Cyberscope

Audit Report

aixCB

January 2025

Repository	https://github.com/aixcb-capital/aixcb-contracts
Commit	5a87c404e509db98f68eec0e1dee821b3685b49b
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	5a87c404e509db98f68eec0e1dee821b3685b49b

Test Deployments:

Contract Name	Deployment Address
AIXCBStaking	0xFd845246EC58E0406D23fe24fD963E749A3C65d5
AIXCBLPStaking	0xc0F2904Edc41Fe2AeA3478D8E52b15C4cA0576d5

Audit Updates

Initial Audit	10 Jan 2025

Source Files

Filename	SHA256
AIXCBStaking.sol	f6cfebb3e8114178cf7248b7d16d18cf43ed7b4adeff40dac13e9aa118d281bc
AIXCBLPStaking.sol	37823c0d1d1c6c370886d7db1dab6b04b60734866e22196f8c4388827a853985

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()

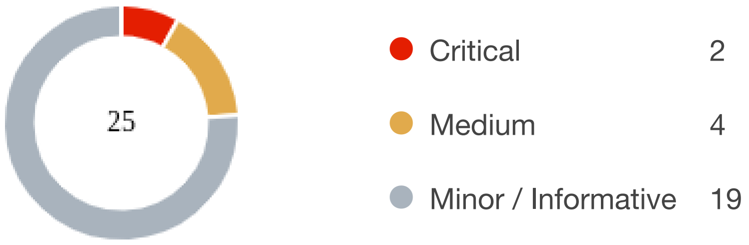
Reward Manager

- 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	2	0	0	0
●	Medium	4	0	0	0
●	Minor / Informative	19	0	0	0

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
🔴	IST	Incorrect Streak Tracking	Unresolved
🔴	USA	Unfair Streak Advantage	Unresolved
🟠	IRR	Incorrect Reward Rate	Unresolved
🟠	MAC	Missing Access Control	Unresolved
🟠	RRU	Redundant Reward Updates	Unresolved
🟠	UCV	Unused Code Variables	Unresolved
⚪	CR	Code Repetition	Unresolved
⚪	CCR	Contract Centralization Risk	Unresolved
⚪	DF	Duplicate Functionality	Unresolved
⚪	ITP	Inconsistent Token Precision	Unresolved
⚪	IRBW	Insufficient Rewards Block Withdrawals	Unresolved
⚪	LCO	Loop Code Optimization	Unresolved
⚪	MEE	Missing Events Emission	Unresolved
⚪	MVI	Missing VIP Implementation	Unresolved

●	MU	Modifiers Usage	Unresolved
●	PAOR	Potential Arbitrage Opportunity Rewards	Unresolved
●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	RLM	Redundant Loyalty Metrics	Unresolved
●	RTC	Redundant Type Casting	Unresolved
●	RDI	Rewards Distribution Issue	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved
●	UDV	Unused Declared Variable	Unresolved
●	UM	Unused Modifier	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L19	Stable Compiler Version	Unresolved

IST - Incorrect Streak Tracking

Criticality	Critical
Location	AIXCBLPStaking.sol#L435
Status	Unresolved

Description

The contract is incorrectly handling the logic for updating the `longestStreak` and `currentStreak` variables in the `_updateLoyaltyMetrics` function. Specifically, the `longestStreak` is updated whenever the `currentStreak` increases, rather than correctly tracking the longest consecutive streak of staking activity. Additionally, due to a logical flaw, the `else if` condition (`daysSinceLastUpdate > 1`) is never executed, preventing the `currentStreak` from being reset. This can result in crucial miscalculations, allowing the `currentStreak` to continuously increment regardless of user activity patterns, which invalidates the intended functionality of tracking loyalty accurately.

```
 function _updateLoyaltyMetrics(address user, uint256 amount,
bool isStaking) internal {
        ...
        if (isStaking) {
            stats.stakingPower += amount;

            if (stats.lastUpdateTime == 0) {
                stats.currentStreak = 1;
                stats.totalStakingDays = 1;
            } else {
                uint256 daysSinceLastUpdate = (currentTime -
stats.lastUpdateTime) / 1 days;
                if (daysSinceLastUpdate >= 1) {
                    stats.currentStreak++;
                    if (stats.currentStreak >
stats.longestStreak) {
                        stats.longestStreak =
stats.currentStreak;
                    }
                    stats.totalStakingDays +=
daysSinceLastUpdate;
                } else if (daysSinceLastUpdate > 1) {
                    stats.currentStreak = 1;
                    stats.totalStakingDays +=
daysSinceLastUpdate;
                }
            }
        } else {
            stats.stakingPower = stats.stakingPower > amount ?
stats.stakingPower - amount : 0;
        }

        ...
    }
```

