Audit Report

Tea-Fi

September 2024

SHA256 b70015f535f8c13dea522e79e10e1b779ae22422ded827bff4cb4ebb5017aa6d

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

Testing Deploy	https://testnet.bscscan.com/address/0x4138F2e9040e78519dF8246839cBdb001978A22c

Audit Updates

Initial Audit	26 Sep 2024

Source Files

Filename	SHA256
TeaStaking.sol	b70015f535f8c13dea522e79e10e1b779ae22422ded827bff4cb4ebb5017aa6d
SignatureHandler.sol	58918d07e9840ea3d76c7b832feb650b8502232e0542b631eab069c66eaeeb46
interfaces/ITeaVesting.sol	eedb14592f8febc941657cb2d79d638a6ba91305c1af2a87a06103cd847200b5
interfaces/ITeaStaking.sol	cae076666a5f825829f3d9e88e16ca16a62c8c9dfe35dac0ff7b0601b4bc2311
interfaces/IStruct.sol	97840d3f62800d7f8822342c44cd14030d02c40820b663d4f3d98e8c0f71306d

Audit Scope

The staking process heavily depends on the TeaVesting contract, as it plays a crucial role not only in determining users' vesting allocations for staking but also in completing the withdrawal functionality. The current contract calls and interacts with the TeaVesting contract to validate vesting amounts and to facilitate the unlocking of rewards during the withdrawal process. Any inaccurate data or malfunction within the TeaVesting contract can halt the correct execution of this contract, potentially leading to blocked withdrawals or incorrect reward calculations. Therefore, it is highly recommended that the team deploy and utilize a TeaVesting contract that has been thoroughly audited to ensure the overall integrity of the system and prevent any disruptions or vulnerabilities in the staking and withdrawal processes.

Overview

The `TeaStaking` contract is designed to facilitate the staking of Tea tokens and presale tokens, allowing users to earn allocation rewards and manage their stakes. It enables users to stake their tokens to earn rewards over time, withdraw their tokens after a certain period, and ensures the distribution of rewards based on specific rules. The contract is equipped with mechanisms to handle various scenarios, such as staking with vesting allocations, managing VIP stakes, and ensuring the security and integrity of the staking process.

Stake Functionality

The `stake` function allows users to stake their Tea tokens or use their vested tokens allocations from the TeaVesting contract. Users can stake multiple tokens in a single transaction, provided the token is valid and the user hasn't already staked it. If a user stakes more than a predefined amount, they are granted VIP status, and their tokens are locked for a year. The function updates the total staked tokens and records the new stake with a unique ID, maintaining a record of the user's stakes and associated details.

Unstake Functionality

The `unstake` function enables users to initiate the process of unstaking their tokens. It first verifies the authenticity of the unstake request using a signature and then processes the unstake operation. If the staked tokens are not Tea tokens, they are immediately available for withdrawal. However, if the staked tokens are Tea tokens, a cooldown period of two weeks is applied before the user can withdraw them. This function also updates the user's stake information and the total staked tokens in the contract.

Withdraw Functionality

The `withdraw` function allows users to claim their staked tokens and rewards after the claim cooldown period has passed, in case it is applied. If the user attempts to withdraw Tea tokens before the cooldown period is over, the function reverts the transaction. For other tokens, the withdrawal is processed immediately. During withdrawal, the function transfers the staked tokens and calculated rewards to the user, updates the contract's records, and emits a withdrawal event.

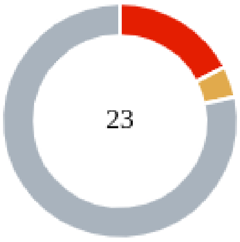
Rewards Distribution Functionality

Rewards are distributed based on the `updateRewardPerShare` function, which calculates the rewards per staked token. The contract continuously updates the rewards per share based on the passage of time and the total staked tokens. The rewards are then distributed to stakers proportionally based on their staked amount. This mechanism ensures that all users are fairly rewarded according to the duration and amount of their stakes.

DEFAULT_ADMIN_ROLE Functionalities

The `DEFAULT_ADMIN_ROLE` holds significant authority within the contract, including the ability to initialize the staking process and perform emergency withdrawals. During initialization, the admin sets the total allocation and the start time for reward distribution. The admin can also withdraw all tea tokens from the contract in emergency situations, ensuring the security and management of the staking process.

Findings Breakdown



Critical	4
Medium	1
Minor / Informative	18

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	4	0	0	0
Medium	1	0	0	0
Minor / Informative	18	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	IPH	Inconsistent Parameter Handling	Unresolved
	TSI	Tokens Sufficiency Insurance	Unresolved
	UOD	Unchecked Off-Chain Data	Unresolved
	VSV	Vulnerable Signature Verification	Unresolved
	ITV	Inconsistent Token Valuation	Unresolved
	RIC	Redundant If Check	Unresolved
	DPHI	Decimal Precision Handling Inconsistency	Unresolved
	IPT	Immutable Presale Tokens	Unresolved
	IVT	Incorrect Vesting Transfer	Unresolved
	ILO	Inefficient Loop Operations	Unresolved
	MSE	Misleading Staking Error	Unresolved
	MVN	Misleading Variable Naming	Unresolved
	MC	Missing Check	Unresolved
	ODV	Operator Dependency Vulnerability	Unresolved

●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	SVMC	Signature Validation Missing ChainID	Unresolved
●	OCTD	Transfers Contract's Tokens	Unresolved
●	UVC	Unfavorable VIP Conditions	Unresolved
●	UEE	Unnecessary Event Emission	Unresolved
●	UTT	Unoptimized Token Transfers	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L16	Validate Variable Setters	Unresolved

