Cyberscope

Audit Report

Sphere

September 2023

Repository https://github.com/solstarter-org/sphere-staking-ethereum/blob/master/contracts/Staking.sol

Commit 6d61066a2c302ebe595acb5acb8a503550f9dbb6

Audited by © cyberscope

Table of Contents

Table of Contents **1**
Review **3**
Audit Updates 3
Source Files 3
Overview **4**
Stake Functionality 4
Unstake Functionality 4
Roles 6
Owner 6
User 6
Findings Breakdown **7**
Diagnostics **8**
IBA - Inconsistent Balance Allocation 10
Description 10
Recommendation 11
CR - Centralization Risk 12
Description 12
Recommendation 13
MC - Missing Check 14
Description 14
Recommendation 15
ILT - Inconsistent Lockup Times 16
Description 16
Recommendation 17
RCI - Reward Calculation Inconsistency 18
Description 18
Recommendation 19
MSC - Missing Sanity Check 21
Description 21
Recommendation 22
RNRM - Redundant No Reentrant Modifier 23
Description 23
Recommendation 23
MVN - Misleading Variables Naming 24
Description 24
Recommendation 25
PTAI - Potential Transfer Amount Inconsistency 26
Description 26
Recommendation 27

EIS - Excessively Integer Size 28
Description 28
Recommendation 28
OCTD - Transfers Contract's Tokens 29
Description 29
Recommendation 29
IEI - Inconsistent Event Indexing 30
Description 30
Recommendation 30
IUC - Incorrect Unstake Check 31
Description 31
Recommendation 31
RSML - Redundant SafeMath Library 32
Description 32
Recommendation 32
L04 - Conformance to Solidity Naming Conventions 33
Description 33
Recommendation 34
L07 - Missing Events Arithmetic 35
Description 35
Recommendation 35
L19 - Stable Compiler Version 36
Description 36
Recommendation 36
Functions Analysis **37**
Inheritance Graph **39**
Flow Graph **40**
Summary **41**
Disclaimer **42**
About Cyberscope **43**

Review

Repository	https://github.com/solstarter-org/sphere-staking-ethereum/blob/master/contracts/Staking.sol
Commit	6d61066a2c302ebe595acb5acb8a503550f9dbb6

Audit Updates

Initial Audit	15 Sep 2023

Source Files

Filename	SHA256
Staking.sol	50642b2b6ca508c624c8448339eb54ee4c39dbf5b697f1cd960b131f9af5e088

Overview

The `Staking` contract is a staking that allows users to stake tokens in various pools to earn `stakingToken` as a reward. The contract is upgradeable, meaning that its logic can be updated by the contract owner. It also uses OpenZeppelin's libraries for standard functionalities like ownership, reentrancy guard, and safe math operations. The contract has an early pool with specific standards and the contract allows the owner to add more pools with different configurations. The contract also has admin functionalities to update the pool information and withdraw the `stakingToken`.

Stake Functionality

The `stake` function enables users to deposit `stakingToken` into a designated pool, identified by its `pid` number. After validating the pool index and ensuring the staking amount is within the pool's minimum and maximum limits, the function proceeds to handle additional conditions. In the case of the early pool (where pid == 0), staking is permitted only within a specific timeframe since the contract's initialization. The function also checks that the user has not already staked an amount in the specific `pid` pool. The function updates both the pool's and the user's information, including their staked balances and the last time of staking. The staking tokens are then transferred from the user's address to the contract, and the pool's balance and reward allocations are decreased accordingly. Rewards are calculated based on the staked amount and the pool's APY percentage. Finally, an event is emitted to log the staking action.

Unstake Functionality

The `unstake` function enables users to withdraw their staked tokens and collect any applicable rewards. It first verifies that the user has tokens staked and that the pool's freeze time has elapsed since the last staking event. The function then checks whether the `poolLockupSeconds` duration has passed. If not, the function proceeds to unstake only the original staked amount, excluding any additional rewards. However, if the `poolLockupSeconds` duration has passed, rewards are calculated based on the pool's APY and added to the staked amount. Subsequently, the function updates both the pool's and the user's information, including staked balances. The staked tokens, along with any

earned rewards if applicable, are then transferred back to the user's address. Finally, an event is emitted to log the unstaking action.