Recommendation

It is recommended to thoroughly review and refactor the `_updateLoyaltyMetrics` function to ensure the `longestStreak` correctly represents the maximum consecutive streak of staking activity. Adjust the logic so that the `else if` condition is reachable and accurately resets the `currentStreak` when there is a significant gap in user activity. Additionally, consider adding unit tests to validate that the loyalty metrics function as intended across various scenarios. This will ensure accurate tracking and prevent further miscalculations.

USA - Unfair Streak Advantage

Criticality	Critical
Location	AIXCBLPStaking.sol#L448,478
Status	Unresolved

Description

The contract allows users who call `_updateLoyaltyMetrics` through other functions daily to gain an advantage over those who update after longer periods. This occurs because the `currentStreak` is incremented by one each day a user interacts with the contract, directly affecting the calculation of `engagementScore`. Users who interact daily accumulate higher `currentStreak` values, resulting in disproportionately greater rewards or benefits compared to users who do not update their metrics as frequently, despite potentially similar staking behaviour.

```
   function _updateLoyaltyMetrics(address user, uint256 amount,
bool isStaking) internal {
        ...

        if (isStaking) {
            stats.stakingPower += amount;

            if (stats.lastUpdateTime == 0) {
                stats.currentStreak = 1;
                stats.totalStakingDays = 1;
            } else {
                uint256 daysSinceLastUpdate = (currentTime -
stats.lastUpdateTime) / 1 days;
                if (daysSinceLastUpdate >= 1) {
                    stats.currentStreak++;
                    if (stats.currentStreak >
stats.longestStreak) {
                        stats.longestStreak =
stats.currentStreak;
                    }
                    stats.totalStakingDays +=
daysSinceLastUpdate;
                } else if (daysSinceLastUpdate > 1) {
                    stats.currentStreak = 1;
                    stats.totalStakingDays +=
daysSinceLastUpdate;
                }
            }
        } else {
            stats.stakingPower = stats.stakingPower > amount ?
stats.stakingPower - amount : 0;
        }

        stats.lastUpdateTime = currentTime;
        stats.engagementScore =
_calculateEngagementScore(user);
    ...
    }

    function _calculateEngagementScore(address user) internal
view returns (uint256) {
        LoyaltyStats storage stats = loyaltyStats[user];
        return (stats.stakingPower * stats.totalStakingDays *
(100 + stats.currentStreak)) / 100;
    }
```

Recommendation

It is recommended to revise the logic to ensure fair treatment of all users, regardless of the frequency of their interactions. Consider implementing a mechanism to normalise the `currentStreak` increment based on the actual duration of staking activity rather than daily updates. This adjustment will ensure that rewards or scores accurately reflect user behaviour over time, promoting fairness and reducing the incentive for unnecessary daily interactions.

IRR - Incorrect Reward Rate

Criticality	Medium
Location	AIXCBStaking.sol#L483
Status	Unresolved

Description

The contract sets an incorrect `rewardRate` in the `fundRewardPool` function. When the function is called, the `rewardRate` is recalculated based solely on the newly provided funding amount without considering the previously funded amount already stored in the reward pool. This approach overwrites the previous reward rate, potentially resulting in an inaccurate distribution of rewards. Such behaviour can lead to inconsistencies in reward allocation and user dissatisfaction, especially in cases of frequent funding.

