Cyberscope

Audit Report

GLUTECH

July 2023

Network	BSC TESTNET
Address	0x0Dd17C6D85168A6E7841149945f166A0d4780e51
Audited by	© cyberscope

Table of Contents

Review

Explorer	https://testnet.bscscan.com/address/0x0dd17c6d85168a6e7841149945f166a0d4780e51

Audit Updates

Initial Audit	01 Jul 2023

Source Files

Filename	SHA256
contracts/StakingManager.sol	02105d10655b7221f95312a4aba75c3b7f359c753a5141e434f4c0161be48eb4
@openzeppelin/contracts/utils/Address.sol	aafa8f3e41700a8353aabcdf020e06735753e6bc4b615279b43de53cfbb4f2cd
@openzeppelin/contracts/utils/math/SafeMath.sol	0dc33698a1661b22981abad8e5c6f5ebca0dfe5ec14916369a2935d888ff257a
@openzeppelin/contracts/token/ERC20/IERC20.sol	94f23e4af51a18c2269b355b8c7cf4db8003d075c9c541019eb8dcf4122864d5
@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol	b5a1340c5232f387b15592574f27eef78f6017bdc66542a1cea512ad4f78a0d2
@openzeppelin/contracts/security/ReentrancyGuard.sol	aa73590d5265031c5bb64b5c0e7f84c44cf5f8539e6d8606b763adac784e8b2e

Overview

The StakingManager contract is a comprehensive staking mechanism that enables users to stake tokens according to various staking plans and earn rewards based on their stake amount and the duration of the stake.

The contract owner has pre-set the staking plans, the leadership ranks and referral Levels. Each staking plan has unique parameters, including minimum and maximum stake amounts, daily Return on Investment (ROI), staking period, and keeps track of the total staked, and total payouts amounts.

Stake

The StakingManager contract allows users to stake tokens in a specific staking plan by depositing a certain amount. The rewards for the staked tokens accrue over time, based on the parameters of the staking plan they have chosen.

Unstake

Users have the flexibility to unstake their tokens, including all of their accumulated rewards, at any time. However, if the unstaking action occurs before the end of the staking period defined in the plan, a penalty fee is applied.

Harvest

In addition to staking and unstaking, users can also harvest their rewards by calling the `harvest` and `harvestReferralEarnings` functions. This allows users to claim their accumulated rewards without unstaking their tokens.

Fees

The contract implements several fees to manage the staking process. When users stake tokens, a fee of 30% is applied. Additionally, a harvest fee of 3% is charged every time users harvest their rewards or their referral rewards. If users unstake their tokens before the end of the staking plan period, a penalty fee of 25% is applied. These fees are designed to incentivize users to maintain their stakes for longer periods and to support the sustainability of the staking platform.

Rewards

The contract enables users to accumulate rewards in two distinct ways. The first way is through staking rewards. These are calculated based on the daily ROI set for each staking plan, the amount of tokens a user has staked, and the duration of the stake. This calculation is performed by the `getRewards` function, which takes into account the user's stake amount, the staking plan's daily ROI, and the elapsed time since the user's earning start time.

The second way is through team sales earnings, which are calculated based on the user's leadership rank. The `teamEarnings` function calculates these rewards, taking into account the user's current rank and the weekly earnings associated with that rank. However, in order for the `teamEarnings` function to be called for a user, the user must first call the `unstake` or the `harvest` function. This is necessary because these functions update the `lastTeamHarvest` variable, which must not be zero for the `teamEarnings` function to work correctly.

Users are encouraged to claim their rewards as near to the conclusion of the staking period as feasible. This is due to the fact that rewards are increased with the number of weeks that have elapsed from the time of staking until the end of the staking period. Once the staking period has concluded, the user's account will not accrue any additional rewards.

Leadership Ranks

Users can ascend to higher leadership ranks by fulfilling certain criteria. These criteria include increasing the total number of totalDirectReferrals, which is the total referrals each user have, increasing the totalInvestments which is the total amount each user has staked and increasing the leadershipScore, which is the total amount that user how referred the user have staked.