Roles

Owner

The owner has the authority to initialize the Staking contract. The owner is responsible for setting up the staking token, early pool information, pool lifetime, and pool freeze time.

The owner can interact with the following functions:

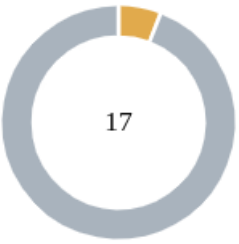
- function __Staking_init(address _token, PoolInfo calldata _earlyPoolData, uint256 _lifeTime, uint256 freezeTime)
- function adminWithdraw(uint256 amount)
- function setPoolFreezeTime(uint256 freezeTime)
- function addPoolInfo(PoolInfo calldata _poolData)
- function setPoolInfo(uint256 pid, PoolInfo calldata _poolData)

User

The user can interact with the following functions:

- function stake(uint256 pid, uint256 amount)
- function unstake(uint256 pid)
- function readPoolInfo(uint256 pid)
- function getUserInfo(uint256 pid, address user)
- function getUserTotalBalance(address user)
- function getPoolLength()

Findings Breakdown



Critical	0
Medium	1
Minor / Informative	16

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	1	0	0	0
Minor / Informative	16	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	IBA	Inconsistent Balance Allocation	Unresolved
Minor	CR	Centralization Risk	Unresolved
Minor	MC	Missing Check	Unresolved
Minor	ILT	Inconsistent Lockup Times	Unresolved
Minor	RCI	Reward Calculation Inconsistency	Unresolved
Minor	MSC	Missing Sanity Check	Unresolved
Minor	RNRM	Redundant No Reentrant Modifier	Unresolved
Minor	MVN	Misleading Variables Naming	Unresolved
Minor	PTAI	Potential Transfer Amount Inconsistency	Unresolved
Minor	EIS	Excessively Integer Size	Unresolved
Minor	OCTD	Transfers Contract's Tokens	Unresolved
Minor	IEI	Inconsistent Event Indexing	Unresolved
Minor	IUC	Incorrect Unstake Check	Unresolved
Minor	RSML	Redundant SafeMath Library	Unresolved

●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L19	Stable Compiler Version	Unresolved

IBA - Inconsistent Balance Allocation

Criticality	Medium
Location	Staking.sol#L145
Status	Unresolved

Description

The contract proceeds with the unstaking of the tokens and checks the lockup period defined by `poolLockupSeconds`. In the `_unstake` function, if a user unstakes before the lockup period has elapsed, the contract updates the `poolBalanceAllocation` by adding the `unstakeAmount` to it. Simultaneously, the contract also transfers the `unstakeAmount` back to the user. This creates an inconsistency as the `poolBalanceAllocation` is increased, suggesting that the pool has more tokens allocated, while the actual tokens are transferred back to the user. As a result, this could lead to inaccurate variable calculations.

```
    function _unstake(uint256 pid) internal {
        ...
        uint256 unstakeAmount = user.amount;
        uint256 rewardAmount =
user.amount.mul(poolInfo[pid].poolAPY).div(
            10000
        );

        if (block.timestamp - user.lastStakeTime <
poolInfo[pid].poolLockupSeconds) {
            poolInfo[pid].poolBalanceAllocation = poolInfo[pid]
                .poolBalanceAllocation
                .add(unstakeAmount);
            poolInfo[pid].poolRewardAllocation = poolInfo[pid]
                .poolRewardAllocation
                .add(rewardAmount);
        }
        ...

        poolInfo[pid].poolStakedBalance =
poolInfo[pid].poolStakedBalance.sub(
            user.amount
        );
        stakingToken.safeTransfer(address(msg.sender), unstakeAmount);
        user.amount = 0;
        ...
    }
```

Recommendation

It is recommended to reconsider the code implementation. Since the `unstakeAmount` is transferred back to the user, the `poolBalanceAllocation` should take this into consideration and be updated properly. Specifically, if tokens are being returned to the user, the `poolBalanceAllocation` should not be increased by the `unstakeAmount`. By doing this, the contract will maintain a consistent state between the `poolBalanceAllocation` and the actual tokens in the pool, reducing the risk of logical errors or vulnerabilities.

