Cyberscope

Audit Report

aixCB

January 2025

Repository	https://github.com/aixcb-capital/aixcb-contracts
Commit	f745d989ade0fe8d79d4e19a0d2683c7451c94c2
Audited by	© cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/aixcb-capital/aixcb-contracts
Commit	f745d989ade0fe8d79d4e19a0d2683c7451c94c2

Test Deployments:

Contract Name	Deployment Address
AIXCBStaking	0xdB2584A46Db3bEE9426F1bb3Ae151C84E1338ed1
AIXCBLPStaking	0x523fFaF4Cf03E9B1350476364dB14629Dea69d7e

Audit Updates

Initial Audit	10 Jan 2025 https://github.com/cyberscope-io/audits/blob/main/aixcb/v1/audit.pdf
Corrected Phase 2	28 Jan 2025

Source Files

Filename	SHA256
AIXCBStaking.sol	d1623af7b0b8f0b414b6bb637ce0c0a7960de52af3b0ed8e11449700b072e35a
AIXCBLPStaking.sol	019e86ad80f8b698815d53db95d56850b87291432eec197a97c4a519eb2fdb24

Overview

The AIXCB project provides a robust staking solution that enables users to stake tokens and earn multi-token rewards while offering advanced features such as loyalty tracking, VIP status, and emergency controls. The two staking contracts, AIXCBLPStaking and AIXCBStaking, are designed for different token types but share the functionality of managing rewards distribution, enforcing security through role-based access control, and supporting user engagement metrics. These contracts follow the upgradeable proxy pattern for future enhancements and ensure safe user interactions via reentrancy protection and circuit breakers.

AIXCBLPStaking Contract

The AIXCBLPStaking contract is tailored for staking LP tokens, allowing users to earn multiple reward tokens. It implements features like real-time reward updates, loyalty tracking, based on staking activity. Users can withdraw their stakes at any time and claim rewards independently. Advanced security mechanisms include role-based access control, emergency mode with a fee for withdrawals, and circuit breakers to pause critical functions. This contract also maintains detailed user metrics, such as staking streaks and engagement scores, which contribute to the overall rewards system.

AIXCBStaking Contract

The AIXCBStaking contract provides a more flexible staking system for tokens, supporting multiple lock periods with varying reward rates. Users can select from predefined lock periods, and the contract tracks their loyalty metrics, including streaks, engagement scores and VIP status eligibility. Rewards are distributed based on staking duration and period-specific rates. The contract also supports emergency controls, allowing for safe withdrawal in critical scenarios. By handling multi-token rewards and incorporating custom loyalty mechanisms, this contract offers a comprehensive staking solution tailored to diverse user needs.

Roles

In the **AIXCBStaking** contract, the users can interact with the following functions:

Admin

- function startStaking()
- function _authorizeUpgrade(address newImplementation)
- function removeRewardToken(address token)
- function recoverERC20(address tokenAddress, uint256 tokenAmount)

Emergency Admin

- function toggleCircuitBreaker(bytes32 circuit)
- function enableEmergencyMode()
- function disableEmergencyMode()
- function emergencyWithdrawRewardToken(address token, uint256 amount, address recipient)
- function pause()
- function unpause()
- function stopStaking()

Reward Manager

- function fundRewardPool(uint256 periodIndex, address token, uint256 amount)
- function addRewardToken(address token)

Users

- function stake(StakeParams calldata params)
- function withdraw(uint256 periodIndex)
- function claimRewards(uint256 periodIndex)
- function emergencyWithdraw(uint256 periodIndex)

In the **AIXCBLPStaking** contract, the users can interact with the following functions:

Admin

- function _authorizeUpgrade(address newImplementation)
- function removeRewardToken(address token)
- function recoverERC20(address tokenAddress, uint256 tokenAmount)

Emergency Admin

- function enableEmergencyMode()
- function disableEmergencyMode()
- function toggleCircuitBreaker(bytes32 circuit)
- function pause()
- function unpause()

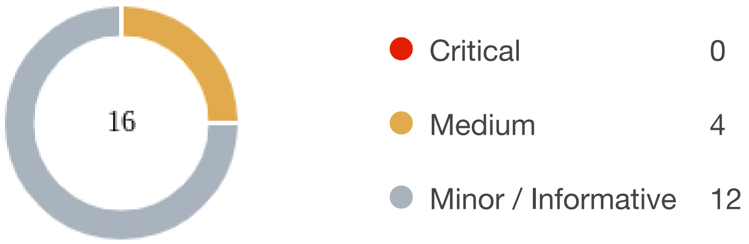
Reward Manager

- function addRewardToken(address token)
- function fundRewardPool(uint256 periodIndex, address token, uint256 amount)

