public	✓	onlyOwner
	togglePoolActive	Public	✓	onlyOwner
	deletePool	Public	✓	onlyOwner
	_deleteUserPool	Internal	✓	
	refreshSpending	Public	✓	onlyOwner

	stake	External	Payable	nonReentrant
	unstake	External	✓	nonReentrant
	getUserPools	Public		-
	calculateAPYForUser	Public		-
	calculateInterestForStake	Public		-
	getAllStakesDetails	Public		-
	calculateProjectedReturns	Public		-
	getAllPools	Public		-
	containsNoded	Internal		
	calculateNodedInterest	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
renounceOwnership
OwnableUnauthorizedAccount
_getOwnableStorage
__Ownable_init
__Ownable_init_unchained
_transferOwnership
transferOwnership
OwnableInvalidOwner
ReentrancyGuardUpgradeable
__ReentrancyGuard_init
__ReentrancyGuard_init_unchained
nonReentrant
_nonReentrantAfter
_reentrancyGuardEntered
_nonReentrantBefore
AddressUpgradeable (lib)
sendValue
functionCallWithValue
functionCall
verifyCallResult
_revert
functionStaticCall
verifyCallResultFromTarget
isContract
functionDelegateCall
ContextUpgradeable
__Context_init
__Context_init_unchained
_msgSender
_msgData
target
staticcall
delegatecall
Initializable
initializer
reinitializer
onlyInitializing
_disableInitializers
type
_getInitializedVersion
_isInitializing
PoolManager
initialize
__ReentrancyGuard_init
setNodedApr
__Ownable_init
setPoolFee
type
setPoolApr
createPool
togglePoolActive
getPoolDataFromBalancer
deletePool
refreshSpending
containsNoded
stake
_deleteUserPool
unstake
calculateNodedInterest
getUserPools
payable
calculateAPYForUser
calculateInterestForStake
emergencyWithdraw
calculateProjectedReturns
getAllPools
getAllStakesDetails
emergencyWithdrawErc20
<Receive Ether>
IBalancerPool (iface)
joinPool
getPoolTokens
getPool
exitPool
IERC20Upgradeable (iface)
transfer
totalSupply
balanceOf
allowance
approve
transferFrom

Summary

Token is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler error or critical issues. The contract Owner can access some admin functions that can not be used in a malicious way to disturb the users' transactions. There is also a limit of max 25% fees.

There are some functions that can be abused by the owner, like manipulating fees and transferring funds to the team's wallet. The maximum fee percentage that can be set is 25%. A multi-wallet signing pattern will provide security against potential hacks. Temporarily locking the contract or renouncing ownership will eliminate all the contract threats.

The contract can be converted into a honeypot and prevent users from selling if the owner abuses the admin functions.

This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

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