CR - Centralization Risk

Criticality	Minor / Informative
Location	Staking.sol#L77,216,225
Status	Unresolved

Description

The contract poses a centralization risk. Specifically, the `__Staking_init` function allows the contract owner to initialize critical parameters like the `stakingToken`, the initial PoolInfo for early pool data, as well as important time-related variables such as `_lifeTime` and `freezeTime`. This poses a significant risk to the overall security and decentralized nature of the contract.

In addition to setting the initial parameters, the contract owner also has the authority to transfer the `stakingToken` amount to the contract which will be used as rewards.

Furthermore, the contract owner can set any variable within the `PoolInfo` struct, enabling the owner to add new pools or update existing ones. The contract owner can modify key attributes such as `poolLockupSeconds`, `stakeMinAmount`, `stakeMaxAmount`, `poolAPY`, `poolBalanceAllocation`, and `poolRewardAllocation`. This level of control held by a single entity undermines the trustless and decentralized attributes generally expected of blockchain-based systems.

```
function __Staking_init(
        address _token,
        PoolInfo calldata _earlyPoolData,
        uint256 _lifeTime,
        uint256 freezeTime
    ) public initializer {
        stakingToken = IERC20(_token);
        pool0CreatedTime = block.timestamp;
        poolInfo.push(_earlyPoolData);
        pool0LifeTime = _lifeTime;
        poolFreezeTime = freezeTime;
        __Ownable_init();
        __UUPSUpgradeable_init();
    }

    function addPoolInfo(PoolInfo calldata _poolData) public
onlyOwner {
        poolInfo.push(_poolData);
    }

    function setPoolInfo(
        uint256 pid,
        PoolInfo calldata _poolData
    ) public validPID(pid) onlyOwner {
        poolInfo[pid] = _poolData;
    }
```

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.

MC - Missing Check

Criticality	Minor / Informative
Location	Staking.sol#L216,225
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

The contract is designed to allow the addition and modification of staking pools through `addPoolInfo` and `setPoolInfo` functions. These functions take a PoolInfo struct as an argument, and set the values for `poolBalanceAllocation` and `poolRewardAllocation` without any constraints or validation checks. While this provides maximum flexibility, it poses a considerable risk as the contract owner can set these variables to unreasonable values. Such an action could inadvertently disable the staking functionality for the users, leading to a lack of trust and possible financial consequences.

```
function addPoolInfo(PoolInfo calldata _poolData) public
onlyOwner {
        poolInfo.push(_poolData);
    }

    function setPoolInfo(
        uint256 pid,
        PoolInfo calldata _poolData
    ) public validPID(pid) onlyOwner {
        poolInfo[pid] = _poolData;
    }

    struct PoolInfo {
        ...
        uint256 poolBalanceAllocation;  // pool available
balance // tolal token amount that can be staked
        uint256 poolRewardAllocation;   // pool reward
allocation
        uint256 poolStakedBalance;      // pool stake balance
    }
```

Recommendation

The team is advised to properly check the variables according to the required specifications. It is recommended to incorporate additional checks before setting or updating the `poolBalanceAllocation` and `poolRewardAllocation` variables. One suggestion is to enforce a lower limit on the `poolBalanceAllocation` variable that cannot be less than the current `poolStakedBalance`. Similarly, the `poolRewardAllocation` should not be set lower than the reward allocation already used, preventing potential imbalances and ensuring the staking and unstaking mechanisms operate as expected. This will add an extra layer of security and robustness to the contract.

