Cyberscope

Audit Report

Tea-Fi Vesting

September 2024

SHA256 8dea654db0258aa31ea39ee0a24c9505ee7317ba9630f6358b413521ab690ee3

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/0x54a4b4563a07f89a3f6e58c585d72e8cbc329544

Audit Updates

Initial Audit	10 Sep 2024

Source Files

Filename	SHA256
contracts/TeaVesting.sol	8dea654db0258aa31ea39ee0a24c9505ee7317ba9630f6358b413521ab690ee3
contracts/interface/ITeaVesting.sol	3d3b7879fa5b72bdd1f3c880f0c64727a5bc1d5a886a53fedf1ad706a8b12840

Overview

The `TeaVesting` contract is designed to facilitate the vesting of "Tea" tokens by allowing users to exchange their presale tokens for vested tokens that are released over a period of time. The contract supports functionalities such as initiating and managing vesting schedules, claiming vested tokens, transferring vesting ownership, and administrative function for the contract owner. It provides a secure and efficient method for handling token vesting, ensuring that tokens are distributed fairly and transparently according to predefined schedules.

Vest Functionality

The vesting functionality allows users to lock their presale tokens in the contract to receive "Tea" tokens over a specific period. The tokens are vested linearly over time, meaning the tokens are gradually released according to a schedule rather than all at once. To initiate vesting, the user provides the presale tokens, which are then transferred to the contract. An initial portion of the tokens is unlocked immediately, and the remaining tokens can be claimed proportionally over the duration of the vesting period, by utilizing the `claim` function.

Claim Functionality

The contract provides a claim functionality, allowing users to retrieve the "Tea" tokens that have not been vested by the `vest` function up to the current point in time. The claim process also follows a linear distribution over the vesting period, meaning that users can claim tokens that have been unlocked incrementally over time. This means that if there are any tokens that were not allocated from the `vest` function, these tokens can be claimed through this function. The claiming process ensures that users receive their vested tokens in a fair and transparent manner.

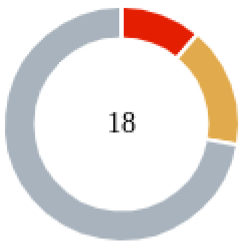
Transfer Ownership of Vesting Functionality

The contract allows users to transfer the ownership of their vesting schedule to another address. This functionality is useful for situations where the user wants to delegate the claiming of vested tokens to another party. Once the ownership is transferred, the new owner has the authority to perform actions, such as claiming the vested tokens on behalf of the original user. This flexibility ensures that users have control over who can access and manage their vesting schedules.

Owner Functionalities

The owner of the contract has the authority to set the correct initial configurations during the contract's deployment and initialization phase. Additionally, the owner can withdraw any token balance from the contract using the `forceTransfer` function. This function allows the owner to transfer tokens from the contract to any specified address, providing a mechanism to handle exceptional cases or correct errors in token distribution.

Findings Breakdown



●	Critical	2
●	Medium	3
●	Minor / Informative	13

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	2	0	0	0
●	Medium	3	0	0	0
●	Minor / Informative	13	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	UVM	Unauthorized Vesting Manipulation	Unresolved
●	VUV	Vesting Underflow Vulnerability	Unresolved
●	DPI	Decimals Precision Inconsistency	Unresolved
●	MCV	Missing Check Validations	Unresolved
●	PSU	Potential Subtraction Underflow	Unresolved
●	ALM	Array Length Mismatch	Unresolved
●	CO	Code Optimization	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	IAC	Insufficient Allowance Checks	Unresolved
●	MCM	Misleading Comment Messages	Unresolved
●	MMN	Misleading Modifier Naming	Unresolved
●	PTAI	Potential Transfer Amount Inconsistency	Unresolved
●	SVMC	Signature Validation Missing ChainID	Unresolved

●	TSI	Tokens Sufficiency Insurance	Unresolved
●	OCTD	Transfers Contract's Tokens	Unresolved
●	ZAV	Zero Amount Vesting	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved

UVM - Unauthorized Vesting Manipulation

Criticality	Critical
Location	contracts/TeaVesting.sol#L420
Status	Unresolved

Description

The contract contains a vulnerability within the `_vest` function, which allows the transfer of tokens from the `_userAddr` parameter to the contract's address and subsequently transfers `tea` tokens to the `msg.sender`. A malicious user could exploit this function by specifying any address that has granted an allowance to the contract, effectively transferring tokens from the `_userAddr` address without its consent and receiving the corresponding `tea` tokens. This creates a high-risk scenario where an attacker can initiate vesting on behalf of another user, thus receiving the vested tokens without being the rightful owner, leading to unauthorized asset transfers and financial loss for the users.

```
    function _vest(
        address _tokenAddr,
        address _userAddr,
        uint256 _amountToBurn,
        uint256 _initialUnlock,
        uint256 _vestedUnlock
    ) internal {
        IERC20(_tokenAddr).safeTransferFrom(_userAddr, address(this),
_amountToBurn);
        IERC20(tea).safeTransferFrom(treasury, _msgSender(),
_vestedUnlock + _initialUnlock);
        ...
    }
```

Recommendation

It is recommended to include additional checks to prevent unauthorized vesting actions. Specifically, the contract should verify that only the address holding the tokens can initiate a vesting transaction on its behalf. This can be achieved by ensuring that `_userAddr` matches `msg.sender` or by implementing other ownership verification mechanisms to guarantee that users cannot vest tokens belonging to others and claim the corresponding vested tokens.