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

        pool.rewardRate = amount / SECONDS_PER_YEAR;   // Raw
tokens per second, no scaling needed
        pool.lastUpdateTime = block.timestamp;
        pool.periodFinish = block.timestamp + SECONDS_PER_YEAR;
// Rewards distributed over a year

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

Recommendation

It is recommended to modify the `fundRewardPool` function to account for both the previous and newly added reward amounts when calculating the `rewardRate`. This can be achieved by incorporating the existing pool's total rewards into the calculation to ensure that the `rewardRate` reflects the actual cumulative rewards. Doing so will improve accuracy in reward distribution and maintain the integrity of the contract's reward mechanism.

MAC - Missing Access Control

Criticality	Medium
Location	AIXCBStaking.sol#L220 AIXCBLPStaking.sol#L173
Status	Unresolved

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#L227,290,366,425
Status	Unresolved

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.

UCV - Unused Code Variables

Criticality	Medium
Location	AIXCBStaking.sol#L199,288
Status	Unresolved

Description

The contract is using the `StakeParams` struct with the `minRate` variable and the `deadline` parameter in the `stake` function, but neither is meaningfully utilised within the code implementation. Specifically, the `StakeParams` struct declares the `minRate` variable, which is not referenced or utilised within the codebase. Additionally, the `deadline` variable is passed as a parameter through the `stake` function and included in a conditional check (`block.timestamp > params.deadline`). However, beyond this validation, it is not meaningfully utilised throughout the contract's logic. The inclusion of these variables without proper utilisation increases code complexity and may cause confusion for developers and users, as they appear redundant and unnecessary in the current implementation.

```
struct StakeParams {
    ...
    uint256 minRate;
}

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

Recommendation

It is recommended to evaluate the necessity of the `minRate` and `deadline` variables. If the code logic requires their use, ensure they are incorporated meaningfully within the implementation to achieve their intended purpose. Otherwise, consider removing them from the codebase to enhance clarity, reduce gas costs, and minimise unnecessary complexity.

CR - Code Repetition

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L358,411 AIXCBLPStaking.sol#L240,276
Status	Unresolved

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.

```
   function withdraw(uint256 periodIndex)
        external
        nonReentrant
        whenNotPaused
        notEmergency
        whenCircuitActive(WITHDRAWAL_CIRCUIT)
    {
        ...
        for (uint256 i = 0; i < rewardTokens.length; i++) {
            address token = rewardTokens[i];
            uint256 reward =
userRewards[msg.sender][periodIndex][token];
            if (reward > 0) {
                userRewards[msg.sender][periodIndex][token] =
0;
                RewardPool storage pool =
rewardPools[periodIndex][token];
                pool.totalDistributed += reward;
                IERC20Metadata(token).safeTransfer(msg.sender,
reward);
                emit RewardPaid(msg.sender, token, reward,
periodIndex);
            }
        }
        }

     function claimRewards(uint256 periodIndex)
        external
        nonReentrant
        whenNotPaused
        notEmergency
        whenCircuitActive(REWARDS_CIRCUIT)
    {
       ...
        for (uint256 i = 0; i < rewardTokens.length; i++) {
            address token = rewardTokens[i];
            uint256 reward =
userRewards[msg.sender][periodIndex][token];

            if (reward > 0) {
                userRewards[msg.sender][periodIndex][token] =
0;
                RewardPool storage pool =
rewardPools[periodIndex][token];
                pool.totalDistributed += reward;

                rewardAmounts[validTokenCount] = reward;
                tokensToTransfer[validTokenCount] = token;
                validTokenCount++;
            }
        }
        ...
        }
```

```
// 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#L272,483,580,587,609,707,784 AIXCBLPStaking.sol#L328,565,586,609,643
Status	Unresolved

Description

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

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

    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#L707,792
Status	Unresolved

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.