Users

- function stake(uint256 amount)
- function withdraw(uint256 amount)
- function claimRewards()
- function emergencyWithdraw()

Findings Breakdown



Severity	Unresolved	Acknowledged	Resolved	Other
● Critical	0	0	0	0
● Medium	0	4	0	0
● Minor / Informative	0	12	0	0

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
🟠	IDV	Ineffective Deadline Validation	Acknowledged
🟠	MAC	Missing Access Control	Acknowledged
🟠	RRU	Redundant Reward Updates	Acknowledged
🟠	ROI	Reward Overwrite Issue	Acknowledged
⚪	CR	Code Repetition	Acknowledged
⚪	CCR	Contract Centralization Risk	Acknowledged
⚪	DF	Duplicate Functionality	Acknowledged
⚪	MIC	Misleading Initialization Comment	Acknowledged
⚪	MSV	Missing Stake Validation	Acknowledged
⚪	PAOR	Potential Arbitrage Opportunity Rewards	Acknowledged
⚪	PTAI	Potential Transfer Amount Inconsistency	Acknowledged
⚪	RTC	Redundant Type Casting	Acknowledged
⚪	RDI	Rewards Distribution Issue	Acknowledged
⚪	TSI	Tokens Sufficiency Insurance	Acknowledged

●	L04	Conformance to Solidity Naming Conventions	Acknowledged
●	L05	Unused State Variable	Acknowledged

IDV - Ineffective Deadline Validation

Criticality	Medium
Location	AIXCBStaking.sol#L274
Status	Acknowledged

Description

The contract is using the `deadline` parameter in the `stake` function, but it is only utilised in a conditional check (`if (block.timestamp > params.deadline) revert DeadlineExpired();`) to validate that the staking operation occurs before the specified deadline. However, this check does not add a meaningful safeguard since the `deadline` is passed through the parameters of the function, allowing a user to pass any value greater than `block.timestamp`. As a result, this value does not serve as a true deadline, given that the variable is not utilised elsewhere in the contract logic. Its inclusion without further purpose increases code complexity and may cause confusion for developers, as it appears redundant and unnecessary in the current implementation.

```
    function stake(StakeParams calldata params)
        external
        nonReentrant
        whenNotPaused
        notEmergency
        whenCircuitActive(STAKING_CIRCUIT)
        validPeriod(params.periodIndex)
        validAmount(params.amount)
    {
        if (block.timestamp > params.deadline) revert
DeadlineExpired();
```

Recommendation

It is recommended to evaluate whether the `deadline` parameter is necessary for enforcing meaningful constraints on staking operations. If it serves no broader purpose, consider removing the parameter to enhance clarity, streamline the code, and reduce gas costs. Alternatively, if the `deadline` is required, implement additional logic to ensure it is used effectively and cannot be manipulated by passing arbitrary values.

MAC - Missing Access Control

Criticality	Medium
Location	AIXCBLPStaking.sol#L157 AIXCBStaking.sol#L201
Status	Acknowledged

Description

The `initialize` functions can be frontrun during deployment, allowing administrative roles to be transferred to third parties not associated with the team. Such third parties would gain access to all the functions of the system.

```
   function initialize(
        address _stakingToken,
        address[] memory _initialRewardTokens,
        address _treasury
    ) external initializer {
        __ReentrancyGuard_init();
        __AccessControl_init();
        __Pausable_init();
        __UUPSUpgradeable_init();
        ...
    }
```

```
   function initialize(
        address _lpToken,
        address[] memory _initialRewardTokens,
        address _treasury
    ) external initializer {
    ...
    }
```

Recommendation

The team is advised to implement proper access controls to ensure that only authorized team members can call these functions.

RRU - Redundant Reward Updates

Criticality	Medium
Location	AIXCBLPStaking.sol#L211,249,275,403
Status	Acknowledged