VUV - Vesting Underflow Vulnerability

Criticality	Critical
Location	contracts/TeaVesting.sol#L208,398,134
Status	Unresolved

Description

The contract is vulnerable to an underflow issue due to improper handling of the `_tokensForVesting` parameter during the vesting process. The vulnerability occurs in the `vest` function, which incorrectly calls the `_updateUserVesting` function with a zero value for `_tokensForVesting` when the vesting period has ended. As a result, the `user.tokensForVesting` is not correctly updated, leading to a discrepancy in the user's vesting state.

Specifically when the `vest` function is called and the `dateEnd` has passed, the `_updateUserVesting` function is invoked with a zero value for the third parameter (`_tokensForVesting`) while the `user.totalVestingClaimed` is increased by the total vested amount. This results in incorrect calculations when the `claim` function is called after the `dateEnd` has passed and the user still has tokens to claim. The `reminder` value, derived from the difference between `tokensForVesting` and `totalVestingClaimed` , underflows because `tokensForVesting` was set to zero and not properly updated while the `totalVestingClaimed` was increased. Consequently, this underflow leads to an unintended negative value, causing the transaction to revert.

```
   function vest(
        address _tokenAddr,
        address _userAddr,
        uint256 _tokenAmount
    ) external isValidTokenAddr(_tokenAddr) isValidAddr(_userAddr)
nonReentrant {
        ...
        if (vestConfig.dateEnd <= currentTime) {  // When vesting
finished return all the tea tokens
            _updateUserVesting(
                _tokenAddr,
                _userAddr,
                0,
                _tokenAmount
            );
            ...
            return;
        ...
    function  _updateUserVesting(
        address tokenAddr,
        address userAddr,
        uint256 _tokensForVesting,
        uint256 vestingClaimed
    ) internal {
        UserVesting storage user = getVestingUsers[userAddr][tokenAddr];
        unchecked {
            user.tokensForVesting += _tokensForVesting;
            user.totalVestingClaimed += vestingClaimed;
        }
    }

    uint256 reminder = userVesting.tokensForVesting -
userVesting.totalVestingClaimed;
```

Recommendation

It is recommended to call the `_updateUserVesting` function with the `_tokenAmount` value instead of zero when the current time is greater than or equal to the end date within the `vest` function, since the contract deducts the tokens that have already been claimed in the claim process. The contract should handle this scenario properly by calculating the correct values and reverting the transaction appropriately if the claim function should revert due to all tokens being instantly claimed from the `vest` function, instead of allowing an underflow in its calculations.

DPI - Decimals Precision Inconsistency

Criticality	Medium
Location	contracts/TeaVesting.sol#L234,411
Status	Unresolved

Description

The decimals field of a contract's ERC20 token can be used to specify the number of decimal places that the token uses. For example, if decimals are set to `8`, it means that the smallest unit of the token is `0.00000001`, and if decimals are set to `18`, it means that the smallest unit of the token is `0.000000000000000001`.

However, there is an inconsistency in the way that the decimals field is handled in some ERC20 contracts. The ERC20 specification does not specify how the decimals field should be implemented, and as a result, some contracts use different precision numbers.

This inconsistency can cause problems when interacting with these contracts, as it is not always clear how the decimals field should be interpreted. For example, if a contract expects the decimals field to be 18 digits, but the contract being interacted with uses 8 digits, the result of the interaction may not be what was expected.

Specifically, the contract is vulnerable due to the assumption that all tokens handled, specifically the `_tokenAddr` tokens and the `tea` token, have the same decimal configurations. The amount of `tea` tokens to vest is calculated based on the `_tokenAmount` of the `_tokenAddr` token, but since there is no validation to ensure that the `_tokenAddr` tokens have the same decimal places as the `tea` token, discrepancies may occur. If the `tea` token has a different decimal configuration compared to the `_tokenAddr` token, it can result in an inaccurate vesting amount, potentially leading to users receiving either a greater or lesser amount of tokens than intended, depending on the decimal difference.

```
_vest(
    _tokenAddr,
    _userAddr,
    _tokenAmount,
    initialUnlock,
    vestingUnlock
);
...
    function _vest(
        address _tokenAddr,
        address _userAddr,
        uint256 _amountToBurn,
        uint256 _initialUnlock,
        uint256 _vestedUnlock
    ) internal {
        // as presale tokens are not burnable - just accumulate them in
the contract
        IERC20(_tokenAddr).safeTransferFrom(_userAddr, address(this),
_amountToBurn);
        IERC20(tea).safeTransferFrom(treasury, _msgSender(),
_vestedUnlock + _initialUnlock);
        emit Vest(_tokenAddr, _userAddr, _amountToBurn, _initialUnlock,
_vestedUnlock);
    }
```