ITP - Inconsistent Token Precision

Criticality	Minor / Informative
Location	AIXCBLPStaking.sol#L402 AIXCBStaking.sol#L413
Status	Unresolved

Description

The contract includes support for multiple reward tokens but operates under the assumption that all tokens have the same decimal precision, without validating this assumption. This lack of validation can lead to discrepancies in reward calculations if any token deviates from the standard 18 decimals. Such an oversight can result in incorrect reward distributions, causing inefficiencies and potential user dissatisfaction.

```
    function _updateUserRewards(address account, address token) internal {
        ...

        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;
    }
```

```
     function _updateReward(address account, uint256
periodIndex) internal {
        for (uint256 i = 0; i < rewardTokens.length; i++) {
            address token = rewardTokens[i];
            RewardPool storage pool =
rewardPools[periodIndex][token];
            uint256 totalStaked =
totalStakedForPeriod[periodIndex];

            uint256 lastTimeRewardApplicable = block.timestamp
<
                pool.periodFinish
                ? block.timestamp
                : pool.periodFinish;

            if (
                totalStaked > 0 &&
                lastTimeRewardApplicable > pool.lastUpdateTime
            ) {
                uint256 timeDelta = lastTimeRewardApplicable -
                    pool.lastUpdateTime;
                uint256 rewardForPeriod = timeDelta *
pool.rewardRate;
                uint256 rewardPerTokenStored =
pool.accumulatedPerShare;
                rewardPerTokenStored +=
                    (rewardForPeriod * PRECISION) /
                    totalStaked;
                pool.accumulatedPerShare =
rewardPerTokenStored;
                pool.lastUpdateTime = lastTimeRewardApplicable;
            }
            ...
```

Recommendation

It is recommended to validate the decimal precision of each reward token during the initialisation or registration process. Ensure that all tokens conform to the standard 18 decimals. Alternatively, if tokens with varying decimals are supported, implement a mechanism to normalise their values to maintain consistency in reward calculations. This will ensure accurate and fair reward distribution across all reward tokens.

IRBW - Insufficient Rewards Block Withdrawals

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L365
Status	Unresolved

Description

The contract's withdrawal mechanism is vulnerable to a scenario where users may be unable to withdraw their initial stake if the reward tokens held by the contract are insufficient to cover the owed rewards. This is due to the reliance on `safeTransfer` for transferring rewards and stake amounts, which reverts the transaction if the contract lacks the required reward tokens. As a result, users' ability to retrieve their staked amount becomes dependent on the availability of adequate reward tokens, leading to potential disruptions in user withdrawals and dissatisfaction.

```
 function withdraw(uint256 periodIndex)
    external
    nonReentrant
    whenNotPaused
    notEmergency
    whenCircuitActive(WITHDRAWAL_CIRCUIT)
  {
    ...
    _updateReward(msg.sender, periodIndex);
    for (uint256 i = 0; i < rewardTokens.length; i++) {
      address token = rewardTokens[i];
      uint256 reward =
userRewards[msg.sender][periodIndex][token];
      if (reward > 0) {
        userRewards[msg.sender][periodIndex][token] =
0;
        RewardPool storage pool =
rewardPools[periodIndex][token];
        pool.totalDistributed += reward;
        IERC20Metadata(token).safeTransfer(msg.sender,
reward);
        emit RewardPaid(msg.sender, token, reward,
periodIndex);
      }
    }

    uint256 amount = userStakeData.amount;
    totalStakedForPeriod[periodIndex] -= amount;
    _totalUserStake[msg.sender] -= amount;

    delete userStakes[msg.sender][periodIndex];

    stakingToken.safeTransfer(msg.sender, amount);

    ...
  }
```

Recommendation

Consider allowing users to withdraw their staked amount even if the contract lacks sufficient rewards to distribute. This can be achieved by separating the logic for transferring rewards from the logic for releasing the staked amount. If adequate rewards are unavailable, users should still be able to remove their staked amount without receiving the full reward.

This approach would improve user experience and ensure that stake withdrawals are not entirely blocked by insufficient rewards.