ILT - Inconsistent Lockup Times

Criticality	Minor / Informative
Location	Staking.sol#L148,154
Status	Unresolved

Description

The contract is designed to control the unstaking of tokens with the inclusion of lockup and freeze time mechanisms. Specifically, it utilizes the `poolFreezeTime` variable to prevent users from unstaking tokens before a specified duration has passed. Additionally, it uses `poolLockupSeconds` to determine if the necessary time has passed to release rewards to the staker. However the `poolLockupSeconds` can be set to a value less than `poolFreezeTime` .

If `poolFreezeTime` is set to be longer (e.g., 4 weeks), and `poolLockupSeconds` is set to be shorter, then the `if` statement `block.timestamp - user.lastStakeTime < poolInfo[pid].poolLockupSeconds` becomes meaningless. As a result, the user cannot unstake before the freeze time has elapsed, making the lockup time for rewards irrelevant, as it will have no practical implementation or effect.

```
function _unstake(uint256 pid) internal {
        ...
        require(block.timestamp - user.lastStakeTime >=
poolFreezeTime, "Pool Freeze Time");
        ...
        if (block.timestamp - user.lastStakeTime <
poolInfo[pid].poolLockupSeconds) {
            // recover pool balance and reward allocation when
unstake before lockup period
            poolInfo[pid].poolBalanceAllocation = poolInfo[pid]
                .poolBalanceAllocation
                .add(unstakeAmount);
            poolInfo[pid].poolRewardAllocation = poolInfo[pid]
                .poolRewardAllocation
                .add(rewardAmount);
        } else {
            // add reward when unstake after lockup period
            unstakeAmount = unstakeAmount.add(rewardAmount);
        }
```

Recommendation

The team is advised to properly check the variables according to the required specifications. It is recommended to introduce additional checks in the contract's logic to prevent the setting of `poolLockupSeconds` to a value that is less than `poolFreezeTime`. This can be accomplished with an assertion or require statement that validates the relationship between these two variables whenever they are set or updated.

RCI - Reward Calculation Inconsistency

Criticality	Minor / Informative
Location	Staking.sol#L128,150
Status	Unresolved

Description

The contract is designed to handle staking and unstaking of tokens with associated rewards. Within the `_stake` function, it calculates the `rewardAmount` based on the `poolAPY` at the time of staking and updates `poolRewardAllocation` accordingly. Similarly, in the `_unstake` function, the `rewardAmount` is again calculated based on the `poolAPY`, but this time at the moment of unstaking. However, the contract does not account for the possibility that the `poolAPY` may have changed between the staking and unstaking events.

This implementation can lead to inconsistencies and incorrect calculations in the `poolRewardAllocation`. For example, if the `poolAPY` has increased since the user staked their tokens, the unstake function would calculate a higher `rewardAmount` than was originally allocated during staking, which could result in an imbalance in the `poolRewardAllocation`.

```
function _stake(uint256 pid, uint256 amount) internal {
        ...
        uint256 rewardAmount =
amount.mul(poolInfo[pid].poolAPY).div(10000);
        ...
        poolInfo[pid].poolRewardAllocation = poolInfo[pid]
            .poolRewardAllocation
            .sub(rewardAmount);
        ...
    }

 function _unstake(uint256 pid) internal {
        ...
        uint256 rewardAmount =
user.amount.mul(poolInfo[pid].poolAPY).div(
            10000
        );

        if (block.timestamp - user.lastStakeTime <
poolInfo[pid].poolLockupSeconds) {
        ...
            poolInfo[pid].poolRewardAllocation = poolInfo[pid]
                .poolRewardAllocation
                .add(rewardAmount);
        } else {
            // add reward when unstake after lockup period
            unstakeAmount = unstakeAmount.add(rewardAmount);
        }
        ...

        stakingToken.safeTransfer(address(msg.sender),
unstakeAmount);
        ...
    }
```

Recommendation

It is recommended to add a separate state variable or mapping to keep track of the `rewardAmount` allocated to each user at the time of staking. This ensures that the same `rewardAmount` is used for both staking and unstaking actions, regardless of any changes in the `poolAPY` .