Recommendation

To avoid these issues, it is important to carefully review the implementation of the decimals field of the underlying tokens. The team is advised to normalize each decimal to one single source of truth. A recommended way is to scale all the decimals to the greatest token's decimal. Hence, the contract will not lose precision in the calculations.

The following example depicts 3 tokens with different decimals precision.

ERC20	Decimals
Token 1	6
Token 2	9
Token 3	18

All the decimals could be normalized to 18 since it represents the ERC20 token with the greatest digits.

It is recommended to either validate that all `_tokenAddr` tokens being used have the same decimal configuration as the `tea` token or implement a mechanism to handle the decimal differences properly under a base decimal calculation. This approach will ensure accurate vesting amounts and prevent unintended discrepancies arising from varying token decimal places.

MCV - Missing Check Validations

Criticality	Medium
Location	contracts/TeaVesting.sol#L42,81
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:

1. **Missing `dateEnd` vs. `dateStart` Validation:** The contract does not verify that the `dateEnd` is greater than the `dateStart`. As a result, a scenario where `dateEnd` is before `dateStart` could occur, leading to an invalid vesting period and potential logical errors in the contract's behavior.
2. **Lack of Check on `percentUnlock`:** The contract does not ensure that the `percentUnlock` value is less than 1000. Given that this value is used in an unchecked block, this omission can lead to an underflow, potentially causing unintended results or erroneous calculations related to token unlock percentages.
3. **No Validation for Non-Zero `dateEnd`:** There is no check to confirm that `dateEnd` is greater than zero. Consequently, the `isValidTokenAddr` modifier, which relies on this condition, can be bypassed. This could allow invalid token addresses to pass through checks, leading to potential misuse or unauthorized access.

```
    constructor(
        ...,
        address[] memory _tokenAddrs,
        uint256[] memory _dataStarts,
        uint256[] memory _dataEnds,
        uint256[] memory _percentUnlocks
    ) Ownable(_initialOwner) EIP712(_name, "1")
ERC2771Context(_trustedForwarder) {
        ...
        uint256 len = _tokenAddrs.length;
        for(uint256 i=0; i<len; i++){
            if (_tokenAddrs[i] == ZERO_ADDRESS) {
                revert ZeroAddress();
            }

            getVestingTokens[_tokenAddrs[i]] = VestingOption({
                dateEnd: _dataEnds[i],
                dateStart: _dataStarts[i],
                dateDuration: _dataEnds[i] - _dataStarts[i],
                percentUnlock: _percentUnlocks[i]
            });
        }
    }

    modifier isValidTokenAddr(address _token) {
        require(
            getVestingTokens[_token].dateEnd != 0,
            "TeaVesting: INVALID_TOKEN_ADDRESS"
        );
        _;
    }
```

Recommendation

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

1. Ensure that `dateEnd` is always greater than `dateStart` to prevent invalid vesting periods.
2. Validate that `percentUnlock` does not exceed `1000` to avoid underflows during calculations.

3. Add a check to confirm that `dateEnd` is greater than zero to prevent bypassing the `isValidTokenAddr` modifier and to ensure that only valid token addresses are processed.

PSU - Potential Subtraction Underflow

Criticality	Medium
Location	contracts/TeaVesting.sol#L151,199,222,320
Status	Unresolved

Description

The contract subtracts two values, the second value may be greater than the first value if the contract owner misuses the configuration. As a result, the subtraction may underflow and cause the execution to revert.

Specifically, the contract performs several arithmetic operations within `unchecked` blocks without ensuring that the deductions will result in a positive value. Since these operations are within `unchecked` blocks, the Solidity compiler will not revert the transaction even if an arithmetic underflow or overflow occurs. Such unchecked arithmetic can lead to incorrect calculations, unexpected behavior, or potential exploits if negative values are treated as large positive values.

```
unchecked {
    elapsedTime = currentTime - vestConfig.dateStart; // 3 weeks
    // calculate unlock amount by time passed
    vestingUnlock = elapsedTime * userVesting.tokensForVesting
            / vestConfig.dateDuration
            - userVesting.totalVestingClaimed;
}
unchecked {
    initialUnlock = _tokenAmount * vestConfig.percentUnlock / 1000; //
50% = 500 // 1000
    tokenLeftAfterUnlock = _tokenAmount - initialUnlock; // 1000
}
unchecked {
    elapsedTime = currentTime - vestConfig.dateStart; // 2 weeks
    vestingUnlock = elapsedTime * tokenLeftAfterUnlock
        / vestConfig.dateDuration; // 500
}
if (currentTime >= vestConfig.dateEnd) {
    unchecked {
        return userVesting.tokensForVesting -
userVesting.totalVestingClaimed;
    }
}
uint256 elapsedTime;
unchecked {
    elapsedTime = currentTime - vestConfig.dateStart;
    return elapsedTime * userVesting.tokensForVesting
            / vestConfig.dateDuration
            - userVesting.totalVestingClaimed;  // vestingUnlock
}
```