IPH - Inconsistent Parameter Handling

Criticality	Critical
Location	TeaStaking.sol#L120
Status	Unresolved

Description

The contract is using the `stake` function that accepts `_tokens` and `_amounts` as arrays but uses `_offChain` as a single structure instead of an array. This inconsistency in parameter handling can lead to issues, especially when staking multiple tokens in a single transaction. If multiple off-chain transactions or data points are needed to be processed for each token staked, the current function structure does not support this, potentially resulting in failed transactions or incorrect staking behavior.

```
   function stake(address[] calldata _tokens, uint256[] calldata
_amounts, OffChainStruct calldata _offChain)
        external
    {
        if (!checkStakingStart()) revert StakingNotStarted();
        if (_tokens.length != _amounts.length) revert
InvalidArrayLengths();
        address user = _msgSender();

        for (uint256 i = 0; i < _tokens.length; i++) {
            address _token = _tokens[i];
            uint256 _amount = _amounts[i];
            ...
ITeaVesting(teaVesting).transferOwnerOffChain(_offChain);
            }

            ...
            userIds[user].push(newId);

            emit Staked(user, _token, _amount);
        }
    }
```

Recommendation

It is recommended to modify the function to accept `_offChain` as an array, with each element corresponding to the respective `_tokens` and `_amounts` arrays. This alignment will ensure that the off-chain data is handled correctly for each token being staked, preventing any inconsistencies and enhancing the overall reliability of the staking process.

TSI - Tokens Sufficiency Insurance

Criticality	Critical
Location	TeaStaking.sol#L100,112,185
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.

Specifically, the contract is not transferring the initial `totalAllocation` of tokens to the contract upon setting it during initialization. This lack of funding means that users may not be able to receive their rewards and staked amounts, especially if the contract's token balance is insufficient since the owner has the ability to withdraw the entire tea token balance. This poses a significant risk, as it can prevent users from claiming not only their rewards, but also their originally staked amount if the contract does not hold enough tokens to cover all liabilities.

```
    function initializeStaking(uint256 _totalAllocation, uint256
_rewardDistributionStartTime) external {
        _checkRole(DEFAULT_ADMIN_ROLE, msg.sender);
        if (!checkStakingStart()) {
            totalAllocation = _totalAllocation;
            ...
        }

    function emergencyWithdraw() external {
        _checkRole(DEFAULT_ADMIN_ROLE, msg.sender);
        IERC20(teaToken).safeTransfer(msg.sender,
IERC20(teaToken).balanceOf(address(this)));
    }
    function withdraw(uint256[] memory _ids) public nonReentrant {
        address user = _msgSender();

        for (uint256 i = 0; i < _ids.length; i++) {
            uint256 _id = _ids[i];

            ….

            address token = userStake.token;
            uint256 _availableTokens = userStake.availableTokens;
            uint256 _reward = userStake.rewardDebt;

            removeFromSystem(user, _id);

            if (token == teaToken) {
                IERC20(teaToken).safeTransfer(user, _availableTokens);
            } else {
                uint256 unlockedAmount =
ITeaVesting(teaVesting).getUserUnlockReward(token, user);
                ITeaVesting(teaVesting).claim(token, user);
                IERC20(teaToken).safeTransfer(user, unlockedAmount);
                ITeaVesting(teaVesting).transferOwnerOnChain(token,
address(this), user);
            }
            IERC20(teaToken).safeTransfer(user, _reward);

            emit Withdrawal(user, token, _availableTokens, _reward);

        }
    }
```

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.

The contract should ensure that the `totalAllocation` of tokens is transferred to the contract at the time of initialization to guarantee that sufficient funds are available for user withdrawals. Additionally, the owner's ability to withdraw funds should be restricted or modified to prevent the withdrawal of staked tokens that belong to users. Implementing a separate emergency withdrawal mechanism that does not affect user staked balances will help safeguard user funds and ensure the integrity of the staking and reward distribution process.

UOD - Unchecked Off-Chain Data

Criticality	Critical
Location	TeaStaking.sol#L120
Status	Unresolved

Description

The contract is utilizing the `stake` function, which accepts `_tokens` as a parameter and calls the `transferOwnerOffChain` function of the vesting contract using the `_offChain` data. However, the function does not validate the integrity of the `_offChain` data. Specifically, it does not ensure that the `_offChain.token` matches the corresponding `_token` being staked, nor does it verify that `_offChain.to` is set to this contract's address to properly transfer ownership of the vesting tokens. This lack of validation can result in incorrect token transfers and potential loss of funds.

```
function stake(address[] calldata _tokens, uint256[] calldata _amounts,
OffChainStruct calldata _offChain)
        external
    {
        ...
        address user = _msgSender();

        for (uint256 i = 0; i < _tokens.length; i++) {
            ...

            if (_token == teaToken) {
                IERC20(teaToken).safeTransferFrom(user, address(this),
_amount);
            } else {
                if (
                    ITeaVesting(teaVesting).getVestingUsers(user,
_token).tokensForVesting
                        - ITeaVesting(teaVesting).getVestingUsers(user,
_token).totalVestingClaimed < _amount
                ) {
                    revert NotEnoughLockedTokens();
                }
                ITeaVesting(teaVesting).transferOwnerOffChain(_offChain);
            }

            ...
            uint256 newId = ++counter;
            stakes[newId] = (
                Stake({
                    vip: _vip,
                    token: _token,
                    stakedTokens: _amount,
                    availableTokens: 0,
                    rewardDebt: _amount * rewardPerShare /
ACCUMULATED_PRECISION,
                    claimCooldown: 0,
                    lockedPeriod: _lockedPeriod
                })
            );

            totalStakedTokens += _amount;
            userIds[user].push(newId);

            emit Staked(user, _token, _amount);
        }
    }

    struct OffChainStruct {
        address token;
        address from;
```

```
        address to;
        uint256 deadline;
        uint8 v;
        bytes32 r;
        bytes32 s;
    }
```

Recommendation

It is recommended to implement checks to verify the correctness of the `OffChainStruct` parameters before proceeding with the off-chain transaction. Ensure that `_offChain.token` corresponds to the token being staked and that `_offChain.to` is set to this contract's address to securely acquire ownership of the vesting tokens. Adding these validations will help prevent unintended behavior and ensure the security of the staking process.

