Cyberscope

Audit Report

Tea-Fi Vesting

September 2024

SHA256 cfc72db2db387a3fa4fc9a48041595780c6b60bd87ed2d78f684ab28d988da75

Audited by © cyberscope

Table of Contents

Table of Contents 1
Risk Classification 3
Review 4
Audit Updates 4
Source Files 4
Overview 5
Vest Functionality 5
Claim Functionality 5
Transfer Ownership of Vesting Functionality 5
Owner Functionalities 6
Findings Breakdown 7
Diagnostics 8
CO - Code Optimization 9
Description 9
Recommendation 11
CCR - Contract Centralization Risk 13
Description 13
Recommendation 15
Team Update 15
IAC - Insufficient Allowance Checks 16
Description 16
Recommendation 17
Team Update 17
PTAI - Potential Transfer Amount Inconsistency 18
Description 18
Recommendation 18
Team Update 19
TSI - Tokens Sufficiency Insurance 20
Description 20
Recommendation 20
Team Update 20
OCTD - Transfers Contract's Tokens 22
Description 22
Recommendation 22
Team Update 23
Functions Analysis 24
Inheritance Graph 26
Flow Graph 27
Summary 28

Disclaimer **29**
About Cyberscope **30**

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/0x0d741dc9d7376b0907fc5f4e443fc5516ede5616

Audit Updates

Initial Audit	10 Sep 2024 https://github.com/cyberscope-io/audits/blob/main/1-tea/v1/audit.pdf
Corrected Phase 2	27 Sep 2024 https://github.com/cyberscope-io/audits/blob/main/1-tea/v2/audit.pdf
Corrected Phase 3	01 Oct 2024

Source Files

Filename	SHA256
contracts/TeaVesting.sol	cfc72db2db387a3fa4fc9a48041595780c6b60bd87ed2d78f684ab28d988da75
contracts/interface/ITeaVesting.sol	af409247ee17a8530f0eacb69d05fecff946ce01a473994d0bb20ee8a90fa1ec

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

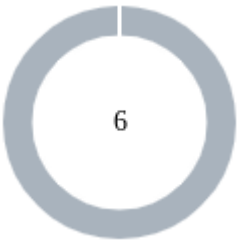
The contract allows users to transfer the ownership of their vesting schedule to another address. This functionality is useful for situations where the user wants to delegate the claiming of vested tokens to another party. Once the ownership is transferred, the new owner has the authority to perform 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	0
Medium	0
Minor / Informative	6

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	0	6	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	CO	Code Optimization	Acknowledged
●	CCR	Contract Centralization Risk	Acknowledged
●	IAC	Insufficient Allowance Checks	Acknowledged
●	PTAI	Potential Transfer Amount Inconsistency	Acknowledged
●	TSI	Tokens Sufficiency Insurance	Acknowledged
●	OCTD	Transfers Contract's Tokens	Acknowledged

CO - Code Optimization

Criticality	Minor / Informative
Location	contracts/TeaVesting.sol#L125,190
Status	Acknowledged

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)
        isNonZeroAddress(userAddr)
        nonReentrant
    {
        ...
        if (vestConfig.dateEnd <= currentTime) {
            uint256 reminder = userVesting.tokensForVesting -
userVesting.totalVestingClaimed;
            _updateUserVesting(
                tokenAddr,
                userAddr,
                0,
                reminder
            );
            _claim(
                tokenAddr,
                userAddr,
                reminder
            );
            return;
        }

        uint256 elapsedTime = currentTime - vestConfig.dateStart;
        uint256 vestingUnlock = elapsedTime *
userVesting.tokensForVesting
                / vestConfig.dateDuration
                - userVesting.totalVestingClaimed;

        _updateUserVesting(
            tokenAddr,
            userAddr,
            0,
            vestingUnlock
        );
        _claim(
            tokenAddr,
            userAddr,
            vestingUnlock
        );
    }

    function vest(
        address tokenAddr,
        uint256 tokenAmount
    ) external isValidTokenAddr(tokenAddr) nonReentrant {
        ...
        if (vestConfig.dateEnd <= currentTime) {
            _updateUserVesting(
```