Recommendation

The team is advised to properly handle the code to avoid underflow subtractions and ensure the reliability and safety of the contract. The contract should ensure that the first value is always greater than the second value. It should add a sanity check in the setters of the variable or not allow executing the corresponding section if the condition is violated.

It is recommended to implement proper checks before performing arithmetic operations within `unchecked` blocks, ensuring that all deductions will result in positive values. Specifically, the contract should verify that the values being subtracted or divided will not cause underflows or overflows. This can be achieved by either removing the `unchecked` blocks where critical operations are performed or by adding conditions that ensure the safety of these operations. This will help prevent vulnerabilities related to unchecked arithmetic and ensure the contract functions correctly in all scenarios.

ALM - Array Length Mismatch

Criticality	Minor / Informative
Location	contracts/TeaVesting.sol#L42
Status	Unresolved

Description

The contract is designed to handle the process of elements from arrays through functions that accept multiple arrays as input parameters. These functions are intended to iterate over the arrays, processing elements from each array in a coordinated manner. However, there are no explicit checks to verify that the lengths of these input arrays are equal. This lack of validation could lead to scenarios where the arrays have differing lengths, potentially causing out-of-bounds access if the function attempts to process beyond the end of the shorter array. Such situations could result in unexpected behavior or errors during the contract's execution, compromising its reliability and security.

```
    constructor(
        ...
        address[] memory _tokenAddrs,
        uint256[] memory _dataStarts,
        uint256[] memory _dataEnds,
        uint256[] memory _percentUnlocks
    ) Ownable(_initialOwner) EIP712(_name, "1")
ERC2771Context(_trustedForwarder) {
        ...

        uint256 len = _tokenAddrs.length;
        for(uint256 i=0; i<len; i++){
            if (_tokenAddrs[i] == ZERO_ADDRESS) {
                revert ZeroAddress();
            }

            getVestingTokens[_tokenAddrs[i]] = VestingOption({
                dateEnd: _dataEnds[i],
                dateStart: _dataStarts[i],
                dateDuration: _dataEnds[i] - _dataStarts[i],
                percentUnlock: _percentUnlocks[i]
            });
        }
    }
```

Recommendation

To mitigate this, it is recommended to incorporate a validation check at the beginning of the function that accepts multiple arrays to ensure that the lengths of these arrays are identical. This can be achieved by implementing a conditional statement that compares the lengths of the arrays, and reverts the transaction if the lengths do not match. Such a validation step will prevent out-of-bounds errors and ensure that elements from each array are processed in a paired and coordinated manner, thus preserving the integrity and intended functionality of the contract.

CO - Code Optimization

Criticality	Minor / Informative
Location	contracts/TeaVesting.sol#L107,178
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 unnecessarily complex due to redundant logic splitting within the `claim` and `vest` functions. These functions handle the vesting and claiming processes differently depending on whether the current time is before or after the vesting end date. As a result, the contract contains two separate cases for each function, leading to duplicated logic that could have been consolidated. This approach increases the contract's complexity, making it harder to read and maintain, while also increasing the potential for errors or inconsistencies in the two code paths.

```
   function  claim(address _tokenAddr, address _userAddr)
        external
        isValidTokenAddr(_tokenAddr)
        isValidAddr(_userAddr)
        nonReentrant
    {
        ...
        if (vestConfig.dateEnd <= currentTime) {
            uint256 reminder = userVesting.tokensForVesting -
userVesting.totalVestingClaimed;
            _updateUserVesting(
                _tokenAddr,
                _userAddr,
                0,
                reminder
            );
            _claim(
                _tokenAddr,
                _userAddr,
                reminder
            );
            return;
        }

        uint256 elapsedTime;
        uint256 vestingUnlock;
        unchecked {
            elapsedTime = currentTime - vestConfig.dateStart; // 3 weeks
            // calculate unlock amount by time passed
            vestingUnlock = elapsedTime * userVesting.tokensForVesting
                    / vestConfig.dateDuration
                    - userVesting.totalVestingClaimed;
        }

        _updateUserVesting(
            _tokenAddr,
            _userAddr,
            0,
            vestingUnlock
        );
        _claim(
            _tokenAddr,
            _userAddr,
            vestingUnlock
        );
    }

    function vest(
        address _tokenAddr,
        address _userAddr,
        uint256 _tokenAmount
    ) external isValidTokenAddr(_tokenAddr) isValidAddr(_userAddr)
nonReentrant {
        ...
        if (vestConfig.dateEnd <= currentTime) {  // When vesting
finished return all the tea tokens
            _updateUserVesting(
                _tokenAddr,
                _userAddr,
                0,
                _tokenAmount
            );
            _vest(
                _tokenAddr,
                _userAddr,
                _tokenAmount,
                initialUnlock, // initial unlock
                tokenLeftAfterUnlock // amount - initial unlock
            );
            return;
        }
        uint256 elapsedTime;
        uint256 vestingUnlock;
        unchecked {
            elapsedTime = currentTime - vestConfig.dateStart; // 2 weeks
            vestingUnlock = elapsedTime * tokenLeftAfterUnlock
                / vestConfig.dateDuration; // 500
        }

        _updateUserVesting(
            _tokenAddr,
            _userAddr,
            tokenLeftAfterUnlock,
            vestingUnlock
        );
        _vest(
            _tokenAddr,
            _userAddr,
            _tokenAmount,
            initialUnlock,
            vestingUnlock
        );
    }
```