VSV - Vulnerable Signature Verification

Criticality	Critical
Location	TeaStaking.sol#L177,273 SignatureHandler.sol#L27
Status	Unresolved

Description

The contract's `unstake` function takes `UnstakeParam` parameters, including an `operator` address, and verifies the signature based on this `operator`. However, since the `operator` is not restricted to a specific address, users can set any address as the operator and sign the transaction themselves. This vulnerability allows users to bypass signature verification and set arbitrary values in the `_rewardsWithLoyalty` parameter, enabling them to withdraw 1.5 times their reward allocation. A malicious user could repeatedly exploit this vulnerability to drain the contract's funds.

```
    function unstake(UnstakeParam calldata _params) external {
        (bool success, string memory errorReason) =
_verifyUnstakeSignature(msg.sender, _params);
        require(success, errorReason);

        _unstake(_params.ids, _params.rewardsWithLoyalty);
    }
    function _unstake(uint256[] memory _ids, uint256[] calldata
_rewardsWithLoyalty) private {
        ...

            uint256 rewardWithThreshold = userStake.rewardDebt +
(userStake.rewardDebt / 2);

            if (_proof > rewardWithThreshold) {
                revert InvalidCalculationReward(_proof,
rewardWithThreshold);
            } else {
                userStake.rewardDebt = _proof;
            }

            if (userStake.token != teaToken) {
                _withdraw(_id);
            } else {
                userStake.claimCooldown = block.timestamp + 2 weeks;
            }

            emit Unstaked(user, _id, amount);
        }
    }
    ...
    }

     function _verifySignature(
        bytes memory encodedData,
        address from,
        address operator,
        uint256 nonce,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) internal returns (bool result, string memory errorReason) {
        ...
        address recoveredAddress = ECDSA.recover(digest, v, r, s);
        if (recoveredAddress != operator) {
            return (false, "INVALID_SIGNATURE");
        }
        return (true, "");
    }
```

```
    struct UnstakeParam {
        address user;
        address operator;
        uint256[] ids;
        uint256[] rewardsWithLoyalty;
        uint256 nonce;
        uint256 deadline;
        uint8 v;
        bytes32 r;
        bytes32 s;
    }
```

Recommendation

It is recommended to restrict the `operator` address to a predefined, trusted address, or use a more secure signature scheme that ties the `operator` to a specific authorized entity. Additionally, implement checks to ensure that only legitimate reward allocations can be set in the `_rewardsWithLoyalty` parameter. These measures will prevent unauthorized access and manipulation of the reward amounts, safeguarding the contract's funds from potential abuse.

ITV - Inconsistent Token Valuation

Criticality	Medium
Location	TeaStaking.sol#L169
Status	Unresolved

Description

The contract is handling the staking functionality under the assumption that all presale tokens, including the tea tokens, have the same price and decimal characteristics. This is reflected in the way the `totalStakedTokens` variable is incremented by the raw `amount` of each staked token, without accounting for the differing prices and decimal places of each token. As a result, the contract treats all staked tokens as having the same weight and value, which can lead to inaccurate reward calculations, unfair staking conditions, and incorrect token distribution.

```
    function stake(address[] calldata _tokens, uint256[] calldata
_amounts, OffChainStruct calldata _offChain)
        external
    {
    ...
        address user = _msgSender();

        for (uint256 i = 0; i < _tokens.length; i++) {
            address _token = _tokens[i];
            uint256 _amount = _amounts[i];

            ...

            if (_token == teaToken) {
                IERC20(teaToken).safeTransferFrom(user, address(this),
_amount);
            } else {
                if (
                    ITeaVesting(teaVesting).getVestingUsers(user,
_token).tokensForVesting
                        - ITeaVesting(teaVesting).getVestingUsers(user,
_token).totalVestingClaimed < _amount
                ) {
                    revert NotEnoughLockedTokens();
                }

ITeaVesting(teaVesting).transferOwnerOffChain(_offChain);
            }

            ...

            totalStakedTokens += _amount;
            userIds[user].push(newId);

            emit Staked(user, _token, _amount);
        }
    }
```

Recommendation

It is recommended to implement a mechanism that normalizes the staked token amounts based on their individual prices and decimal characteristics before updating the `totalStakedTokens` variable. This could involve converting all staked token amounts to a common value metric, such as the equivalent amount in tea tokens or another base unit. By ensuring that each token's unique properties are accounted for, the contract can accurately reflect the value of the staked tokens and maintain fair and consistent staking and reward calculations.

RIC - Redundant If Check

Criticality	Minor / Informative
Location	TeaStaking.sol#L194
Status	Unresolved

Description

The contract is using an `if` statement within the `withdraw` function to verify if an unstake operation has occurred by checking both the `claimCooldown` and `availableTokens` variables. However, since `availableTokens` only increments within the `unstake` function, the first part of the check that evaluates `claimCooldown == 0` is redundant. This check does not provide any additional validation beyond what is already accomplished by verifying `availableTokens`, leading to unnecessary complexity and potential confusion in the code.

```
    function withdraw(uint256[] memory _ids) public nonReentrant {
      ...
          if ((checkTeaAddrAndNotVip(_id) && userStake.claimCooldown == 0)
|| userStake.availableTokens == 0) {
              revert NeedToUnstakeFirst(_id);
          }
      ...
    }
```

Recommendation

It is recommended to consider the removal of the first part of the `if` check, specifically `checkTeaAddrAndNotVip(_id) && userStake.claimCooldown == 0`, as it does not add any meaningful validation. Simplifying this condition to only check `userStake.availableTokens == 0` will streamline the logic, making the function easier to understand and maintain while preserving its intended functionality.