When all these criteria are met, the user is eligible to ascend to a higher leadership rank, which can result in increased rewards. This system incentivizes users to actively participate in the staking process and to refer new users to the platform, thereby contributing to the growth and sustainability of the staking ecosystem.

Harvest Referral Earnings

In addition to the staking rewards, the contract also provides a mechanism for users to earn rewards from referrals. Users can harvest their referral earnings by calling the `harvestReferralEarnings` function.

The contract uses the OpenZeppelin library for secure and standard compliant implementations of ERC20 token interactions, safe math operations, and protection against re-entrancy attacks.

The StakingManager contract is a powerful solution for managing staking activities, providing users with a variety of options to earn rewards, and ensuring the security and integrity of the staking process.

Roles

Owner

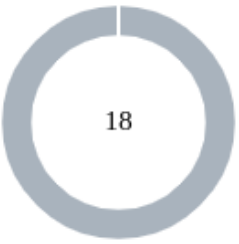
The owner has authority to set the StakingManager contract. The owner is responsible for setting up the 'admin' and 'liquiditySupportBot' wallets in the constructor function during the deployment process.

User

The user can interact with the following functions:

- `function teamEarnings(address _user)`
- `function getRewards(address _account,uint256 planId)`
- `function getUserPlanDetails(address _address,uint256 planId)`
- `function getAllUserPlansEarnings(address _address)`
- `function getUserDetails(address _address)`
- `function getAvailableReferralRewards(address _account)`
- `function getReferralRewards(address _account)`
- `function stake(uint256 planId)`
- `function harvest(uint256 planId)`
- `function harvestReferralEarnings()`
- `function unstake(uint256 planId)`
- `function recordReferral(address _referrer)`

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	18

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	18	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	PRCM	Potential Referrals Counter Manipulation	Unresolved
	PCR	Potential Circular Referral	Unresolved
	PCB	Potential Criteria Bypass	Unresolved
	RFP	Redundant Function Parameter	Unresolved
	ADTS	Appropriate Data Type Size	Unresolved
	MCM	Misleading Comment Messages	Unresolved
	LAFI	Leadership Amount Format Inconsistency	Unresolved
	AOI	Arithmetic Operations Inconsistency	Unresolved
	MU	Modifiers Usage	Unresolved
	CO	Code Optimization	Unresolved
	ULI	Unnecessary Loop Iterations	Unresolved
	RSML	Redundant SafeMath Library	Unresolved
	RSK	Redundant Storage Keyword	Unresolved

●	IDI	Immutable Declaration Improvement	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved

PRCM - Potential Referrals Counter Manipulation

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L477
Status	Unresolved

Description

The contract is designed to record referrals using the `recordReferral` function. This function is public, which means that anyone can call it. A malicious actor could potentially exploit this by using multiple wallets to refer the same account repeatedly. This would artificially inflate the `totalDirectReferrals` count for a user, as each call to the function would increment this value. This could lead to an inaccurate representation of the actual referrals made by a user.

```
    function recordReferral(address _referrer) public nonReentrant {
        ...

        user.referrals.push(msg.sender);
        user.totalDirectReferrals = user.totalDirectReferrals.add(1);

        totalTeams = totalTeams.add(1);

        ...
    }
```

Recommendation

The team is recommended to implement a mechanism that requires a user to have already staked a certain amount in the contract before they can call the `recordReferral` function. This would add an additional layer of security and make it more difficult for a user to manipulate the referral count.

This way ensures that the `recordReferral` function can only be called by a user who has staked an amount greater than zero.

PCR - Potential Circular Referral

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L479
Status	Unresolved

Description

The contract is using the `record1Referral` function to record referrals between users. The function includes a check to prevent direct circular referrals, where a user cannot refer themselves `( referrals[_referrer] != msg.sender )`. However, the function does not prevent indirect circular referrals, where a third user can refer the first user, creating a circular referral chain.

For example, if UserA passes as `_referrer` UserB, UserB passes UserC, and UserC passes UserA, a circular referral chain is created. This can lead to issues when the `getUplines` function is called, as it can result in a continuous loop of referrals, leading to an array of circular addresses.