Additionally, consider updating the logic in the `_unstake` function to refer to the `rewardAmount` captured at the time of staking, rather than recalculating it based on the potentially changed `poolAPY`.

By doing these, the code can prevent the inconsistencies and potential imbalances in the `poolRewardAllocation`, making the contract's behavior more predictable and robust.

MSC - Missing Sanity Check

Criticality	Minor / Informative
Location	Staking.sol#L216,225
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

The contract is designed to allow the addition and modification of staking pools through `addPoolInfo` and `setPoolInfo` functions. These functions take a `PoolInfo` struct as an argument, which includes fields for `stakeMinAmount` and `stakeMaxAmount`. These two variables are intended to set the minimum and maximum staking amounts for a given pool. However, there is lack of validation checks to ensure that the `stakeMinAmount` is strictly less than or equal to the `stakeMaxAmount`. The absence of these checks introduces a risk where a pool could be initialized or modified with invalid bounds, potentially leading to logical errors in the staking mechanism, frustrated users, or even exploitation scenarios.

```
 function addPoolInfo(PoolInfo calldata _poolData) public
onlyOwner {
        poolInfo.push(_poolData);
    }

    function setPoolInfo(
        uint256 pid,
        PoolInfo calldata _poolData
    ) public validPID(pid) onlyOwner {
        poolInfo[pid] = _poolData;
    }

   struct PoolInfo {
        ...
        uint256 stakeMinAmount;          // stake minimum amount
        uint256 stakeMaxAmount;          // stake maximum amount
        ...
    }
```

Recommendation

The team is advised to properly check the variables according to the required specifications. It is recommended that the team incorporate validation checks within the `addPoolInfo` and `setPoolInfo` functions to ensure that `stakeMinAmount` is less than or equal to `stakeMaxAmount`. Such checks should be designed to revert transactions that do not meet these criteria, thereby avoiding the addition or modification of a pool with invalid staking bounds.

RNRM - Redundant No Reentrant Modifier

Criticality	Minor / Informative
Location	Staking.sol#L182,193
Status	Unresolved

Description

The contract uses the `nonReentrant` modifier to the `stake` and `unstake` functions, which suggests an intention to prevent potential reentrancy attacks. However, these functions exclusively deals with the `stakingToken` token, which is considered a trusted source within the contract.

Given that the `stakingToken` token is a trusted entity and no external address can be executed through the `stake()` and `unstake` functions, the risk of reentrancy vulnerabilities is effectively mitigated. Consequently, the usage of the `nonReentrant` modifier becomes redundant, adding unnecessary complexity to the codebase.

```
    function stake(
        uint256 pid,
        uint256 amount
    ) public virtual validPID(pid) validStakeAmount(pid,
amount) nonReentrant {
        _stake(pid, amount);
    }

    function unstake(uint256 pid) public virtual validPID(pid)
nonReentrant {
        _unstake(pid);
    }
```

Recommendation

To address this finding and enhance code simplicity and clarity, it is recommended to remove the unnecessary `nonReentrant` modifier from the `stake` and `unstake` functions. By removing the modifier, the code becomes more streamlined and easier to comprehend, reducing the gas consuption.

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	Staking.sol#L125,150
Status	Unresolved

Description

Variables can have misleading names if their names do not accurately reflect the value they contain or the purpose they serve. The contract uses some variable names that are too generic or do not clearly convey the information stored in the variable. Misleading variable names can lead to confusion, making the code more difficult to read and understand.

Specifically, the contract is using the variable `poolAPY` to calculate the `rewardAmount`. The name `poolAPY` suggests that it should represent the Annual Percentage Yield (APY) calculated based on the time that has passed. However, the `poolAPY` is a fixed percentage number that is set, rather than being dynamically calculated to reflect the true APY. This can be misleading for users who may assume that `rewardAmount` is calculated based on a true APY, potentially leading to incorrect expectations and calculations.