DPHI - Decimal Precision Handling Inconsistency

Criticality	Minor / Informative
Location	TeaStaking.sol#L24
Status	Unresolved

Description

The contract is declaring the `VIP_AMOUNT` under the assumption that the token has 18 decimals, as indicated by the declaration `uint256 public constant VIP_AMOUNT = 1_000_000e18;`. This assumption can lead to inconsistencies if the deployed token uses a different number of decimals. Tokens on the Ethereum network can have varying decimal places, and hardcoding the decimal assumption without verifying it can cause incorrect calculations and potential loss of funds or unexpected behavior.

```
uint256 public constant VIP_AMOUNT = 1_000_000e18;
```

Recommendation

It is recommended to refactor the code to utilize the `.decimals()` function of the ERC-20 token standard, which dynamically retrieves the actual decimal places used by the token. This will ensure that the contract handles the token amounts correctly, irrespective of the token's decimal configuration.

IPT - Immutable Presale Tokens

Criticality	Minor / Informative
Location	TeaStaking.sol#L88,350
Status	Unresolved

Description

The contract is utilizing presale tokens whose addresses are set during the constructor and cannot be modified throughout the contract's lifecycle. This inflexibility poses a risk if the presale token addresses are not aligned with those used by the vesting contract, potentially leading to inconsistencies in token management and allocation. Ensuring that the presale tokens are consistent across all related contracts is crucial to avoid discrepancies in token distribution and access control.

```
    constructor(
        ...
        address[] memory _presaleTokens
    ) ERC2771Context(_trustedForwarder) {
        ...
        teaToken = _teaToken;
        for (uint256 i = 0; i < _presaleTokens.length; i++) {
            presaleTokens.push(_presaleTokens[i]);
        }
    }

    /// @dev Internal function to check if token address is valid
    /// @param _token The token of user's stake
    function checkTokenValidity(address _token) private view returns
(bool) {
        if (_token == teaToken) {
            return true;
        } else {
            for (uint256 i = 0; i < presaleTokens.length;) {
                if (presaleTokens[i] == _token) {
                    return true;
                } else {
                    ++i;
                }
            }
            return false;
        }
    }
```

Recommendation

It is recommended to set the presale tokens to match those used by the vesting contract. The contract should retrieve the token addresses directly from the vesting contract to ensure consistency and prevent any misalignment between the presale and vesting phases. This approach will enhance the robustness of the contract and ensure a seamless token allocation process.

IVT - Incorrect Vesting Transfer

Criticality	Minor / Informative
Location	TeaStaking.sol#L213
Status	Unresolved

Description

The contract, during the execution of the `withdraw` function, calls the `transferOwnerOnChain` function of the vesting contract to transfer the ownership of the vesting to the user. However, the usage of the `user` parameter is inaccurate and does not align with the actual implementation of the vesting contract. The vesting contract may be designed to transfer ownership to the zero address (`ZERO_ADDRESS`), irrespective of the provided `user` address. This discrepancy indicates that the current code does not reflect the intended functionality and may lead to unexpected behavior or errors in the ownership transfer process.

```
function withdraw(uint256[] memory _ids) public nonReentrant {
        address user = _msgSender();

        for (uint256 i = 0; i < _ids.length; i++) {
            ...
                ITeaVesting(teaVesting).transferOwnerOnChain(token,
address(this), user);
            }
            ...
            emit Withdrawal(user, token, _availableTokens, _reward);
        }
    }
```

Recommendation

It is recommended to set the `ZERO_ADDRESS` instead of the `user` in the `transferOwnerOnChain` function call to accurately reflect the actual behavior of the vesting contract. This change will ensure that the contract behaves as expected and that ownership transfer is executed correctly, preventing any potential inconsistencies or misuse of the function.

ILO - Inefficient Loop Operations

Criticality	Minor / Informative
Location	TeaStaking.sol#L326,370,393,352
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 is utilizing `for` loops in multiple functions, such as `checkUserStakeExists` , `checkTokenValidity` , `checkIdValidity` , and `removeIdFromList` , to iterate over arrays. This approach is gas-inefficient, especially as the array sizes grow. These loops could lead to increased gas costs and potential issues with transaction execution limits, particularly if the arrays become large due to high user activity or token interactions.

```
    function checkUserStakeExists(address _token, address _user) private
view returns (bool) {
        uint256[] memory allIds = userIds[_user];
        for (uint256 i = 0; i < allIds.length;) {
            if (stakes[allIds[i]].token == _token) {
                return true;
            } else {
                ++i;
            }
        }
        return false;
    }

    function checkTokenValidity(address _token) private view returns
(bool) {
        if (_token == teaToken) {
            return true;
        } else {
            for (uint256 i = 0; i < presaleTokens.length;) {
                if (presaleTokens[i] == _token) {
                    return true;
                } else {
                    ++i;
                }
            }
            return false;
        }
    }

    function checkIdValidity(address _user, uint256 _id) private view
returns (bool) {
        uint256[] memory allIds = userIds[_user];
        for (uint256 i = 0; i < allIds.length;) {
            if (allIds[i] == _id) {
                return true;
            } else {
                ++i;
            }
        }
        return false;
    }

    function removeIdFromList(address _user, uint256 _id) private {
        uint256[] storage allIds = userIds[_user];
        for (uint256 i = 0; i < allIds.length;) {
            if (allIds[i] == _id) {
                allIds[i] = allIds[allIds.length - 1];
                allIds.pop();
                break;
            } else {
```

```
                ++i;
            }
        }
    }
```

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 contract to use mappings instead of `for` loops for these operations. Mappings allow for constant-time lookups, making them significantly more gas-efficient than iterative loops. Implementing mappings for user stakes, token validity, and ID checks will optimize the contract's performance, reduce gas costs, and improve the overall scalability of the contract.