Description

The contract contains the `_updateRewards` and `_claimRewards` functions, both of which internally call `_updateUserRewards`. This design results in `_updateUserRewards` being invoked twice each time `_updateRewards` and `_claimRewards` are both executed, as seen in the `stake`, `withdraw` and `claimRewards` functions. This redundant execution increases gas costs unnecessarily and introduces inefficiency in the reward update logic. Overlapping functionality between these functions adds complexity without providing additional value.

```
   function stake(uint256 amount)
        external
        nonReentrant
        whenNotPaused
        notInEmergencyMode
        circuitBreakerNotActive(STAKING_CIRCUIT)
    {
        if (amount == 0) revert ZeroAmount();

        UserStake storage userStake = userStakes[msg.sender];
        uint256 newTotalStake = uint256(userStake.stakedAmount)
+ amount;
        if (newTotalStake > MAX_STAKE_AMOUNT) revert
ExceedsMaxStake();

        _updateRewards(msg.sender);

        // If user already has a stake, claim pending rewards
first
        if (userStake.stakedAmount > 0) {
            _claimRewards(msg.sender);
        }
        ...
        }

    function withdraw(uint256 amount)
        external
        nonReentrant
        whenNotPaused
        notInEmergencyMode
        circuitBreakerNotActive(WITHDRAW_CIRCUIT)
    {
       ...

        _updateRewards(msg.sender);
        _claimRewards(msg.sender);
    ...
    }

    function claimRewards()
        external
        nonReentrant
        whenNotPaused
        notInEmergencyMode
        circuitBreakerNotActive(REWARD_CIRCUIT)
    {
        _updateRewards(msg.sender);
        _claimRewards(msg.sender);
    }
    function _updateRewards(address account) internal {
        for (uint256 i = 0; i < rewardTokens.length; i++) {
            _updateRewardPool(rewardTokens[i]);
            _updateUserRewards(account, rewardTokens[i]);
        }
    }

    function _claimRewards(address account) internal {
        for (uint256 i = 0; i < rewardTokens.length; i++) {
            _updateUserRewards(account, rewardTokens[i]);
        }
    }
```

Recommendation

It is recommended to refactor the `_updateRewards` and `_claimRewards` functions to eliminate redundant calls to `_updateUserRewards`. Consider consolidating the overlapping logic into a single function or restructuring the reward update and claiming process to ensure each user reward is updated only once per transaction. This optimisation will reduce gas costs and improve the overall efficiency of the contract.

ROI - Reward Overwrite Issue

Criticality	Medium
Location	AIXCBStaking.sol#L440
Status	Acknowledged

Description

The `fundRewardPool` function has a logic flaw when it is called after the reward period has expired (`periodFinish`), typically one year from the initial setup. In such cases, the function overwrites the existing `rewardRate` with a new value based solely on the newly added amount, without considering any unclaimed or remaining tokens from the previews time period.

This behaviour disregards leftover rewards that should still contribute to the distribution pool. Consequently, these remaining tokens are effectively excluded from the reward system, leading to potential inconsistencies in reward allocation, inaccurate calculations of total rewards, and a lack of fairness for users. These issues can result in user dissatisfaction and undermine the trustworthiness of the reward mechanism.

```
    function fundRewardPool(uint256 periodIndex, address token,
uint256 amount)
        external
        nonReentrant
        onlyRole(REWARD_MANAGER_ROLE)
        validPeriod(periodIndex)
        validRewardToken(token)
        validAmount(amount)
    {
        RewardPool storage pool =
rewardPools[periodIndex][token];
        _updateReward(address(0), periodIndex);

        IERC20Metadata(token).safeTransferFrom(msg.sender,
address(this), amount);
        pool.totalReward += amount;

        if (block.timestamp >= pool.periodFinish) {
            pool.rewardRate = amount / SECONDS_PER_YEAR;
            pool.periodFinish = block.timestamp +
SECONDS_PER_YEAR;
        } else {
            uint256 remaining = pool.periodFinish -
block.timestamp;
            uint256 leftover = remaining * pool.rewardRate;
            pool.rewardRate = (leftover + amount) / remaining;
        }

        pool.lastUpdateTime = block.timestamp;
    }
```

Recommendation

To address this issue, the function should be updated to account for any remaining tokens from the expired reward period when recalculating the `rewardRate`. By including these tokens in the calculations, the function will ensure accurate and fair reward distribution while maintaining the integrity of the total reward pool. Otherwise, consider handling the previous `rewardRate` value to maintain consistency and properly incorporate it into the logic for the updated reward distribution.

CR - Code Repetition

Criticality	Minor / Informative
Location	AIXCBLPStaking.sol#L225,257,345 AIXCBStaking.sol#L341,384
Status	Acknowledged

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

```
uint256 reward = userRewards[msg.sender][periodIndex][token];
if (reward > 0) {
    uint256 contractBalance =
IERC20Metadata(token).balanceOf(address(this));
    uint256 transferAmount = Math.min(reward, contractBalance);
    if (transferAmount > 0) {
        userRewards[msg.sender][periodIndex][token] -=
transferAmount;
        RewardPool storage pool =
rewardPools[periodIndex][token];
        pool.totalDistributed += transferAmount;
        IERC20Metadata(token).safeTransfer(msg.sender,
transferAmount);
        emit RewardPaid(msg.sender, token, transferAmount,
periodIndex);
    }
```

```
// Update reward debts for all tokens
for (uint256 i = 0; i < rewardTokens.length; i++) {
    address token = rewardTokens[i];
    userStake.rewardDebt[token] = (newTotalStake *
rewardPools[token].accumulatedPerShare) / PRECISION;
}
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L254,440,469,599,686 AIXCBLPStaking.sol#L304,464,485,509,543
Status	Acknowledged

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.