Additionally, the `poolFreezeTime` variable, which is used to enforce a minimum time before unstaking, can be updated by calling the `setPoolFreezeTime` function. This contradicts the documentation, which suggests that `poolFreezeTime` is a fixed number. As a result, `poolFreezeTime` does not represent a fixed number and can change, leading to potential inconsistencies and unexpected behavior.

```
uint256 rewardAmount =
amount.mul(poolInfo[pid].poolAPY).div(10000);
...
require(block.timestamp - user.lastStakeTime >= poolFreezeTime,
"Pool Freeze Time");
```

```
function setPoolFreezeTime(uint256 freezeTime) public onlyOwner
{
    poolFreezeTime = freezeTime;
}
```

Recommendation

It's always a good practice for the contract to contain variable names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

It is recommended to rename the variable `poolAPY` to better reflect its actual implementation and avoid confusion. This will make it clear that the `poolAPY` variable is not dynamically calculating the APY but is instead a fixed rate used for reward calculations. For `poolFreezeTime` variable, it is recommended to use a fixed number as suggested in the documentation, rather than allowing it to be updated dynamically.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	Staking.sol#L117
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are 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

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

            poolInfo[pid].poolStakedBalance =
poolInfo[pid].poolStakedBalance.add(
                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
```

EIS - Excessively Integer Size

Criticality	Minor / Informative
Location	Staking.sol#L30
Status	Unresolved

Description

The contract uses a bigger unsigned integer data type that the maximum size that is required. The contract is utilizing `uint256` data types for the variables `lastStakeTime`, `pid`, and `index`. According to the documentation, the maximum value for `pid` could be 7. This indicates that the `pid`, and `index` variables could be represented using a smaller data type, specifically `uint8` while `lastStakeTime` as `uint32`. By using an unsigned integer data type larger than necessary, the smart contract consumes more storage space and requires additional computational resources for calculations and operations involving these variables. This can result in higher transaction costs, longer execution times, and potential scalability bottlenecks.

```
uint256 lastStakeTime;
...
uint256 pid
...
uint256 index
```

Recommendation

To address the inefficiency associated with using an oversized unsigned integer data type, it is recommended to accurately determine the required size based on the range of values the variable needs to represent. The team is advised to use `uint32` instead of `uint256` for the variable `lastStakeTime`, and `uint8` for the variables `pid`, and `index`. By doing so, the smart contract can optimize its gas usage.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	Staking.sol#L200
Status	Unresolved

Description

The contract owner has the authority to claim all the balance of the contract. The owner may take advantage of it by calling the `adminWithdraw` function.

```
    function adminWithdraw(uint256 amount) public virtual
onlyOwner {
        stakingToken.safeTransfer(msg.sender, amount);
    }
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions. Some suggestions are:

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-sign wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.
- Renouncing the ownership will eliminate the threats but it is non-reversible.

IEI - Inconsistent Event Indexing

Criticality	Minor / Informative
Location	Staking.sol#L47
Status	Unresolved

Description

The contract is utilizing both `Stake` and `UnStake` events to emit important information about staking activities. The `Stake` event uses the `indexed` keyword for both the `user` and `stakedAmount` fields, allowing for easier and more efficient searches of these events within the blockchain. However, the `UnStake` event lacks the `indexed` keyword for the `stakedAmount` field, which leads to inconsistency in event indexing between the two types of staking activities. This discrepancy can make it more cumbersome for off-chain services to filter, and act upon these events.

```
    event Stake(
         uint256 pid,
         address indexed user,
         uint256 indexed stakedAmount
     );

     event UnStake(uint256 pid, address indexed user, uint256
stakedAmount);
```

Recommendation

It is recommended to align the `UnStake` event with the `Stake` event by adding the `indexed` keyword to the `stakedAmount` field in the `UnStake` event. This will ensure that both events can be easily and efficiently queried, providing a consistent interface for off-chain services.