MSE - Misleading Staking Error

Criticality	Minor / Informative
Location	TeaStaking.sol#L123,388
Status	Unresolved

Description

The contract is using the `checkStakingStart()` function to determine if the staking period has begun, returning `false` when the `block.timestamp` is greater than `endDate`. This causes the `StakingNotStarted` error to be thrown, even when the staking period has already ended. This misleading error message can create confusion for users, as it suggests that staking has not started when, in fact, it has already started and ended.

```
   function stake(address[] calldata _tokens, uint256[] calldata
_amounts, OffChainStruct calldata _offChain)
        external
    {
        if (!checkStakingStart()) revert StakingNotStarted();
        ...
        }

    /// @dev Internal function to check if staking period has started
    function checkStakingStart() private view returns (bool) {
        return block.timestamp < endDate;
    }
```

Recommendation

It is recommended to modify the error handling to clearly distinguish between staking not having started and staking having ended. Implement a separate condition to check if the current time is within the staking period range (i.e., between `startDate` and `endDate`). Use appropriate error messages to reflect the actual status of the staking period, ensuring users receive accurate feedback on why their staking attempt is being rejected.

MVN - Misleading Variable Naming

Criticality	Minor / Informative
Location	TeaStaking.sol#L163,290,302,317
Status	Unresolved

Description

The contract uses the variable `rewardDebt` within the `stake` function correctly to represent the user's proportional share of accumulated rewards that the contract should deduct from the user, based on their staking amount and the time of the staking. However, in other parts of the contract, such as the `_unstake` function, the same `rewardDebt` variable is used to represent the user's pending rewards to claim, which as a result is misleading. This inconsistency in naming can cause confusion, as `rewardDebt` in the `stake` function is intended to represent a debt or liability of the contract, while in other parts it denotes the user's unclaimed rewards.

```
    function stake(address[] calldata _tokens, uint256[] calldata
_amounts, OffChainStruct calldata _offChain)
        external
    {
        ...
            uint256 newId = ++counter;
            stakes[newId] = (
                Stake({
                    vip: _vip,
                    token: _token,
                    stakedTokens: _amount,
                    availableTokens: 0,
                    rewardDebt: _amount * rewardPerShare /
ACCUMULATED_PRECISION,
                    claimCooldown: 0,
                    lockedPeriod: _lockedPeriod
                })
            );
            ...
            }

  function _unstake(uint256[] memory _ids, uint256[] calldata
_rewardsWithLoyalty) private {
        ...

        for (uint256 i = 0; i < _ids.length; i++) {
            ...
            userStake.rewardDebt = harvest(_id);

            ...

            uint256 rewardWithThreshold = userStake.rewardDebt +
(userStake.rewardDebt / 2);

            if (_proof > rewardWithThreshold) {
                revert InvalidCalculationReward(_proof,
rewardWithThreshold);
            } else {
                userStake.rewardDebt = _proof;
            }

            ...

            emit Unstaked(user, _id, amount);
        }
    }

    /// @dev Internal function to harvest user's rewards
    /// @param _id The ID of user's stake to be harvested
    function harvest(uint256 _id) private view returns (uint256 reward)
```

```
{
        Stake storage userStake = stakes[_id];
        uint256 accumulatedReward = userStake.stakedTokens *
rewardPerShare / ACCUMULATED_PRECISION;
        return accumulatedReward - userStake.rewardDebt;
    }
```

Recommendation

It is recommended to maintain the current use of `rewardDebt` within the `stake` function as it accurately reflects the users's debt. However, in other parts of the contract where `rewardDebt` is used to denote pending rewards, consider renaming it to `pendingReward` or `unclaimedReward` to better represent its purpose. This will ensure clarity and consistency in the code, reducing the risk of misunderstandings and potential errors during maintenance and auditing.

MC - Missing Check

Criticality	Minor / Informative
Location	TeaStaking.sol#L73
Status	Unresolved

Description

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

Specifically, the contract is missing checks to verify that the addresses is not set to the zero address.

```
    constructor(
        address _admin,
        address _operation,
        address _trustedForwarder,
        address _teaVesting,
        address _teaToken,
        address[] memory _presaleTokens
    ) ERC2771Context(_trustedForwarder) {
        _grantRole(DEFAULT_ADMIN_ROLE, _admin);
        _grantRole(OPERATION_ROLE, _operation);

        teaVesting = _teaVesting;

        teaToken = _teaToken;
        for (uint256 i = 0; i < _presaleTokens.length; i++) {
            presaleTokens.push(_presaleTokens[i]);
        }
    }
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

ODV - Operator Dependency Vulnerability

Criticality	Minor / Informative
Location	TeaStaking.sol#L177,185
Status	Unresolved

Description

The contract contains the `unstake` function that must be called before the `withdraw` function, allowing users to claim their tokens. However, the `unstake` function requires a signature verification from an operator. If the operator is unable to sign the transaction, due to issues like a backend failure or unavailability, the required signature cannot be obtained, and users will be unable to proceed with unstaking and withdrawing their tokens. This dependency will lock users out of their funds under those conditions.