Furthermore, this loophole could potentially be exploited to manipulate the `leadershipScore`. Since the referral rewards are given each time a referral is made, a user in a circular referral chain could receive the same amount three times. This could artificially inflate their `leadershipScore`, leading to an unfair advantage and potentially destabilizing the system.

```
    function recordReferral(address _referrer) public nonReentrant {
        require(msg.sender.code.length == 0, "Contracts not allowed.");
        if (
            msg.sender != address(0) &&
            _referrer != address(0) &&
            msg.sender != _referrer &&
            referrals[msg.sender] == address(0) &&
            referrals[_referrer] != msg.sender // this is to avoid
circular referral
        )
```

```
    function getUplines(
        address _address,
        uint256 limit
    ) internal view returns (address[] memory) {
        address[] memory uplines = new address[](limit);
        address current = _address;
        for (uint i = 0; i < limit; i++) {
            if (referrals[current] == address(0)) {
                break;
            }
            uplines[i] = referrals[current];
            current = uplines[i];
        }
        return uplines;
    }
```

Recommendation

The team is advised to implement additional checks in the `recordReferral` function to prevent indirect circular referrals. This could be achieved by checking if the new referral would lead to a circular chain before recording it. This change will ensure that the referral system is robust and prevents both direct and indirect circular referrals, improving the overall referral system of the contract.

PCB - Potential Criteria Bypass

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L604
Status	Unresolved

Description

The contract is using the `recordReferral` function to record referrals between users. While it includes a check to prevent direct circular referrals, it does not prevent indirect circular referrals, as described in the `PCR - Potential Circular Referral` finding. This allows users to create a circular referral chain, which can be exploited to manipulate the `leadershipScore` variable and also the `totalDirectReferrals` variable, as described in the `PRCM - Potential Referrals Counter Manipulation` finding.

Additionally, the `totalInvestments` variable can also be manipulated by users stacking and unstacking amounts in the contract by using the `stake` and `unstake` functions, if they are paying the fees required.

By exploiting these vulnerabilities, users can bypass the criteria necessary to update their ranks and ascend to higher rank levels, which offer greater rewards. This could lead to an unfair distribution of rewards and potentially destabilize the contract's economy.

```
if (
    user.leadershipScore >= leadershipRank.teamVolume &&
    user.totalInvestments >= leadershipRank.investments &&
    user.totalDirectReferrals >= leadershipRank.directReferrals
) {
    currentPosition = currentPosition.add(1);
}
```

Recommendation

The team is advised to implement additional checks in the `recordReferral` function to prevent indirect circular referrals. This could involve maintaining a record of all referral

chains and checking for circularity whenever a new referral is added or require users to have staked an amount before they are allowed to call the `recordReferral` function.

Furthermore, the team should conduct a thorough review of the rank update logic in the `updateLeadershipRank` function. This is to ensure that it cannot be exploited to unfairly gain higher ranks. This could involve additional checks or modifications to the rank update criteria.

RFP - Redundant Function Parameter

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L570
Status	Unresolved

Description

The contract is using the `getUplines` function that accepts two parameters: `_address` and `limit`. The `limit` parameter is used to define the size of the uplines array and to control the number of iterations in the for loop.

However, in all instances where the `getUplines` function is called, the `limit` parameter is always set to 10 and never changes. This makes the `limit` parameter redundant, as its value is constant.

```
    function getUplines(
        address _address,
        uint256 limit
    ) internal view returns (address[] memory) {
        address[] memory uplines = new address[](limit);
        address current = _address;
        for (uint i = 0; i < limit; i++) {
            if (referrals[current] == address(0)) {
                break;
            }
            uplines[i] = referrals[current];
            current = uplines[i];
        }
        return uplines;
    }
```

Recommendation

The team is advised to remove the `limit` parameter from the `getUplines` function and replace it with a constant value of 10 within the function. This will simplify the function signature and avoid potential confusion or misuse in the future.