```
   function addRewardToken(address token)
        external
        ...
    {
       ...
    }

    function fundRewardPool(
        uint256 periodIndex,
        address token,
        uint256 amount
    ) {
    ...}

    function pause() external onlyRole(EMERGENCY_ADMIN_ROLE) {
        _pause();
    }

    function unpause() external onlyRole(EMERGENCY_ADMIN_ROLE)
{
        _unpause();
    }

    function enableEmergencyMode() external
onlyRole(EMERGENCY_ADMIN_ROLE) {
        emergencyMode = true;
        emit CircuitBreakerToggled("EMERGENCY_MODE", true,
block.timestamp);
    }

    function disableEmergencyMode() external
onlyRole(EMERGENCY_ADMIN_ROLE) {
        emergencyMode = false;
        emit CircuitBreakerToggled("EMERGENCY_MODE", false,
block.timestamp);
    }
    function startStaking() external onlyRole(ADMIN_ROLE) {
        require(areRewardPoolsFunded(), "Reward pools not
funded");
        _unpause();
        emit StakingStarted(block.timestamp);
    }

    function stopStaking() external
onlyRole(EMERGENCY_ADMIN_ROLE) {
        _pause();
        emit StakingStopped(block.timestamp);
    }
```

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.

DF - Duplicate Functionality

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L599,694
Status	Acknowledged

Description

The contract includes two functions, `pause` and `stopStaking` , that perform the same core functionality of pausing the contract by calling `_pause()` . The `stopStaking` function additionally emits an event, but this distinction does not justify maintaining two separate functions with overlapping behaviour. This duplication increases the codebase size unnecessarily, reduces maintainability, and may lead to confusion or inconsistency in usage.

```
   function pause() external onlyRole(EMERGENCY_ADMIN_ROLE) {
        _pause();
    }

    function stopStaking() external
onlyRole(EMERGENCY_ADMIN_ROLE) {
        _pause();
        emit StakingStopped(block.timestamp);
    }
```

Recommendation

It is recommended to consolidate the overlapping functionality into a single function or refactor the code to differentiate the purpose of these functions more clearly. If emitting an event is essential, consider adding a parameter to a single function to control whether the event should be emitted, or restructure the logic to avoid redundancy while preserving required functionality. Simplifying the code will improve clarity and maintainability.

MIC - Misleading Initialization Comment

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L222
Status	Acknowledged

Description

The contract includes a comment indicating that the `periodRates` variable is initialised for backward compatibility.However, the variable is not actually initialised in the code, making the comment misleading. As a result, this can cause confusion for developers and auditors, who may assume the variable is set or utilised somewhere in the contract logic when it is not. This discrepancy may also lead to unintended behaviour or overlooked issues during maintenance or upgrades.

```
// periodRates initialized but unused for backward
compatibility
```

Recommendation

Ensure that the comment accurately reflects the actual implementation. If `periodRates` is not meant to be initialised or used, update the comment to clarify its purpose, or remove the comment entirely to avoid confusion. If it should be initialised, ensure it is properly done in the code.

MSV - Missing Stake Validation

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L297,886
Status	Acknowledged

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 missing a validation check within the `upgradeStakePeriod` functionality. Specifically, unlike the `stake` function, the `upgradeStakePeriod` function does not verify whether the current block timestamp is greater than the `endTime` of the stake. This missing validation allows users to upgrade a stake period even after the `endTime` has passed. In contrast, the `stake` function enforces this restriction, ensuring that no additional stakes or modifications are allowed once the staking period has expired. The inconsistency between these two functions creates an exploitable loophole, enabling unauthorised modifications to expired stakes, potentially leading to unexpected behaviours or unintended reward distributions.