```
   function unstake(UnstakeParam calldata _params) external {
        (bool success, string memory errorReason) =
_verifyUnstakeSignature(msg.sender, _params);
        require(success, errorReason);

        _unstake(_params.ids, _params.rewardsWithLoyalty);
    }

    function withdraw(uint256[] memory _ids) public nonReentrant {
        address user = _msgSender();

        for (uint256 i = 0; i < _ids.length; i++) {
            uint256 _id = _ids[i];

            if (!checkIdValidity(user, _id)) revert InvalidId(_id);

            Stake storage userStake = stakes[_id];
            if ((checkTeaAddrAndNotVip(_id) && userStake.claimCooldown
== 0) || userStake.availableTokens == 0) {
                revert NeedToUnstakeFirst(_id);
            ...
            }
```

Recommendation

It is recommended to include an emergency withdrawal function that allows users to claim their staked amounts without requiring a signature verification in cases where the operator is unavailable. This will ensure that users have a fallback mechanism to access their funds in emergency situations, improving the resilience and reliability of the contract.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	TeaStaking.sol#L136
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

```
function stake(address[] calldata _tokens, uint256[] calldata _amounts,
OffChainStruct calldata _offChain)
        external
    {
        ...
        address user = _msgSender();

        for (uint256 i = 0; i < _tokens.length; i++) {
            address _token = _tokens[i];
            uint256 _amount = _amounts[i];
            ...
            if (_token == teaToken) {
                IERC20(teaToken).safeTransferFrom(user, address(this),
_amount);
            } else {

            ...

            uint256 newId = ++counter;
            stakes[newId] = (
                Stake({
                    vip: _vip,
                    token: _token,
                    stakedTokens: _amount,
                    availableTokens: 0,
                    rewardDebt: _amount * rewardPerShare /
ACCUMULATED_PRECISION,
                    claimCooldown: 0,
                    lockedPeriod: _lockedPeriod
                })
            );

            totalStakedTokens += _amount;
            userIds[user].push(newId);

            emit Staked(user, _token, _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
```

SVMC - Signature Validation Missing ChainID

Criticality	Minor / Informative
Location	SignatureHandler.sol#L27
Status	Unresolved

Description

The contract's `_verifySignature` function is designed to validate off-chain signatures for operations involving token transfers between addresses. However, the function does not include the `chainId` as part of the parameters in the signature verification process. While the use of a nonce can prevent replay attacks within the same network by ensuring each signature is unique for a particular transaction, it does not safeguard against replay attacks across different networks. Without the inclusion of `chainId`, a legitimate signature on one blockchain could be maliciously reused on another chain, potentially resulting in unintended or unauthorized token transfers, thus exposing the contract to cross-network vulnerabilities.

```
function _verifySignature(
        bytes memory encodedData,
        address from,
        address operator,
        uint256 nonce,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) internal returns (bool result, string memory errorReason) {
        if (operator == address(0)) {
            return (false, "UNAUTHORIZED_OPERATION");
        }
        if (deadline < block.timestamp) {
            return (false, "SIGNATURE_EXPIRED");
        }
        if (nonce != operatorUserNonces[operator][from]++) {
            return (false, "MISMATCHING_NONCES");
        }
        bytes32 digest = _hashTypedDataV4(keccak256(encodedData));
        address recoveredAddress = ECDSA.recover(digest, v, r, s);
        if (recoveredAddress != operator) {
            return (false, "INVALID_SIGNATURE");
        }
        return (true, "");
    }
```

Recommendation

It is recommended to incorporate the `chainId` in the signature verification process by including it in the parameters hashed during the signature construction. By doing so, the signatures will be explicitly tied to a specific network, effectively preventing them from being reused across different chains.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	TeaStaking.sol#L112
Status	Unresolved

Description

The `DEFAULT_ADMIN_ROLE` role has the authority to claim all the balance of the tea tokens on the contract. The `DEFAULT_ADMIN_ROLE` role may take advantage of it by calling the `emergencyWithdraw` function.

```
    function emergencyWithdraw() external {
            _checkRole(DEFAULT_ADMIN_ROLE, msg.sender);
            IERC20(teaToken).safeTransfer(msg.sender,
IERC20(teaToken).balanceOf(address(this)));
    }
```

Recommendation

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

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

UVC - Unfavorable VIP Conditions

Criticality	Minor / Informative
Location	TeaStaking.sol#L283
Status	Unresolved

Description

The contract assigns VIP status to users who stake an amount greater than or equal to the `VIP_AMOUNT` . However, instead of providing additional benefits or rewards to these VIP users, the contract imposes an additional lock duration of one year on their staked tokens. As a result, users who stake large amounts and attain VIP status will be unable to unstake or withdraw their tokens until the lock period has passed, which could be a disincentive for high-value stakers.

```
    function stake(address[] calldata _tokens, uint256[] calldata
_amounts, OffChainStruct calldata _offChain)
        external
    {
        ...

            bool _vip = false;
            uint256 _lockedPeriod = 0;
            if (_amount >= VIP_AMOUNT) {
                _vip = true;
                _lockedPeriod = block.timestamp + ONE_YEAR;
            }
            ...
            }

    function _unstake(uint256[] memory _ids, uint256[] calldata
_rewardsWithLoyalty) private {
        ...
        address user = _msgSender();

        for (uint256 i = 0; i < _ids.length; i++) {
            ...

            Stake storage userStake = stakes[_id];
            if (userStake.vip && !(block.timestamp > endDate + 30 days))
{
                if (block.timestamp < userStake.lockedPeriod) revert
LockedPeriodNotPassed(_id);
            }
        ...
```

Recommendation

It is recommended to reconsider the design of the VIP status and its associated conditions. Instead of imposing a lengthy lock period, the contract should provide additional rewards or incentives for VIP users, such as higher reward rates or exclusive benefits. This change will make the VIP status more attractive and encourage users to stake larger amounts without the concern of being locked out from their funds for an extended period.

UEE - Unnecessary Event Emission

Criticality	Minor / Informative
Location	TeaStaking.sol#L240
Status	Unresolved

Description

The contract's `updateRewardPerShare` function emits an event and updates the `lastUpdatedTimestamp` and `lastRewardBlockNumber` variables, even when `stakingRun` is set to `false`. This behavior is unnecessary and could lead to misleading information in the emitted events and inefficient use of gas. Since the reward calculations should only be updated while staking is active, the function's current implementation might confuse stakeholders about the status of staking rewards.