LCO - Loop Code Optimization

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L426
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 contract uses two separate `for` loops within the `claimRewards` function to handle reward distribution. The first loop collects valid reward tokens and amounts, while the second loop performs the actual transfers and emits events. This design introduces unnecessary iterations, leading to higher gas consumption and inefficiency in the execution of the function. By consolidating the logic into a single loop, the contract could save gas and simplify the reward claim process without affecting functionality.

```
   function claimRewards(uint256 periodIndex)
        external
        nonReentrant
        whenNotPaused
        notEmergency
        whenCircuitActive(REWARDS_CIRCUIT)
    {
        ...
        for (uint256 i = 0; i < rewardTokens.length; i++) {
            address token = rewardTokens[i];
            uint256 reward =
userRewards[msg.sender][periodIndex][token];

            if (reward > 0) {
                userRewards[msg.sender][periodIndex][token] =
0;
                RewardPool storage pool =
rewardPools[periodIndex][token];
                pool.totalDistributed += reward;

                rewardAmounts[validTokenCount] = reward;
                tokensToTransfer[validTokenCount] = token;
                validTokenCount++;
            }
        }

        for (uint256 i = 0; i < validTokenCount; i++) {
            address token = tokensToTransfer[i];
            uint256 amount = rewardAmounts[i];

            require(
                IERC20Metadata(token).balanceOf(address(this))
>= amount,
                "Insufficient reward balance"
            );

            IERC20Metadata(token).safeTransfer(msg.sender,
amount);
            emit RewardPaid(msg.sender, token, amount,
periodIndex);
        }
    }
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

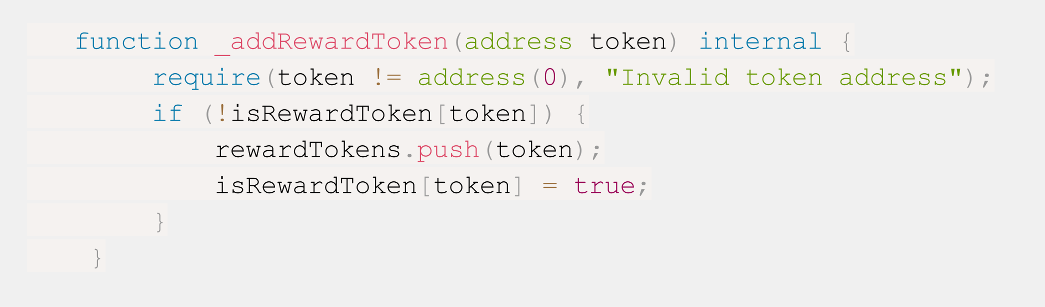
It is recommended to refactor the `claimRewards` function to combine the logic of the two `for` loops into a single loop. This would reduce the number of iterations required and optimise gas usage. Ensure that all necessary checks and operations, such as transferring rewards and emitting events, are performed within the consolidated loop for efficiency and clarity.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L259
Status	Unresolved

Description

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



```
    function _addRewardToken(address token) internal {
        require(token != address(0), "Invalid token address");
        if (!isRewardToken[token]) {
            rewardTokens.push(token);
            isRewardToken[token] = true;
        }
    }
```

Recommendation

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

MVI - Missing VIP Implementation

Criticality	Minor / Informative
Location	AIXCBLPStaking.sol#L20
Status	Unresolved

Description

The contract mentions tracking "loyalty and VIP status" in its comments, suggesting that a VIP system is an integral feature. However, while the contract tracks metrics like `engagementScore` and calculates VIP-related statuses, there is no meaningful implementation or external functionality to utilise this VIP status. This creates a misleading expectation of a VIP feature that does not actually exist, potentially confusing developers and users.

```
Track loyalty and VIP status
```

Recommendation

It is recommended to either implement meaningful functionality for the VIP status feature, such as exclusive benefits or access tied to the calculated metrics, or remove references to the VIP system if it is not planned for future use. This will ensure the contract aligns with its stated purpose and avoids misleading stakeholders.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L488,611,904
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(isRewardToken[token], "Token not accepted as reward");
...
require(isRewardToken[token], "Not a reward token");
...
if (isRewardToken[tokenAddress]) revert
CannotRecoverRewardToken();
...
```

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.