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 simplify the contract by consolidating the logic in the `claim` and `vest` functions. Instead of splitting the logic into two cases based on the current time, the contract should first calculate the correct values (such as the amount to be vested or claimed) based on the time passed. Once these values are determined, the functions should apply the appropriate amounts in a unified way. This will reduce redundancy, improve code clarity, and minimize the risk of inconsistencies or errors in the contract's logic.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	contracts/TeaVesting.sol#L42
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.

Specifically, the owner has significant authority to configure and initialize critical parameters of the contract. During the contract's construction, the owner has control over setting key elements, such as the addresses for `tea` , `treasury` , and `trustedForwarder` , as well as defining vesting schedules (`_dataStarts` , `_dataEnds` , `_percentUnlocks`) for different tokens. This centralized control allows the owner to determine the configuration and behavior of the contract, including vesting terms and key addresses, without requiring any external validation or checks. As a result, there is a risk that if the owner's wallet is compromised then could set parameters that are unfavorable to other participants or use their authority to manipulate the contract in ways that may not align with the interests of all stakeholders.

```
 constructor(
        string memory _name,
        address _initialOwner,
        address _tea,
        address _treasury,
        address _trustedForwarder,
        address[] memory _tokenAddrs,
        uint256[] memory _dataStarts,
        uint256[] memory _dataEnds,
        uint256[] memory _percentUnlocks
    ) Ownable(_initialOwner) EIP712(_name, "1")
ERC2771Context(_trustedForwarder) {
        if (_tea == ZERO_ADDRESS || _treasury == ZERO_ADDRESS) {
            revert ZeroAddress();
        }

        tea = _tea;
        treasury = _treasury;

        uint256 len = _tokenAddrs.length;
        for(uint256 i=0; i<len; i++){
            if (_tokenAddrs[i] == ZERO_ADDRESS) {
                revert ZeroAddress();
            }

            getVestingTokens[_tokenAddrs[i]] = VestingOption({
                dateEnd: _dataEnds[i],
                dateStart: _dataStarts[i],
                dateDuration: _dataEnds[i] - _dataStarts[i],
                percentUnlock: _percentUnlocks[i]
            });
        }
    }
```

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.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	contracts/TeaVesting.sol#L57,58
Status	Unresolved

Description

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

```
tea
treasury
```

Recommendation

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

IAC - Insufficient Allowance Checks

Criticality	Minor / Informative
Location	contracts/TeaVesting.sol#L420,430
Status	Unresolved

Description

The contract lacks checks to ensure that there is a sufficient allowance from the `treasury` before attempting to transfer tokens using `safeTransferFrom` in both the `_vest` and `_claim` functions. In these functions, the contract transfers "tea" tokens from the `treasury` to either the `msg.sender` or a user address without verifying if the `treasury` has authorized enough allowance to cover the required amount. If the allowance is inadequate, these transfers will fail, causing the transaction to revert and potentially disrupting the vesting and claiming processes, leading to unexpected contract behavior and user dissatisfaction.

```
   function _vest(
        address _tokenAddr,
        address _userAddr,
        uint256 _amountToBurn,
        uint256 _initialUnlock,
        uint256 _vestedUnlock
    ) internal {
        ...
        IERC20(tea).safeTransferFrom(treasury, _msgSender(),
_vestedUnlock + _initialUnlock);
        ...
    }

    function _claim(
        address _tokenAddr,
        address _userAddr,
        uint256 _amountToUnlock
    ) internal {
        ...
        IERC20(tea).safeTransferFrom(treasury, _userAddr,
_amountToUnlock);
        emit Claim(_tokenAddr, _userAddr, _amountToUnlock);
    }
```

Recommendation

It is recommended to implement checks in both the `_vest` and `_claim` functions to confirm that the allowance granted by the `treasury` is sufficient before attempting any token transfers. This can be achieved by using the `allowance` method of the ERC-20 token contract to verify that the current allowance is at least equal to the required transfer amount. Adding these checks will prevent transaction failures due to insufficient allowances, ensuring the contract operates smoothly and providing a reliable and predictable experience for all users.