```
function updateRewardPerShare() public {
        if (block.number > lastRewardBlockNumber) {
            if (totalStakedTokens > 0 && stakingRun) {
                uint256 timePassed;
                if (block.timestamp < endDate) {
                    timePassed = block.timestamp - lastUpdatedTimestamp;
                } else {
                    timePassed = endDate - lastUpdatedTimestamp;
                    stakingRun = false;
                }
                uint256 tokensAccum = timePassed * allocationPerSecond;
                rewardPerShare += tokensAccum * ACCUMULATED_PRECISION /
totalStakedTokens;
            }
            lastUpdatedTimestamp = block.timestamp;
            lastRewardBlockNumber = block.number;

            emit UpdatedShareReward(lastUpdatedTimestamp,
totalStakedTokens, rewardPerShare);
        }
    }
```

Recommendation

It is recommended to add a condition to prevent the function from emitting events and updating timestamps when `stakingRun` is `false`. This will ensure that the reward distribution logic and emitted events accurately reflect the contract's state, preventing unnecessary gas consumption and potential confusion over the staking status.

UTT - Unoptimized Token Transfers

Criticality	Minor / Informative
Location	TeaStaking.sol#L209
Status	Unresolved

Description

The contract performs two separate token transfers to the user within the `withdraw` function. One for the `unlockedAmount` inside the `else` block and another for the `_reward` outside of it. This results in increased gas costs due to the redundant token transfer operations. Moreover, it introduces unnecessary complexity to the function and could potentially lead to confusion when auditing or maintaining the code.

```
   function withdraw(uint256[] memory _ids) public nonReentrant {
        address user = _msgSender();

        for (uint256 i = 0; i < _ids.length; i++) {
            ...
            address token = userStake.token;
            uint256 _availableTokens = userStake.availableTokens;
            uint256 _reward = userStake.rewardDebt;

            removeFromSystem(user, _id);

            if (token == teaToken) {
                IERC20(teaToken).safeTransfer(user, _availableTokens);
            } else {
                uint256 unlockedAmount =
ITeaVesting(teaVesting).getUserUnlockReward(token, user);
                ITeaVesting(teaVesting).claim(token, user);
                IERC20(teaToken).safeTransfer(user, unlockedAmount);
                ITeaVesting(teaVesting).transferOwnerOnChain(token,
address(this), user);
            }
            IERC20(teaToken).safeTransfer(user, _reward);
            …
}
```

Recommendation

It is recommended to optimize the logic by incrementing the `_reward` variable with the `unlockedAmount` within the `else` block. This way, only a single transfer is performed after the conditionals, reducing the gas cost and simplifying the withdrawal process. This change will improve the efficiency and clarity of the function without altering its intended behavior.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	TeaStaking.sol#L97,120,177,185,224,229,317,326,345,352,370,385,393
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 _totalAllocation
uint256 _rewardDistributionStartTime
uint256[] calldata _amounts
address[] calldata _tokens
OffChainStruct calldata _offChain
UnstakeParam calldata _params
uint256[] memory _ids
address _user
uint256 _id
address _token
```

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.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	TeaStaking.sol#L101
Status	Unresolved

Description

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

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

```
allocationPerSecond = _totalAllocation / ONE_YEAR
```

Recommendation

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

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	TeaStaking.sol#L84,86
Status	Unresolved

Description

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

```
teaVesting = _teaVesting
teaToken = _teaToken
```

Recommendation

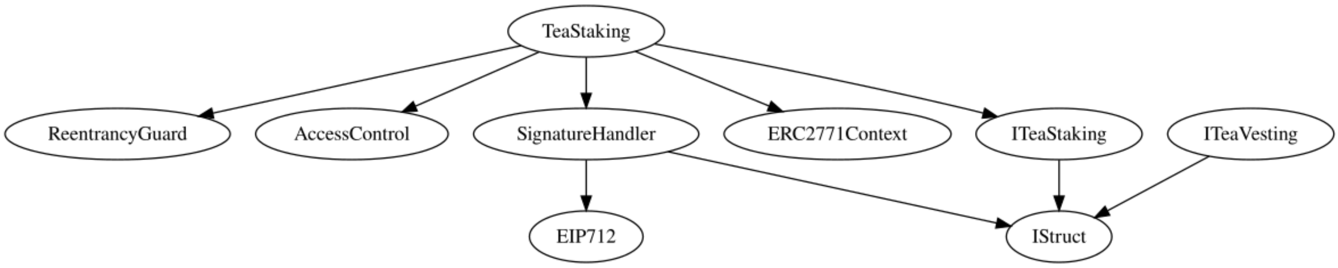
By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