PAOR - Potential Arbitrage Opportunity Rewards

Criticality	Minor / Informative
Location	AIXCBLPStaking.sol#L214,256
Status	Unresolved

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#L321
Status	Unresolved

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`

RLM - Redundant Loyalty Metrics

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L510
Status	Unresolved

Description

The contract calculates and tracks metrics such as `engagementScore` and `VIP status` within the `_updateLoyaltyMetrics` function. However, these metrics lack meaningful utilisation within the current implementation, as there are no external or additional functionalities that rely on these values to produce tangible results. The presence of these unused metrics can be misleading, suggesting functionality that does not exist, and adds unnecessary complexity to the code.

```
  function _updateLoyaltyMetrics(
        address user,
        uint256 amount,
        uint256 periodIndex,
        bool isNewStake
    ) internal {
        LoyaltyStats storage stats = loyaltyStats[user];

        uint256 stakingPower = _calculateStakingPower(amount,
periodIndex);
        stats.stakingPower = isNewStake ? stats.stakingPower +
stakingPower : stats.stakingPower - stakingPower;

        if (isNewStake) {
            stats.currentStreak++;
            stats.longestStreak = Math.max(stats.currentStreak,
stats.longestStreak);
            stats.totalStakingDays += lockPeriods[periodIndex]
/ 1 days;
        } else {
            stats.currentStreak = 0;
        }

        stats.engagementScore =
_calculateEngagementScore(user);
        stats.lastUpdateTime = block.timestamp;

        _updateVIPStatus(user);

        emit LoyaltyMetrics(
            user,
            stats.stakingPower,
            stats.currentStreak,
            stats.totalStakingDays,
            stats.engagementScore,
            block.timestamp
        );
    }
```

Recommendation

It is recommended to evaluate whether the `engagementScore` and `VIP status` metrics are required for future external functionality or integration. If these metrics are not intended to be used meaningfully, consider removing them from the contract to simplify the code and avoid confusion. Eliminating unused features will improve code readability, reduce gas costs, and minimise potential maintenance overhead.

RTC - Redundant Type Casting

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L301,315 AIXCBLPStaking.sol#L235
Status	Unresolved

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#L402
Status	Unresolved

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#L259
Status	Unresolved

Description

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

```
   function _addRewardToken(address token) internal {
        require(token != address(0), "Invalid token address");
        if (!isRewardToken[token]) {
            rewardTokens.push(token);
            isRewardToken[token] = true;
        }
    }

...
    function claimRewards(uint256 periodIndex)
        external
        nonReentrant
        whenNotPaused
        notEmergency
        whenCircuitActive(REWARDS_CIRCUIT)
    {
    ....
        require(
            IERC20Metadata(token).balanceOf(address(this)) >=
amount,
            "Insufficient reward balance"
        );
    ...
    }
```

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.

UDV - Unused Declared Variable

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L241
Status	Unresolved

Description

The contract is declaring the `PERIOD_RATES` variable within the `initialize` function but does not utilise it in any part of the code. This results in unnecessary complexity and potential confusion, as the variable appears redundant and lacks a defined purpose within the current implementation. Variables like this can mislead developers and auditors, suggesting functionality that is absent, while also introducing a minor increase in gas usage during the function's execution without contributing to the contract's overall functionality.

```
    function initialize(
        address _stakingToken,
        address[] memory _initialRewardTokens,
        address _treasury
    ) external initializer {
        ...
        PERIOD_RATES = [200, 600, 1000];
        ...
        }
```

Recommendation

It is recommended to assess whether the `PERIOD_RATES` variable is intended to play a role in the contract's logic. If so, integrate it into the relevant code sections to serve its intended purpose. If it is not required, consider removing the variable entirely to simplify the code, improve readability, and reduce unnecessary gas costs.

UM - Unused Modifier

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L735
Status	Unresolved

Description

A modifier is a special type of function in Solidity that can be used to modify the behavior of other functions. When a modifier is applied to a function, it can perform certain checks or modifications to the input parameters or state variables of the function before it is executed.