ADTS - Appropriate Data Type Size

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L57
Status	Unresolved

Description

The contract is using the uint256 data type in order to store several constant variables, namely `HARVEST_FEE_PERCENTAGE`, `PENALTY_PERCENTAGE`, `REFERRAL_LEVELS`, and `LIQUIDITY_SUPPORT_PERCENTAGE`. These constants are set to the values of `300`, `2500`, `10`, and `3000` respectively. The use of the uint256 data type provides significantly more storage capacity than is necessary for these numerical constants.

```
uint256 private constant HARVEST_FEE_PERCENTAGE = 300;
uint256 private constant PENALTY_PERCENTAGE = 2500;
uint256 private constant REFERRAL_LEVELS = 10;
uint256 public constant LIQUIDITY_SUPPORT_PERCENTAGE = 3000;
```

Recommendation

Since the highest constant value in your contract is 3000, which can be represented within the range of 2^12 (0 to 4095), the team is advised to use a uint16 data type for these constants. The uint16 data will use fewer bits and take up less storage space on the blockchain.

MCM - Misleading Comment Messages

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L60
Status	Unresolved

Description

The contract is using misleading comment messages. These comment messages do not accurately reflect the actual implementation, making it difficult to understand the source code. As a result, the users will not comprehend the source code's actual implementation.

The comment suggests that the `LIQUIDITY_SUPPORT_PERCENTAGE` constant represents a '30% APR' (Annual Percentage Rate), which could be interpreted as the annual yield provided to users.

However, the `LIQUIDITY_SUPPORT_PERCENTAGE` constant is used to determine a fee that is transferred to the `liquiditySupportBot` wallet.

```
uint256 public constant LIQUIDITY_SUPPORT_PERCENTAGE = 3000; // 30% APR

 uint256 liquiditySupportFee = (
            msg.value.mul(LIQUIDITY_SUPPORT_PERCENTAGE)
        ).div(10000);

liquiditySupportBot.transfer(liquiditySupportFee);
```

Recommendation

The team is advised to carefully review the comment in order to reflect the actual implementation. To improve code readability, the team should use more specific and descriptive comment messages.

LAFI - Leadership Amount Format Inconsistency

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L128
Status	Unresolved

Description

The contract assigns values to the leadershipRanks mappings. However, the value assigned to the fourth parameter of `leadershipRanks[5]` is `10799999999999999000`, which is not expressed in the 'ether' unit like the corresponding values in the other LeadershipRank structures.

This could potentially lead to confusion or errors, as it deviates from the standard formatting used elsewhere in the contract.

```
leadershipRanks[5] = LeadershipRank(
    7.21 ether,
    20,
    360.27 ether,
    10799999999999999000
);

leadershipRanks[6] = LeadershipRank(
    18.01 ether,
    20,
    900.69 ether,
    18 ether
);
```

Recommendation

It is advised to review the value assigned to the fourth parameter of `leadershipRanks[5]` to ensure it aligns with the intended logic of the contract.

It is recommended to standardize the usage of ether units throughout the contract. The contract should be modified to either exclusively use the 'ether' keyword for all ether values or entirely rely on wei units, depending on the specific requirements and design

considerations. This consistency will help maintain the contract's integrity and mitigate potential confusion or errors arising from inconsistent unit usage.

AOI - Arithmetic Operations Inconsistency

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L226,348
Status	Unresolved

Description

The contract uses both the SafeMath library and native arithmetic operations. The SafeMath library is commonly used to mitigate vulnerabilities related to integer overflow and underflow issues. However, it was observed that the contract also employs native arithmetic operators (such as +, -, *, /) in certain sections of the code.