MCM - Misleading Comment Messages

Criticality	Minor / Informative
Location	contracts/TeaVesting.sol#L152,184,200,223
Status	Unresolved

Description

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

```
elapsedTime = currentTime - vestConfig.dateStart; // 3 weeks
...
// UserVesting memory userVesting =
getVestingUsers[_userAddr][_tokenAddr];
...
initialUnlock = _tokenAmount * vestConfig.percentUnlock / 1000; // 50% =
500 // 1000
tokenLeftAfterUnlock = _tokenAmount - initialUnlock; // 1000
...
elapsedTime = currentTime - vestConfig.dateStart; // 2 weeks
vestingUnlock = elapsedTime * tokenLeftAfterUnlock
                / vestConfig.dateDuration; // 500
```

Recommendation

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

MMN - Misleading Modifier Naming

Criticality	Minor / Informative
Location	contracts/TeaVesting.sol#L91
Status	Unresolved

Description

The contract is using a modifier named `isValidAddr`, which suggests that it validates the provided address in a comprehensive manner. However, the modifier only checks whether the address is not equal to the zero address. This is misleading, as it implies a more thorough validation of the address format or existence, which is not the case. This could result in confusion for developers and users of the contract, leading to potential misuse or incorrect assumptions about the security and validity of the address being used.

```
    modifier isValidAddr(address _addr) {
            if (_addr == ZERO_ADDRESS) {
                revert ZeroAddress();
            }
            _;
        }
```

Recommendation

It is recommended to rename the `isValidAddr` modifier to something more descriptive, such as `isNonZeroAddress`, to accurately reflect its function and avoid misleading implications about the extent of the address validation. This will improve the clarity of the contract's code and help ensure that its behavior aligns with its intended purpose.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	contracts/TeaVesting.sol#L419
Status	Unresolved

Description

The `safeTransferFrom` function is used to transfer a specified amount of tokens to the 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

```
IERC20(_tokenAddr).safeTransferFrom(_userAddr, address(this),
_amountToBurn);
```

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	contracts/TeaVesting.sol#L446
Status	Unresolved

Description

The contract's `_verify` 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 _verify(OffChainStruct calldata _offChainStruct) internal
returns(bool){
        bytes32 structHash = keccak256(
            abi.encode(
                TRANSFER_OWNER_TYPEHASH,
                _offChainStruct.token,
                _offChainStruct.from,
                _offChainStruct.to,
                _useNonce(_offChainStruct.from),
                _offChainStruct.deadline
            )
        );
        bytes32 hash = _hashTypedDataV4(structHash);
        address recoveredAddress = ECDSA.recover(
            hash,
            _offChainStruct.v,
            _offChainStruct.r,
            _offChainStruct.s
        );
        return recoveredAddress == _offChainStruct.from;
    }
    }
```

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.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract relies on an external administrator (the treasury) to provide the tokens for vesting and claiming. While this external administration allows for flexibility, it also introduces a dependency on the administrator's actions, which can lead to centralization risks and potential issues if the administrator fails to supply the required tokens. The reliance on external transfers, such as `IERC20(tea).safeTransferFrom(treasury, _msgSender(), _vestedUnlock + _initialUnlock)` and `IERC20(tea).safeTransferFrom(treasury, _userAddr, _amountToUnlock)` , could result in failures or delays if the treasury does not hold sufficient tokens.

```
IERC20(tea).safeTransferFrom(treasury, _msgSender(), _vestedUnlock +
_initialUnlock);
...
IERC20(tea).safeTransferFrom(treasury, _userAddr, _amountToUnlock);
```

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, enhancing reliability, security, and participant trust. Additionally, the contract should include a mechanism to verify that the treasury address holds an adequate balance of `tea` tokens before initiating transfers. This would further reduce the risk of transfer failures and improve the overall security and efficiency of the process.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	contracts/TeaVesting.sol#L304
Status	Unresolved

Description

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

```
    function forceTransfer(address _tokenAddr, address _to, uint256
_amount)
        external
        onlyOwner
    {
        IERC20(_tokenAddr).safeTransfer(_to, _amount);
    }
```

Recommendation

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

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.

ZAV - Zero Amount Vesting

Criticality	Minor / Informative
Location	contracts/TeaVesting.sol#L178,411
Status	Unresolved

Description

The contract does not include a check to prevent the execution of the `vest` function when the `_tokenAmount` parameter is zero. As a result, users can call the `vest` function with a zero `_tokenAmount`, leading to the emission of the `Vest` event even when no actual tokens are being vested. This can cause misleading information to be recorded in the event logs, potentially leading to inaccurate data or analytics if event emissions are used for tracking purposes. Moreover, it can open the door to possible misuse or manipulation by malicious actors looking to create false vesting records.