Functions Analysis

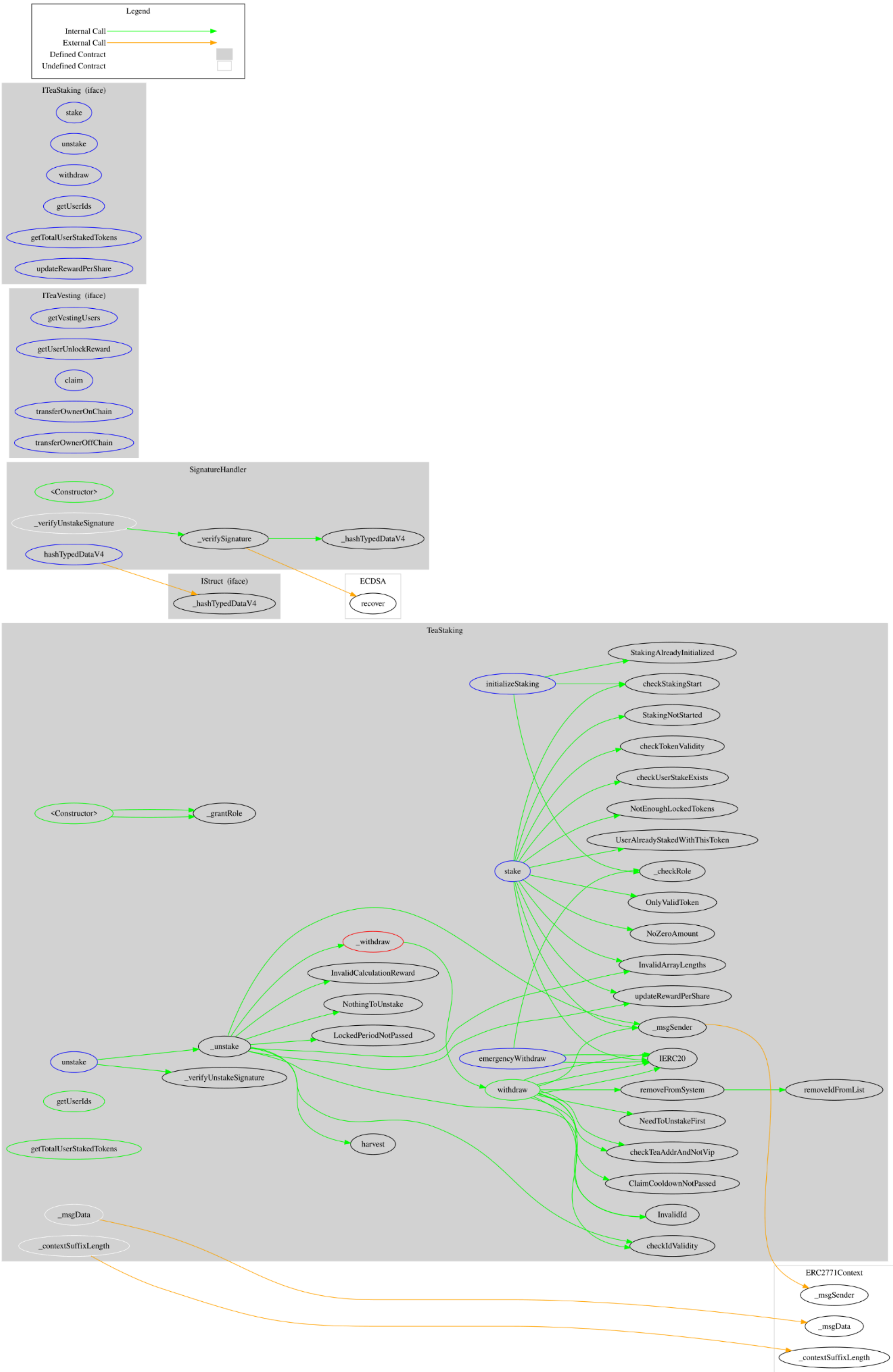
Contract	Type	Bases		
	Function Name	Visibility	Mutability	Modifiers
TeaStaking	Implementation	ITeaStaking, ReentrancyGuard, AccessControl, SignatureHandler, ERC2771Context		
		Public	✓	ERC2771Context
	initializeStaking	External	✓	-
	emergencyWithdraw	External	✓	-
	stake	External	✓	-
	unstake	External	✓	-
	withdraw	Public	✓	nonReentrant
	getUserIds	Public		-
	getTotalUserStakedTokens	Public		-
	updateRewardPerShare	Public	✓	-
	_withdraw	Private	✓	
	_unstake	Private	✓	
	harvest	Private		
	checkUserStakeExists	Private		
	checkStakingStart	Private		
	checkTeaAddrAndNotVip	Private		

	checkTokenValidity	Private		
	checkIdValidity	Private		
	removeFromSystem	Private	✓	
	removeIdFromList	Private	✓	
	_msgSender	Internal		
	_msgData	Internal		
	_contextSuffixLength	Internal		
SignatureHandler	Implementation	EIP712, IStruct		
		Public	✓	EIP712
	_verifyUnstakeSignature	Internal	✓	
	_verifySignature	Internal	✓	
	hashTypedDataV4	External		-
ITeaStaking	Interface	IStruct		
	stake	External	✓	-
	unstake	External	✓	-
	withdraw	External	✓	-
	getUserIds	External		-
	getTotalUserStakedTokens	External		-
	updateRewardPerShare	External	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
ITeaStaking (iface)
stake
unstake
withdraw
getUserIds
getTotalUserStakedTokens
updateRewardPerShare
ITeaVesting (iface)
getVestingUsers
getUserUnlockReward
claim
transferOwnerOnChain
transferOwnerOffChain
SignatureHandler
<Constructor>
_verifyUnstakeSignature
_verifySignature
_hashTypedDataV4
hashTypedDataV4
IStruct (iface)
_hashTypedDataV4
ECDSA
recover
TeaStaking
StakingAlreadyInitialized
initializeStaking
checkStakingStart
StakingNotStarted
checkTokenValidity
checkUserStakeExists
NotEnoughLockedTokens
UserAlreadyStakedWithThisToken
<Constructor>
_grantRole
stake
_checkRole
OnlyValidToken
NoZeroAmount
_withdraw
InvalidCalculationReward
InvalidArrayLengths
updateRewardPerShare
NothingToUnstake
LockedPeriodNotPassed
_msgSender
_unstake
unstake
emergencyWithdraw
IERC20
_verifyUnstakeSignature
withdraw
removeFromSystem
removeIdFromList
getUserIds
NeedToUnstakeFirst
harvest
getTotalUserStakedTokens
checkTeaAddrAndNotVip
ClaimCooldownNotPassed
_msgData
InvalidId
_contextSuffixLength
checkIdValidity
ERC2771Context
_msgSender
_msgData
_contextSuffixLength

Summary

The Tea-Fi Staking contract facilitates staking and reward distribution for Tea and presale tokens. This audit reviews security vulnerabilities, business logic issues, and potential optimizations to ensure safe and efficient operation.

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