The combination of SafeMath library and native arithmetic operations can introduce inconsistencies and undermine the intended safety measures. This discrepancy creates an inconsistency in the contract's arithmetic operations, increasing the risk of unintended consequences such as inconsistency in error handling, or unexpected behavior.

```
uint256 pending = staking.rewardDebt + pendingReward;

user.totalInvestments = user.totalInvestments.add(msg.value);
```

Recommendation

To address this finding and ensure consistency in arithmetic operations, it is recommended to standardize the usage of arithmetic operations throughout the contract. The contract should be modified to either exclusively use SafeMath library functions or entirely rely on native arithmetic operations, depending on the specific requirements and design considerations. This consistency will help maintain the contract's integrity and mitigate potential vulnerabilities arising from inconsistent arithmetic operations.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L337,347
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(planId >= 1 && planId <= 3, "Invalid plan ID");
```

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.

CO - Code Optimization

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L195
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The code could refactored in the `getRewards` function. Specifically, the function performs similar calculations twice, with the only difference being the multiplication of the return amount with the `STAKING_DURATION_DAYS` variable `if timeDiff >= stakingPlan.stakingPeriod` .

The `getRewards` function calculates the reward amount based on the staking amount, the daily ROI, and either the `STAKING_DURATION_DAYS` or the `timeDiff` , depending on whether `timeDiff` is greater than or equal to the `stakingPlan.stakingPeriod` . This results in duplicated code, which can make the contract more difficult to read and maintain.

```
if (staking.amount > 0) {
    StakingPlan storage stakingPlan = stakingPlans[planId];
    uint256 STAKING_DURATION_DAYS = stakingPlan.stakingPeriod / 1 days;
    uint256 stakeAmount = staking.amount;
    uint256 timeDiff;
    unchecked {
        timeDiff = block.timestamp.sub(staking.earningStartTime);
    }
    if (timeDiff >= stakingPlan.stakingPeriod) {
        return
            (
                ((stakeAmount.mul(stakingPlan.dailyROI)).div(10000))
                    .mul(STAKING_DURATION_DAYS)
            ).add(staking.rewardDebt);
    }
    uint256 rewardAmount = (
        ((stakeAmount.mul(stakingPlan.dailyROI)).div(10000)).mul(
            timeDiff
        )
    ).div(1 days);
    pendingReward = rewardAmount;
}

uint256 pending = staking.rewardDebt + pendingReward;
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. It is recommended to refactor the `getRewards` function to eliminate the duplicated code. One possible approach is to calculate the reward amount once, using a variable that is set to either `STAKING_DURATION_DAYS` or `timeDiff` based on the condition `timeDiff >= stakingPlan.stakingPeriod`. This would reduce the complexity of the function and improve the readability of the contract.

```
uint256 duration = (timeDiff >= stakingPlan.stakingPeriod) ?
STAKING_DURATION_DAYS : timeDiff;

uint256 rewardAmount =
((stakeAmount.mul(stakingPlan.dailyROI)).mul(duration).div(10000));
```

That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

ULI - Unnecessary Loop Iterations

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L554
Status	Unresolved

Description

The contract is using a for loop within the `updateUplines` function to iterate over the array `userUplines`. For each iteration, it checks `if (referrer != address(0))`. If this condition is met, the function executes the code inside the if segment.

However, the loop does not terminate after the if condition is met and continues to the next iterations. This could potentially lead to unnecessary iterations, especially if the referrer is not equal to the null address in the first few iterations. Once the if condition is met, there is no need for the loop to continue to the next iterations, as the necessary updates have already been made.

```
    function updateUplines(address _user) internal {
        address[] memory userUplines = getUplines(_user, 10);

        for (uint256 i = 0; i < userUplines.length; i++) {
            address referrer = userUplines[i];

            if (referrer != address(0)) {
                updateLeadershipRank(referrer);
                User storage user = users[referrer];
                user.totalTeam = user.totalTeam.add(1);
            }
        }
    }
```

Recommendation

The team is advised to include a break statement within the if condition to terminate the loop as soon as the if condition is met. This will optimize the function by avoiding unnecessary iterations.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	contracts/StakingManager.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/v0.8.16/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

RSK - Redundant Storage Keyword

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L171,201,204,269,288,296,304,511
Status	Unresolved

Description

The contract uses the `storage` keyword in a view function. The `storage` keyword is used to persist data on the contract's storage. View functions are functions that do not modify the state of the contract and do not perform any actions that cost gas (such as sending a transaction). As a result, the use of the `storage` keyword in view functions is redundant.

```
User storage user
Staking storage staking
StakingPlan storage stakingPlan
User storage referredUser
```