```
   function vest(
        address _tokenAddr,
        address _userAddr,
        uint256 _tokenAmount
    ) external isValidTokenAddr(_tokenAddr) isValidAddr(_userAddr)
nonReentrant {
        ...

        // calculate initial unlock amount by percentUnlock
        unchecked {
            initialUnlock = _tokenAmount * vestConfig.percentUnlock /
1000; // 50% = 500 // 1000
            tokenLeftAfterUnlock = _tokenAmount - initialUnlock; // 1000
        }

        if (vestConfig.dateEnd <= currentTime) {  // When vesting
finished return all the tea tokens
            _updateUserVesting(
                _tokenAddr,
                _userAddr,
                0,
                _tokenAmount
            );
            _vest(
                _tokenAddr,
                _userAddr,
                _tokenAmount,
                initialUnlock, // initial unlock
                tokenLeftAfterUnlock // amount - initial unlock
            );
            return;
        }
        ...
        _vest(
            _tokenAddr,
            _userAddr,
            _tokenAmount,
            initialUnlock,
            vestingUnlock
        );
    }

    function _vest(
        address _tokenAddr,
        address _userAddr,
        uint256 _amountToBurn,
        uint256 _initialUnlock,
        uint256 _vestedUnlock
    ) internal {
        ...
        emit Vest(_tokenAddr, _userAddr, _amountToBurn, _initialUnlock, 
_vestedUnlock);
    }
```

Recommendation

It is recommended to implement a validation check within the `vest` function to ensure that the `_tokenAmount` is greater than zero before proceeding with any operations. This will prevent the function from executing when no actual tokens are being vested and avoid unnecessary or misleading event emissions. Adding this check will enhance the integrity and reliability of the contract's event data and prevent potential misuse.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/TeaVesting.sol#L107,179,180,181,250,251,252,273,304,314,315
Status	Unresolved

Description

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

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

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