An unused modifier means that there is a defined modifier in the contract code that is not actually being used in any of the functions. This can be an indication of a mistake or oversight in the development of the contract. It is generally good practice to remove any unused code in a smart contract to improve its readability and reduce the potential for bugs or security vulnerabilities.

```
    modifier onlyVIP() {
        if (!isVIP[msg.sender]) {
            revert NotVIP();
        }
        _;
    }
```

Recommendation

To avoid creating unused modifiers, it's important to carefully consider the modifiers 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.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L33,221,222,223,911 AIXCBLPStaking.sol#L174,175,176,656
Status	Unresolved

Description

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

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

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

```
uint256[] 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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	AIXCBStaking.sol#L2 AIXCBLPStaking.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 ^0.8.20;
```

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

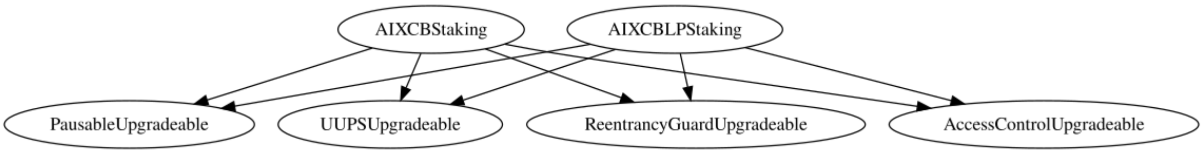
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
AIXCBStaking	Implementation	ReentrancyGuardUpgradeable, AccessControlUpgradeable, PausableUpgradeable, UUPSUpgradeable		
	initialize	External	✓	initializer
	_authorizeUpgrade	Internal	✓	onlyRole
	_addRewardToken	Internal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

	_updateLoyaltyMetrics	Internal	✓	
	_calculateStakingPower	Internal		
	_calculateEngagementScore	Internal		
	_updateVIPStatus	Internal	✓	
	toggleCircuitBreaker	External	✓	onlyRole
	enableEmergencyMode	External	✓	onlyRole
	disableEmergencyMode	External	✓	onlyRole
	emergencyWithdrawRewardToken	External	✓	nonReentrant onlyRole
	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		-
	getRewardPoolInfo	External		-