Recommendation

It is generally considered good practice to avoid using the `storage` keyword in view functions because it is unnecessary and can make the code less readable.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L86
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
contractInitializedAt
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L170,196,231,240,250,286,293,477,522,534,554,569,585,586
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.

```
address _user
address _account
address _address
address _referrer
uint256 _transactionAmount
```

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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L205,212,218
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 rewardAmount = (

((stakeAmount.mul(stakingPlan.dailyROI)).div(10000)).mul(
                    timeDiff
                )
            ).div(1 days)
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	contracts/StakingManager.sol#L84,85
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.

```
admin = _adminAdd
liquiditySupportBot = _liquiditySupportBotAddress
```

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/StakingManager.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 ^0.8.9;
```

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.

Functions Analysis

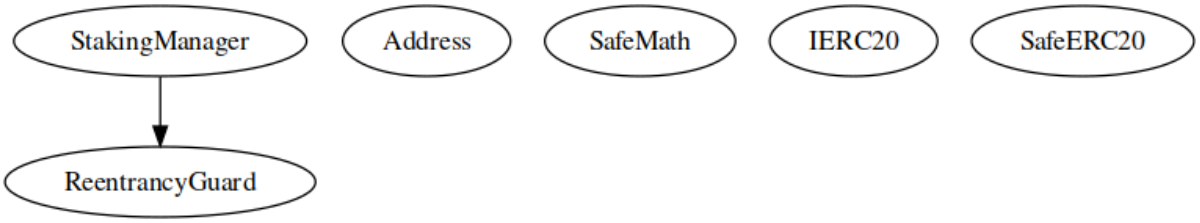
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
StakingManager	Implementation	ReentrancyGuard		
		Public	✓	-
	teamEarnings	Public		-
	getRewards	Public		-
	getUserPlanDetails	External		-
	getAllUserPlansEarnings	Public		-
	getUserDetails	External		-
	getAvailableReferralRewards	Public		-
	getReferralRewards	Public		-
	stake	External	Payable	nonReentrant
	harvest	External	✓	nonReentrant
	harvestReferralEarnings	External	✓	nonReentrant
	unstake	External	✓	nonReentrant
	recordReferral	Public	✓	nonReentrant
	_harvestableAmount	Private		
	updateUserStake	Internal	✓	
	updateUplinesEarnings	Internal	✓	
	updateUplines	Internal	✓	

	getUplines	Internal		
	balanceLeadershipRank	Internal	✓	
	updateLeadershipRank	Internal	✓	
Address	Library			
	isContract	Internal		