```
                tokenAddr,
                userAddr,
                tokenAmount,
                tokenAmount
            );
            _vest(
                tokenAddr,
                userAddr,
                tokenAmount,
                initialUnlock,
                tokenLeftAfterUnlock
            );
            return;
        }

        uint256 elapsedTime = currentTime - vestConfig.dateStart;
        uint256 vestingUnlock = elapsedTime * tokenLeftAfterUnlock
            / vestConfig.dateDuration;

        _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#L43
Status	Acknowledged

Description

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

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 teaDecimal = IERC20Metadata(_tea).decimals();
        uint256 len = tokenAddrs.length;

        if (dataStarts.length != len ||
            dataEnds.length != len ||
            percentUnlocks.length != len
        ) {
            revert WrongTokenConfig();
        }
        for(uint256 i=0; i<len; i++){
            if (tokenAddrs[i] == ZERO_ADDRESS) {
                revert ZeroAddress();
            }
            if(IERC20Metadata(tokenAddrs[i]).decimals() != teaDecimal) {
                revert WrongTokenConfig();
            }
            if(percentUnlocks[i] > 1000) {
                revert WrongTokenConfig();
            }
            if(dataStarts[i] > dataEnds[i]) {
                revert WrongTokenConfig();
            }

            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.

Team Update

The team has acknowledged that this is not a security issue and states:

The owner of the contract will be a multi-signature contract with a multi-signature where there will be 3 people and to initialize the transaction, at least 2 signatures are needed. Which minimizes the centralization of the contract. Also, the owner can only call the forceTransfer function, which does not affect the user flow.

IAC - Insufficient Allowance Checks

Criticality	Minor / Informative
Location	contracts/TeaVesting.sol#L412,430
Status	Acknowledged

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, userAddr, vestedUnlock +
initialUnlock);
        emit Vest(tokenAddr, userAddr, amountToBurn, initialUnlock,
vestedUnlock);
    }

    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.

Team Update

The team has acknowledged that this is not a security issue and states:

The allowance check is done in the ERC20 smart contract. Due to this reason, the second check on the TeaVesting contract will result in more gas consumption.

PTAI - Potential Transfer Amount Inconsistency

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

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`

Team Update

The team has acknowledged that this is not a security issue and states:

The TeaVesting contract was designed to be used with our tokens - TEA and presale tokens. They do not have transfer fees, otherwise the team would have included this logic in the vesting process.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	TeaVesting.sol#L420,438
Status	Acknowledged

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, userAddr, 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, userAddr, 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.

Team Update

The team has acknowledged that this is not a security issue and states:

The treasury address is a multisig contract where the administrators are 3 people and to implement a transaction, at least 2 signatures are needed,which minimizes centralization. All vesting preparation requirements will be handled carefully by the administrators.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	contracts/TeaVesting.sol#L310
Status	Acknowledged

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.

Team Update

The team has acknowledged that this is not a security issue and states:

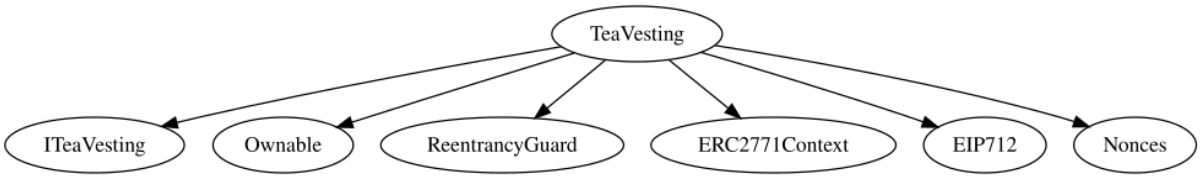
The owner is a multisig contract where the administrators are 3 people and to implement a transaction, at least 2 signatures are needed, which minimizes centralization. This function doesn't affect the main logic, it was added only for tokens not to be locked in smart-contract by mistake.