```
 function upgradeStakePeriod(uint256 currentPeriodIndex,
uint256 newPeriodIndex)
        external
        nonReentrant
        whenNotPaused
        notEmergency
        validPeriod(currentPeriodIndex)
        validPeriod(newPeriodIndex)
    {
        ...

        if (newStake.initialized) {
            newStake.amount += uint128(amount);
        } else {
            ...
        }
        ...
    }

  function stake(StakeParams calldata params)
        external
        nonReentrant
        whenNotPaused
        notEmergency
        whenCircuitActive(STAKING_CIRCUIT)
        validPeriod(params.periodIndex)
        validAmount(params.amount)
    {
        ...

        if (isNewStake) {
           ...
            }
        } else {
            if (block.timestamp >= userStake.endTime) revert
StakeExpired();
            endTime = userStake.endTime;
            userStake.amount =
uint128(uint256(userStake.amount) + params.amount);
        }
        ...
    }
```

Recommendation

The team is advised to properly check the variables according to the required specifications. It is recommended to include a validation check within the `upgradeStakePeriod` function to ensure that the current block timestamp does not exceed the `endTime` of the existing stake. This check should mirror the one implemented in the `stake` function to maintain consistency and prevent users from upgrading stakes after the staking period has expired.

PAOR - Potential Arbitrage Opportunity Rewards

Criticality	Minor / Informative
Location	AIXCBLPStaking.sol#L198,237
Status	Acknowledged

Description

The contract does not enforce a minimum time restriction between staking and withdrawing, which creates an arbitrage opportunity. Users can repeatedly stake and withdraw funds within the same reward period, leveraging this behaviour to manipulate the `accumulatedPerShare` value in their favour. By timing their actions strategically, users can inflate their rewards disproportionately, potentially disadvantaging other stakeholders and draining the reward pool faster than intended.

```
   function stake(uint256 amount)
        external
        nonReentrant
        whenNotPaused
        notInEmergencyMode
        circuitBreakerNotActive(STAKING_CIRCUIT)
    {
        if (amount == 0) revert ZeroAmount();

        UserStake storage userStake = userStakes[msg.sender];
        uint256 newTotalStake = uint256(userStake.stakedAmount)
+ amount;
        if (newTotalStake > MAX_STAKE_AMOUNT) revert
ExceedsMaxStake();

        _updateRewards(msg.sender);

        ...
    }

    function withdraw(uint256 amount)
        external
        nonReentrant
        whenNotPaused
        notInEmergencyMode
        circuitBreakerNotActive(WITHDRAW_CIRCUIT)
    {
        UserStake storage userStake = userStakes[msg.sender];
        if (userStake.stakedAmount == 0) revert NoStakeFound();
        if (amount == 0) revert ZeroAmount();
        if (amount > userStake.stakedAmount) revert
InvalidAmount();

        ...
    }
```

Recommendation

It is recommended to introduce a time-lock mechanism or a minimum holding period between staking and withdrawing to mitigate arbitrage opportunities. This mechanism could enforce a cooldown period during which rewards are either reduced or unavailable if users attempt to withdraw immediately after staking. Such restrictions will help ensure fair reward distribution and maintain the intended functionality of the staking system.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L305
Status	Acknowledged

Description

The `safeTransferFrom` function is 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(msg.sender, address(this),
params.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
```

RTC - Redundant Type Casting

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L285,299 AIXCBLPStaking.sol#L219
Status	Acknowledged

Description

The contract is employing redundant type casting in various parts of the code, such as casting between integer types like `uint256` to smaller types (`uint128` , `uint48` , or `uint32`) and vice versa, without a clear functional necessity. These castings add unnecessary operations to the code, potentially increasing gas costs and introducing risks of data truncation if not properly managed. The repeated use of such castings can also make the code less readable and harder to maintain, especially for developers or auditors unfamiliar with the original design intent.

```
amount: uint128(params.amount),
startTime: uint48(block.timestamp),
endTime: uint48(endTime),
periodIndex: uint32(params.periodIndex),
...
userStake.amount = uint128(uint256(userStake.amount) +
params.amount);
...
```

```
userStake.stakedAmount = uint128(newTotalStake);
userStake.initialStakeTime = userStake.initialStakeTime == 0 ?
uint48(block.timestamp) : userStake.initialStakeTime;
userStake.lastUpdateTime = uint48(block.timestamp);
...
```

Recommendation

It is recommended to review and refactor the code to minimise unnecessary type castings. Ensure that variables are defined with the appropriate data types at the time of declaration to avoid the need for frequent conversions. Only use type casting where absolutely necessary, and document its purpose to improve code clarity and reduce potential errors. This will enhance the contract's efficiency and readability while avoiding unintended side effects.