```
address _userAddr
address _tokenAddr
uint256 _tokenAmount
address _from
address _owner
OffChainStruct calldata _offChainStruct
uint256 _amount
address _to
```

Recommendation

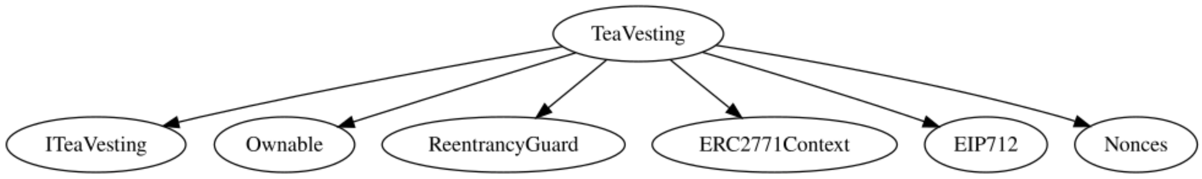
By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

Functions Analysis

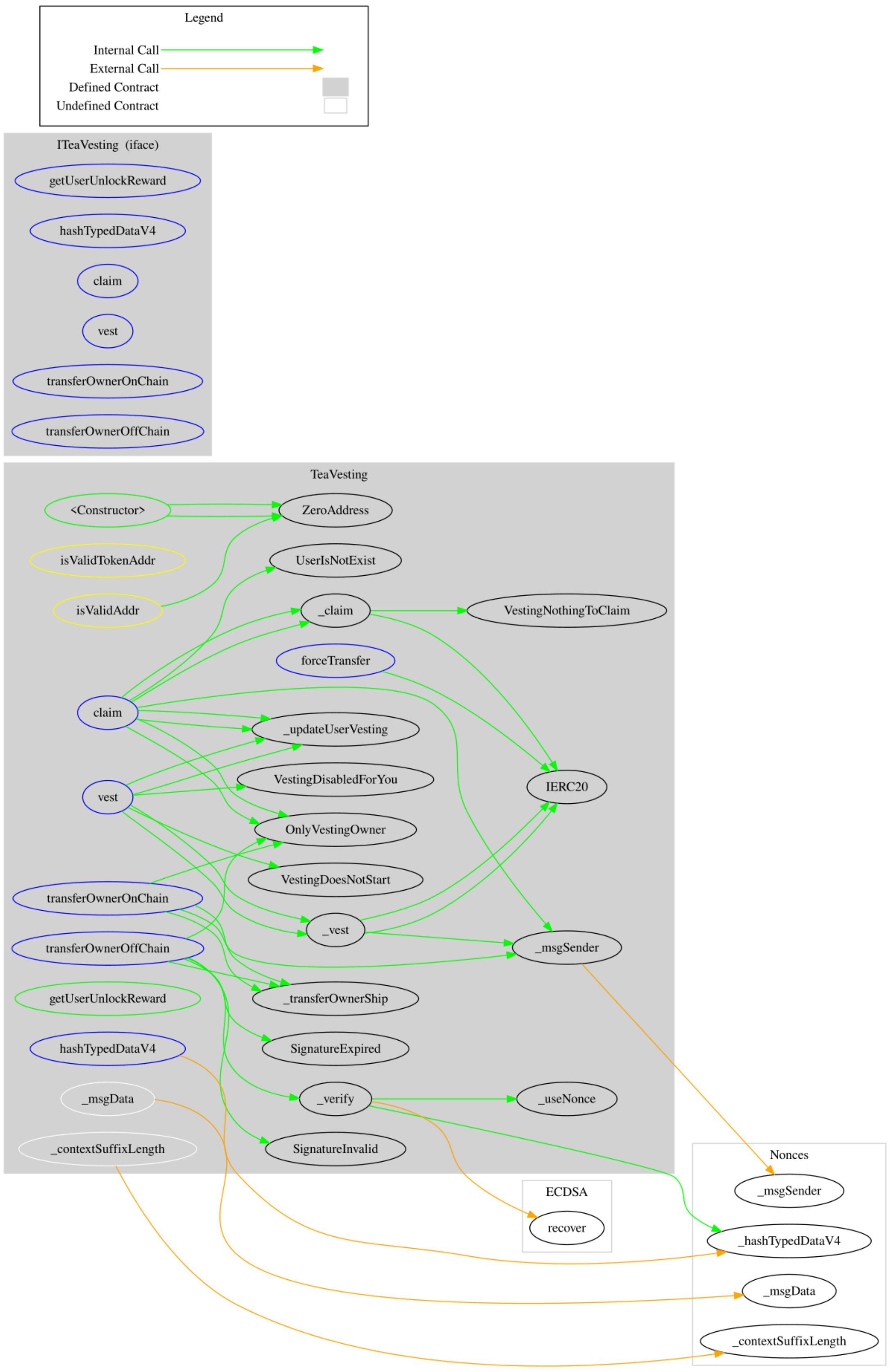
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
TeaVesting	Implementation	ITeaVesting, Ownable, ReentrancyGuard, ERC2771Context, EIP712, Nonces		
		Public	✓	Ownable EIP712 ERC2771Context
	claim	External	✓	isValidTokenAddr isValidAddr nonReentrant
	vest	External	✓	isValidTokenAddr isValidAddr nonReentrant
	transferOwnerOnChain	External	✓	isValidTokenAddr
	transferOwnerOffChain	External	✓	isValidTokenAddr
	forceTransfer	External	✓	onlyOwner
	getUserUnlockReward	Public		-
	hashTypedDataV4	External		-
	_msgSender	Internal		
	_msgData	Internal		
	_contextSuffixLength	Internal		
	_transferOwnerShip	Internal	✓	
	_updateUserVesting	Internal	✓	

	_vest	Internal	✓	
	_claim	Internal	✓	
	_verify	Internal	✓	
ITeaVesting	Interface			
	getUserUnlockReward	External		-
	hashTypedDataV4	External		-
	claim	External	✓	-
	vest	External	✓	-
	transferOwnerOnChain	External	✓	-
	transferOwnerOffChain	External	✓	-

Inheritance Graph



Flow Graph



Legend
Internal Call
External Call
Defined Contract
Undefined Contract
ITeaVesting (iface)
getUserUnlockReward
hashTypedDataV4
claim
vest
transferOwnerOnChain
transferOwnerOffChain
TeaVesting
<Constructor>
ZeroAddress
isValidTokenAddr
UserIsNotExist
isValidAddr
_claim
VestingNothingToClaim
forceTransfer
claim
_updateUserVesting
IERC20
vest
VestingDisabledForYou
OnlyVestingOwner
VestingDoesNotStart
transferOwnerOnChain
_vest
transferOwnerOffChain
_msgSender
getUserUnlockReward
_transferOwnerShip
hashTypedDataV4
SignatureExpired
_msgData
_verify
_useNonce
_contextSuffixLength
SignatureInvalid
ECDSA
recover
Nonces
_msgSender
_hashTypedDataV4
_msgData
_contextSuffixLength

Summary

The TeaVesting contract implements a vesting mechanism for "Tea" tokens, allowing users to exchange presale tokens for vested tokens released over time with functionalities for claiming, transferring vesting ownership, and administrative controls. This audit evaluates the contract's security, logic integrity, and areas for potential optimization.

Disclaimer

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