	sendValue	Internal	✓	
	functionCall	Internal	✓	
	functionCall	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionStaticCall	Internal		
	functionStaticCall	Internal		
	functionDelegateCall	Internal	✓	
	functionDelegateCall	Internal	✓	
	verifyCallResult	Internal		
SafeMath	Library			
	tryAdd	Internal		
	trySub	Internal		
	tryMul	Internal		
	tryDiv	Internal		
	tryMod	Internal		

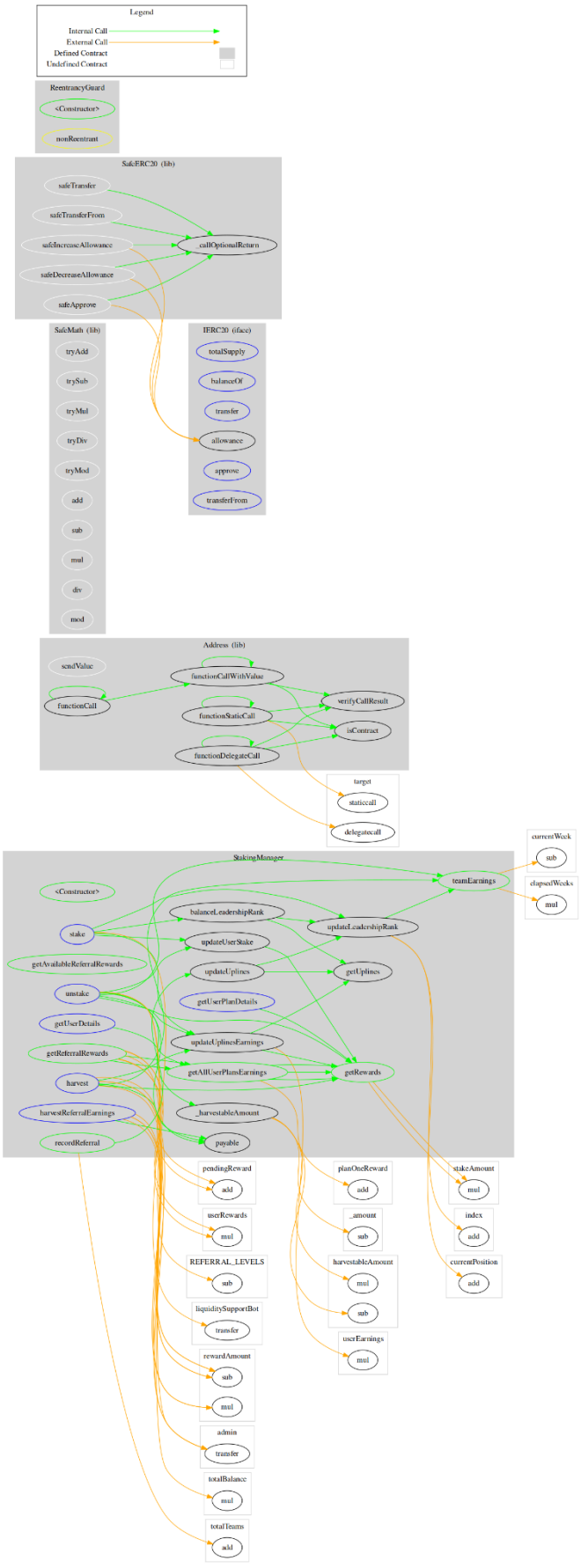
	add	Internal		
	sub	Internal		
	mul	Internal		
	div	Internal		
	mod	Internal		
	sub	Internal		
	div	Internal		
	mod	Internal		
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
SafeERC20	Library			
	safeTransfer	Internal	✓	
	safeTransferFrom	Internal	✓	
	safeApprove	Internal	✓	
	safeIncreaseAllowance	Internal	✓	
	safeDecreaseAllowance	Internal	✓	

	_callOptionalReturn	Private	✓	
ReentrancyGuard	Implementation			
		Public	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
ReentrancyGuard
<Constructor>
nonReentrant
SafeERC20 (lib)
safeTransfer
safeTransferFrom
safeIncreaseAllowance
safeDecreaseAllowance
safeApprove
_callOptionalReturn
SafeMath (lib)
tryAdd
trySub
tryMul
tryDiv
tryMod
add
sub
mul
div
mod
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
Address (lib)
sendValue
functionCall
functionCallWithValue
functionStaticCall
functionDelegateCall
verifyCallResult
isContract
target
staticcall
delegatecall
StakingManager
<Constructor>
stake
getAvailableReferralRewards
unstake
getUserDetails
getReferralRewards
harvest
harvestReferralEarnings
recordReferral
balanceLeadershipRank
updateUserStake
updateUplines
getUserPlanDetails
updateUplinesEarnings
getAllUserPlansEarnings
_harvestableAmount
payable
teamEarnings
updateLeadershipRank
getUplines
getRewards
currentWeek
sub
elapsedWeeks
mul
pendingReward
add
userRewards
mul
REFERRAL_LEVELS
sub
liquiditySupportBot
transfer
rewardAmount
sub
mul
admin
transfer
totalBalance
mul
totalTeams
add
planOneReward
add
_amount
sub
harvestableAmount
mul
sub
userEarnings
mul
stakeAmount
mul
index
add
currentPosition
add

Summary

StakingManager contract implements a staking and rewards 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