RDI - Rewards Distribution Issue

Criticality	Minor / Informative
Location	AIXCBLPStaking.sol#L380
Status	Acknowledged

Description

The contract contains a function `_updateUserRewards` that calculates and distributes rewards based on the `remainingRewards` in the reward pool. However, if the `remainingRewards` is insufficient or depleted, the function proceeds with state updates and allows stake, unstaking or claiming without transferring any rewards. This behaviour can result in users completing actions without receiving their entitled rewards, leading to a poor user experience and potential loss of trust in the system.

```
    function _updateUserRewards(address account, address token)
internal {
        UserStake storage userStake = userStakes[account];
        if (userStake.stakedAmount == 0) return;

        RewardPool storage pool = rewardPools[token];
        uint256 pending = (uint256(userStake.stakedAmount) *
pool.accumulatedPerShare) / PRECISION -
userStake.rewardDebt[token];

        if (pending > 0) {
            uint256 remainingRewards = pool.totalRewardAmount -
pool.totalDistributedAmount;
            uint256 transferAmount = pending > remainingRewards
? remainingRewards : pending;

            if (transferAmount > 0) {
                pool.totalDistributedAmount += transferAmount;
                IERC20Metadata(token).safeTransfer(account,
transferAmount);
                emit RewardsClaimed(account, token,
transferAmount, block.timestamp);
            }
        }

        userStake.rewardDebt[token] =
(uint256(userStake.stakedAmount) * pool.accumulatedPerShare) /
PRECISION;
    }
```

Recommendation

It is recommended to implement a mechanism to consider to pause staking and reward claiming functionalities when the reward pool is depleted. Alternatively, consider tracking pending rewards and storing them for users to claim in the future once the pool is replenished. These measures will ensure users are not left without rewards and maintain the integrity of the contract's reward distribution logic.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L240,254 AIXCBLPStaking.sol#L304,496
Status	Acknowledged

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.

```
   function _addRewardToken(address token) internal {
        ...
    }

    function addRewardToken(address token)
        external
        onlyRole(REWARD_MANAGER_ROLE)
        whenNotPaused
        nonReentrant
    {
        _addRewardToken(token);
    }

    }
```

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	AIXCBStaking.sol#L36,202,203,204,923 AIXCBLPStaking.sol#L158,159,160,556
Status	Acknowledged

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[] public PERIOD_RATES
address _stakingToken
address[] memory _initialRewardTokens
address _treasury
uint256[50] private __gap
address _lpToken
```

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/stable/style-guide.html#naming-conventions.

L05 - Unused State Variable

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L33,923 AIXCBLPStaking.sol#L556
Status	Acknowledged

Description

An unused state variable is a state variable that is declared in the contract, but is never used in any of the contract's functions. This can happen if the state variable was originally intended to be used, but was later removed or never used.

Unused state variables can create clutter in the contract and make it more difficult to understand and maintain. They can also increase the size of the contract and the cost of deploying and interacting with it.