IUC - Incorrect Unstake Check

Criticality	Minor / Informative
Location	Staking.sol#L147
Status	Unresolved

Description

A tautology is a logical statement that is always true, regardless of the values of its variables. A contradiction is a logical statement that is always false, regardless of the values of its variables.

Specifically, the contract allows users to unstake their tokens. The `_unstake` function utilizes a `require` statement to check if the user has a staked balance before proceeding with the unstaking process. The `require` statement checks if `user.amount` is greater than or equal to 0. However, this check is insufficient to prevent users who have already unstaked their tokens from invoking the unstake function again. As a result, a user with a zero balance (i.e., user.amount == 0) can still pass this require check and execute the unstaking logic, which could lead to unexpected behavior or vulnerabilities.

Using tautologies or contradictions can lead to unintended behavior and can make the code harder to understand and maintain. It is generally considered good practice to avoid tautologies and contradictions in the code.

```
require(user.amount >= 0, "User not staked")
```

Recommendation

The team is advised to carefully consider the logical conditions is using in the code and ensure that it is well-defined and make sense in the context of the smart contract. It is recommended to modify the `require` statement to strictly check if the user has a positive staked balance rather than allowing zero balances to pass the check. This will ensure that only users with an actual stake can invoke the unstake function.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	Staking.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.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	Staking.sol#L77,78,79,80,216,227
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.

```
function __Staking_init(
        address _token,
        PoolInfo calldata _earlyPoolData,
        uint256 _lifeTime,
        uint256 freezeTime
    ) public initializer {
        stakingToken = IERC20(_token);
        pool0CreatedTime = block.timestamp;
        poolInfo.push(_earlyPoolData);
        pool0LifeTime = _lifeTime;
        poolFreezeTime = freezeTime;
        __Ownable_init();
        __UUPSUpgradeable_init();
    }

...
```

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	Staking.sol#L209
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.

```
poolFreezeTime = freezeTime
```

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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	Staking.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

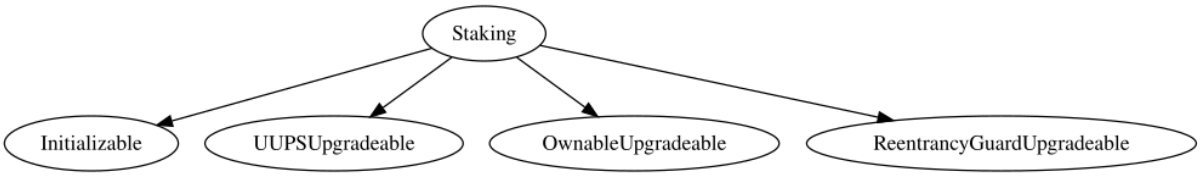
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest 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.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Staking	Implementation	Initializable, UUPSUpgradeable, OwnableUpgradeable, ReentrancyGuardUpgradeable		
	__Staking_init	Public	✓	initializer
	_authorizeUpgrade	Internal	✓	onlyOwner
	_stake	Internal	✓	
	_unstake	Internal	✓	
	stake	Public	✓	validPID validStakeAmount nonReentrant
	unstake	Public	✓	validPID nonReentrant
	adminWithdraw	Public	✓	onlyOwner
	setPoolFreezeTime	Public	✓	onlyOwner
	addPoolInfo	Public	✓	onlyOwner
	setPoolInfo	Public	✓	validPID onlyOwner
	readPoolInfo	Public		validPID
	getUserInfo	Public		-
	getUserTotalBalance	Public		-

	getPoolLength	Public		-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Staking
validStakeAmount
validPID
adminWithdraw
_authorizeUpgrade
__Staking_init
unstake
stake
setPoolFreezeTime
addPoolInfo
setPoolInfo
readPoolInfo
getUserInfo
getUserTotalBalance
getPoolLength
IERC20
__UUPSUpgradeable_init
__Ownable_init
_unstake
_stake
IERC20
safeTransfer
safeTransferFrom
pool0CreatedTime
add
amount
mul
unstakeAmount
add
totalBalance
add

Summary

Sphere 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