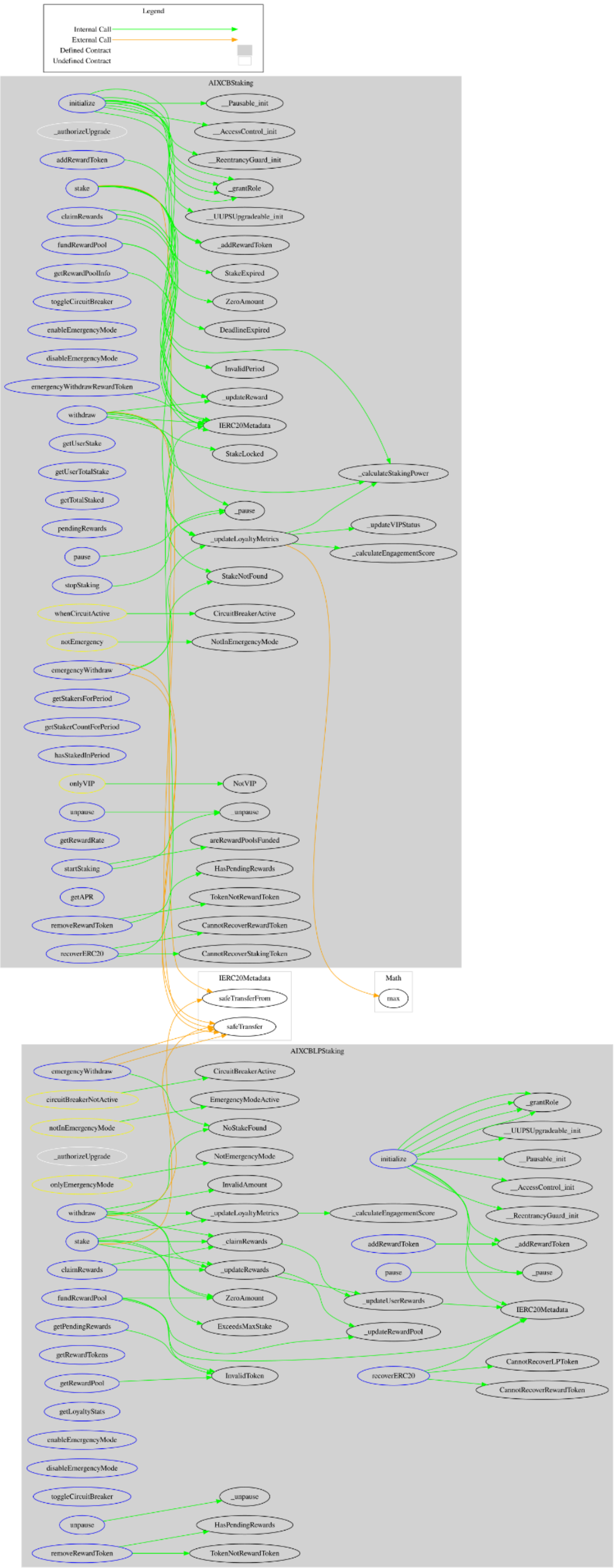
	getAPR	External		-
	removeRewardToken	External	✓	onlyRole whenNotPaused nonReentrant
	recoverERC20	External	✓	onlyRole nonReentrant
AIXCBLPStaking	Implementation	ReentrancyGuardUpgradeable, AccessControlUpgradeable, PausableUpgradeable, UUPSUpgradeable		
	initialize	External	✓	initializer
	_authorizeUpgrade	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	✓	
	_updateLoyaltyMetrics	Internal	✓	
	_calculateEngagementScore	Internal		
	getPendingRewards	External		-
	getRewardTokens	External		-
	getRewardPool	External		-
	getLoyaltyStats	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
Defined Contract
Undefined Contract
AIXCBStaking
initialize
__Pausable_init
_authorizeUpgrade
__AccessControl_init
addRewardToken
__ReentrancyGuard_init
stake
_grantRole
claimRewards
__UUPSUpgradeable_init
fundRewardPool
_addRewardToken
getRewardPoolInfo
StakeExpired
toggleCircuitBreaker
ZeroAmount
enableEmergencyMode
DeadlineExpired
disableEmergencyMode
InvalidPeriod
emergencyWithdrawRewardToken
_updateReward
withdraw
IERC20Metadata
getUserStake
StakeLocked
getUserTotalStake
_calculateStakingPower
getTotalStaked
_pause
pendingRewards
_updateVIPStatus
_updateLoyaltyMetrics
pause
_calculateEngagementScore
stopStaking
StakeNotFound
whenCircuitActive
CircuitBreakerActive
notEmergency
NotInEmergencyMode
emergencyWithdraw
getStakersForPeriod
getStakerCountForPeriod
hasStakedInPeriod
onlyVIP
NotVIP
unpause
_unpause
getRewardRate
areRewardPoolsFunded
startStaking
HasPendingRewards
getAPR
TokenNotRewardToken
removeRewardToken
CannotRecoverRewardToken
recoverERC20
CannotRecoverStakingToken
IERC20Metadata
safeTransferFrom
safeTransfer
Math
max
AIXCBLPStaking
emergencyWithdraw
CircuitBreakerActive
circuitBreakerNotActive
EmergencyModeActive
_grantRole
notInEmergencyMode
NoStakeFound
__UUPSUpgradeable_init
_authorizeUpgrade
NotEmergencyMode
initialize
__Pausable_init
onlyEmergencyMode
InvalidAmount
__AccessControl_init
withdraw
_updateLoyaltyMetrics
_calculateEngagementScore
__ReentrancyGuard_init
stake
_claimRewards
addRewardToken
_addRewardToken
claimRewards
_updateRewards
pause
_pause
fundRewardPool
ZeroAmount
_updateUserRewards
IERC20Metadata
getPendingRewards
ExceedsMaxStake
_updateRewardPool
getRewardTokens
CannotRecoverLPToken
getRewardPool
InvalidToken
recoverERC20
getLoyaltyStats
CannotRecoverRewardToken
enableEmergencyMode
disableEmergencyMode
toggleCircuitBreaker
_unpause
unpause
HasPendingRewards
removeRewardToken
TokenNotRewardToken

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.

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