```
uint256 private constant EMERGENCY_WITHDRAW_FEE = 2000
```

Recommendation

To avoid creating unused state variables, it's important to carefully consider the state variables that are needed for the contract's functionality, and to remove any that are no longer needed. This can help improve the clarity and efficiency of the contract.

Functions Analysis

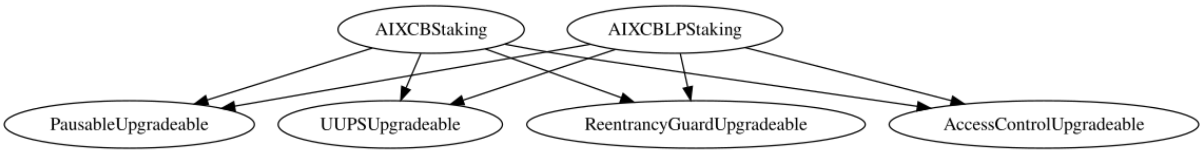
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
AIXCBStaking	Implementation	ReentrancyGuardUpgradeable, AccessControlUpgradeable, PausableUpgradeable, UUPSUpgradeable		
	initialize	External	✓	initializer
	_authorizeUpgrade	Internal	✓	onlyRole
	_addRewardToken	Internal	✓	
	addRewardToken	External	✓	onlyRole whenNotPaused nonReentrant
	stake	External	✓	nonReentrant whenNotPaused notEmergency whenCircuitActive validPeriod validAmount
	withdraw	External	✓	nonReentrant whenNotPaused notEmergency whenCircuitActive validPeriod
	claimRewards	External	✓	nonReentrant whenNotPaused notEmergency whenCircuitActive validPeriod
	_updateReward	Internal	✓	
	fundRewardPool	External	✓	nonReentrant onlyRole validPeriod

				validRewardToken validAmount
	toggleCircuitBreaker	External	✓	onlyRole
	enableEmergencyMode	External	✓	onlyRole
	disableEmergencyMode	External	✓	onlyRole
	emergencyWithdrawRewardToken	External	✓	nonReentrant onlyRole validRewardToken validAmount
	emergencyWithdraw	External	✓	nonReentrant
	getUserStake	External		-
	getUserTotalStake	External		-
	getTotalStaked	External		-
	pendingRewards	External		-
	pause	External	✓	onlyRole
	unpause	External	✓	onlyRole
	getStakersForPeriod	External		-
	getStakerCountForPeriod	External		-
	hasStakedInPeriod	External		-
	areRewardPoolsFunded	Public		-
	startStaking	External	✓	onlyRole
	stopStaking	External	✓	onlyRole
	getRewardRate	External		validRewardToken
	getRewardPoolInfo	External		validRewardToken
	getAPR	External		validPeriod validRewardToken

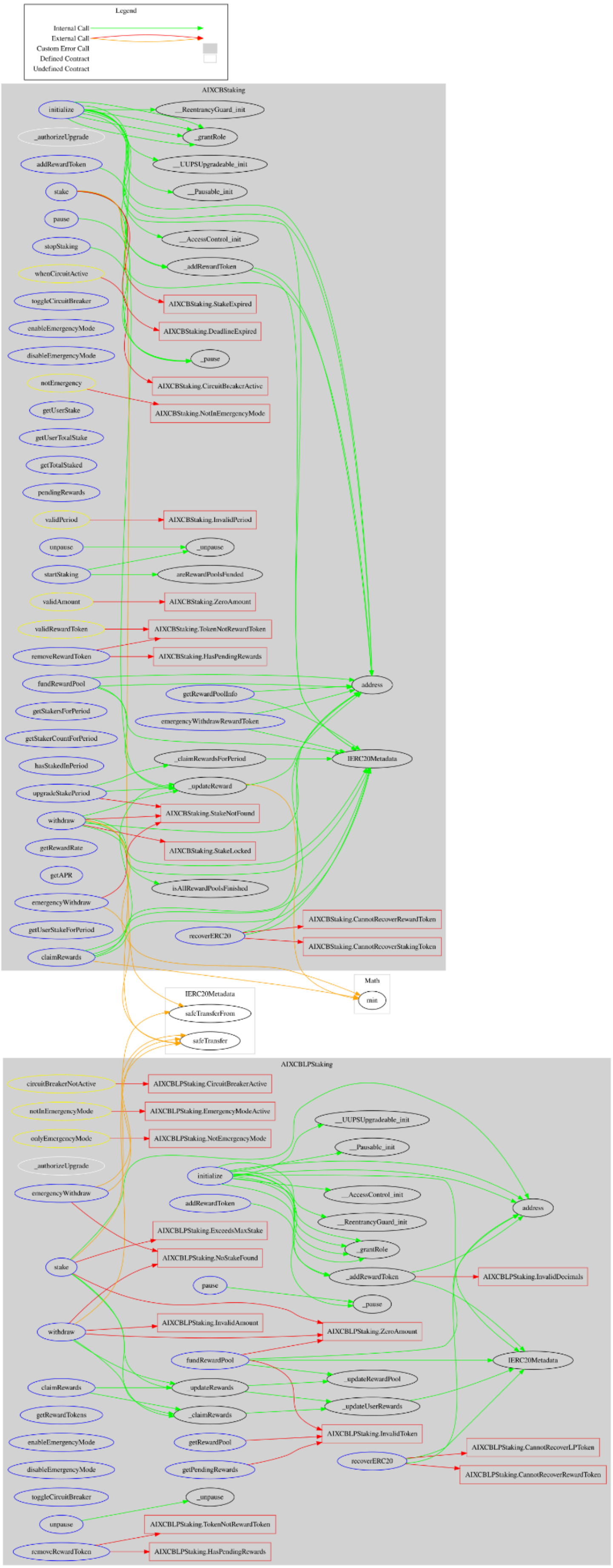
	removeRewardToken	External	✓	onlyRole whenNotPaused nonReentrant
	recoverERC20	External	✓	onlyRole nonReentrant
	getUserStakeForPeriod	External		validPeriod
	isAllRewardPoolsFinished	Public		-
	upgradeStakePeriod	External	✓	nonReentrant whenNotPaused notEmergency validPeriod validPeriod
	_claimRewardsForPeriod	Internal	✓	
AIXCBLPStaking	Implementation	ReentrancyGuardUpgradeable, AccessControlUpgradeable, PausableUpgradeable, UUPSUpgradeable		
	initialize	External	✓	initializer
	_authorizeUpgrade	Internal	✓	onlyRole
	stake	External	✓	nonReentrant whenNotPaused notInEmergencyMode circuitBreakerNotActive
	withdraw	External	✓	nonReentrant whenNotPaused notInEmergencyMode circuitBreakerNotActive
	claimRewards	External	✓	nonReentrant whenNotPaused notInEmergencyMode

				circuitBreakerNotActive
	emergencyWithdraw	External	✓	nonReentrant onlyEmergencyMode
	addRewardToken	External	✓	onlyRole whenNotPaused nonReentrant
	fundRewardPool	External	✓	onlyRole whenNotPaused nonReentrant
	_updateRewards	Internal	✓	
	_updateRewardPool	Internal	✓	
	_updateUserRewards	Internal	✓	
	_claimRewards	Internal	✓	
	getPendingRewards	External		-
	getRewardTokens	External		-
	getRewardPool	External		-