Functions Analysis

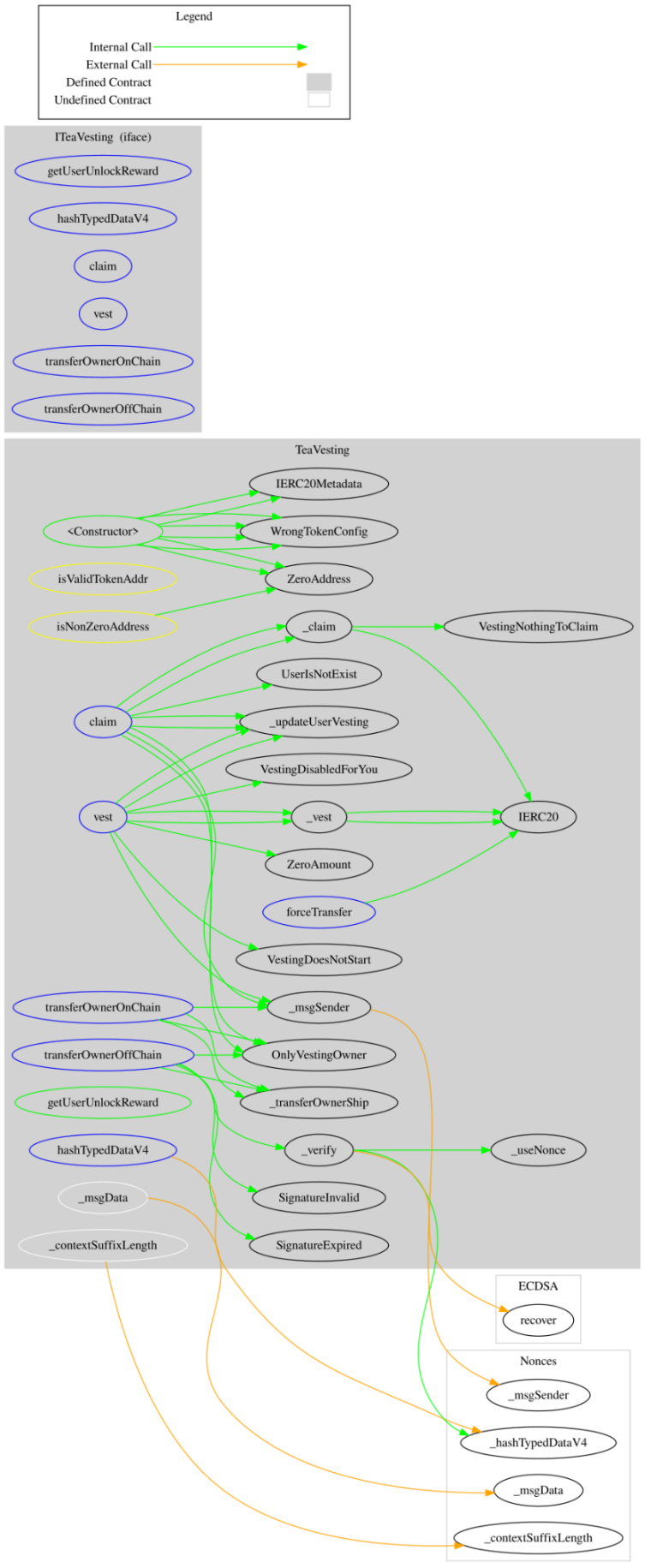
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
TeaVesting	Implementation	ITeaVesting, Ownable, ReentrancyGuard, ERC2771Context, EIP712, Nonces		
		Public	✓	Ownable EIP712 ERC2771Context
	claim	External	✓	isValidTokenAddr isNonZeroAddress nonReentrant
	vest	External	✓	isValidTokenAddr 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
IERC20Metadata
<Constructor>
WrongTokenConfig
isValidTokenAddr
ZeroAddress
isNonZeroAddress
_claim
VestingNothingToClaim
UserIsNotExist
claim
_updateUserVesting
VestingDisabledForYou
vest
_vest
IERC20
ZeroAmount
forceTransfer
VestingDoesNotStart
transferOwnerOnChain
_msgSender
transferOwnerOffChain
OnlyVestingOwner
getUserUnlockReward
_transferOwnerShip
hashTypedDataV4
_verify
_useNonce
_msgData
SignatureInvalid
_contextSuffixLength
SignatureExpired
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. The team has acknowledged the findings.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io