	enableEmergencyMode	External	✓	onlyRole
	disableEmergencyMode	External	✓	onlyRole
	toggleCircuitBreaker	External	✓	onlyRole
	pause	External	✓	onlyRole
	unpause	External	✓	onlyRole
	_addRewardToken	Internal	✓	
	removeRewardToken	External	✓	onlyRole whenNotPaused nonReentrant
	recoverERC20	External	✓	onlyRole nonReentrant

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
AIXCBStaking
initialize
__ReentrancyGuard_init
_authorizeUpgrade
_grantRole
addRewardToken
__UUPSUpgradeable_init
stake
__Pausable_init
pause
stopStaking
__AccessControl_init
whenCircuitActive
_addRewardToken
toggleCircuitBreaker
AIXCBStaking.StakeExpired
enableEmergencyMode
AIXCBStaking.DeadlineExpired
disableEmergencyMode
_pause
notEmergency
AIXCBStaking.CircuitBreakerActive
getUserStake
AIXCBStaking.NotInEmergencyMode
getUserTotalStake
getTotalStaked
pendingRewards
validPeriod
AIXCBStaking.InvalidPeriod
unpause
_unpause
startStaking
areRewardPoolsFunded
validAmount
AIXCBStaking.ZeroAmount
validRewardToken
AIXCBStaking.TokenNotRewardToken
removeRewardToken
AIXCBStaking.HasPendingRewards
fundRewardPool
address
getRewardPoolInfo
getStakersForPeriod
emergencyWithdrawRewardToken
getStakerCountForPeriod
_claimRewardsForPeriod
IERC20Metadata
hasStakedInPeriod
_updateReward
upgradeStakePeriod
AIXCBStaking.StakeNotFound
withdraw
getRewardRate
AIXCBStaking.StakeLocked
getAPR
isAllRewardPoolsFinished
emergencyWithdraw
AIXCBStaking.CannotRecoverRewardToken
getUserStakeForPeriod
recoverERC20
AIXCBStaking.CannotRecoverStakingToken
claimRewards
Math
min
IERC20Metadata
safeTransferFrom
safeTransfer
AIXCBLPStaking
circuitBreakerNotActive
AIXCBLPStaking.CircuitBreakerActive
notInEmergencyMode
AIXCBLPStaking.EmergencyModeActive
onlyEmergencyMode
AIXCBLPStaking.NotEmergencyMode
__UUPSUpgradeable_init
_authorizeUpgrade
__Pausable_init
initialize
emergencyWithdraw
__AccessControl_init
addRewardToken
address
__ReentrancyGuard_init
AIXCBLPStaking.ExceedsMaxStake
_grantRole
AIXCBLPStaking.NoStakeFound
stake
_addRewardToken
AIXCBLPStaking.InvalidDecimals
pause
_pause
AIXCBLPStaking.InvalidAmount
withdraw
AIXCBLPStaking.ZeroAmount
fundRewardPool
IERC20Metadata
_updateRewardPool
claimRewards
updateRewards
_updateUserRewards
getRewardTokens
_claimRewards
AIXCBLPStaking.InvalidToken
enableEmergencyMode
getRewardPool
AIXCBLPStaking.CannotRecoverLPToken
recoverERC20
disableEmergencyMode
getPendingRewards
AIXCBLPStaking.CannotRecoverRewardToken
toggleCircuitBreaker
_unpause
unpause
AIXCBLPStaking.TokenNotRewardToken
removeRewardToken
AIXCBLPStaking.HasPendingRewards

Summary

The AIXCBStaking and AIXCBLPStaking contracts implement flexible staking mechanisms for tokens and tokens, respectively. These contracts support multiple lock periods, reward distribution, loyalty tracking, and emergency controls tailored to their specific token types. This audit investigates security issues, business logic concerns, and potential improvements, focusing on role-based access control, reward management, and emergency mechanisms to ensure robust and secure staking functionality for both contracts. The team has acknowledged the findings.

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

cyberscope.io