Cyberscope

Audit Report

Tea-Fi

July 2024

Files Presale.sol, PresaleToken.sol, Quoter.sol

Audited by © cyberscope

Table of Contents

Review

Testing Deploy	https://testnet.bscscan.com/address/0xd4c7070f0da65c5e7e01d3747868a52e06861515

Audit Updates

Initial Audit	28 Jun 2024 https://github.com/cyberscope-io/audits/blob/main/tea-fi/v1/audit.pdf
Corrected Phase 2	08 Jul 2024 https://github.com/cyberscope-io/audits/blob/main/tea-fi/v2/audit.pdf
Corrected Phase 3	11 Jul 2024

Source Files

Filename	SHA256
contracts/Quoter.sol	93c26acfdd0416ee7b9684c6e9608ae4e334ba7c1b48ae0b0348bc3ea7f90820
contracts/Presale.sol	27f8b561d0261b4dbbb00d7f7558bd8e37d77b573c1204fcdc313151d8315620
contracts/token/PresaleToken.sol	ba654de10617ba146fdcdc6b465443936bba2b5c0567e38360e5dd1ee6a2e9ea
contracts/token/IPresaleToken.sol	663ded2ba20243d5c6b0b6711192fec7f822eda125d0b23e5a53e95fd3b48cdd
@openzeppelin/contracts/utils/ReentrancyGuard.sol	8d0bac508a25133c9ff80206f65164cef959ec084645d1e7b06050c2971ae0fc

@openzeppelin/contracts/utils/Pausable.sol	6543160582b3c0319a180f31660faf6ba0a8444acbdb03357c09790a96256835
@openzeppelin/contracts/utils/Context.sol	847fda5460fee70f56f4200f59b82ae622bb03c79c77e67af010e31b7e2cc5b6
@openzeppelin/contracts/utils/Address.sol	b3710b1712637eb8c0df81912da3450da6ff67b0b3ed18146b033ed15b1aa3b9
@openzeppelin/contracts/token/ERC20/IERC20.sol	6f2faae462e286e24e091d7718575179644dc60e79936ef0c92e2d1ab3ca3cee
@openzeppelin/contracts/token/ERC20/ERC20.sol	2d874da1c1478ed22a2d30dcf1a6ec0d09a13f897ca680d55fb49fbcc0e0c5b1
@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol	471157c89111d7b9eab456b53ebe9042bc69504a64cb5cc980d38da9103379ae
@openzeppelin/contracts/token/ERC20/extensions/IERC20Permit.sol	912509e0e9bf74e0f8a8c92d031b5b26d2d35c6d4abf3f56251be1ea9ca946bf
@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol	1d079c20a192a135308e99fa5515c27acfbb071e6cdb0913b13634e630865939
@openzeppelin/contracts/interfaces/draft-IERC6093.sol	4aea87243e6de38804bf8737bf86f750443d3b5e63dd0fd0b7ad92f77cdbd3e3
@openzeppelin/contracts/access/Ownable.sol	38578bd71c0a909840e67202db527cc6b4e6b437e0f39f0c909da32c1e30cb81

Overview

The Presale contract is designed to facilitate the sale of tokens before they are officially launched. This contract allows users to purchase tokens using different cryptocurrencies, including Ether (ETH), by interacting with various presale options. The contract manages the entire lifecycle of a presale, from creating new presale options to handling token purchases and withdrawals. The presale process involves several key functionalities:

Token Purchase

Users can buy presale tokens by providing the necessary payment in either ETH or other ERC20 tokens. The purchase functions (buyExactPresaleTokens and buyExactPresaleTokensETH) calculate the exact payment amount required and execute the token purchase, ensuring that the user receives the correct amount of presale tokens. The contract also supports referral tracking, where users can specify a referrer ID, and the system tracks sales attributed to each referrer.

Presale Options Management

The contract allows the owner to create, initialize, and delete presale options. Each presale option is associated with specific parameters, such as the price per token, the amount available at the Token Generation Event (TGE), and the vesting schedule.

Payment Token Management

The contract enables the owner to add and remove payment tokens, specifying whether they are pegged to USD and defining their conversion paths for decentralized exchanges.

Withdrawal

The owner can withdraw funds collected from the presale, either in ETH or other ERC20 tokens, ensuring that the contract's balance is transferred to a specified multisig wallet for secure fund management.

Price Quoting

The contract includes functions to retrieve price quotes for token swaps using Uniswap V2. These quotes are used to determine the amount of tokens required or received during the presale transactions.

The Presale contract interacts with two other key contracts :

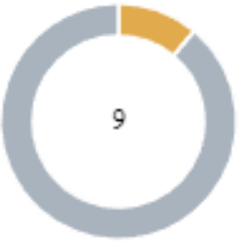
Quoter Contract

This contract is used to obtain price quotes for token swaps through Uniswap V2. It provides functions to get the expected amount of output tokens for a given input, aiding in the accurate calculation of payment amounts during the presale.

PresaleToken Contract

This contract represents the tokens being sold in the presale. It includes functionality for minting new tokens, which iscrucial for managing the supply of presale tokens as users make purchases.

Findings Breakdown



Critical	0
Medium	1
Minor / Informative	8

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	1	0	0
Minor / Informative	1	7	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	OMS	Oracle Manipulation Susceptibility	Acknowledged
Minor	ITM	Immediate Token Minting	Acknowledged
Minor	CCR	Contract Centralization Risk	Acknowledged
Minor	IRS	Incomplete Referral System	Acknowledged
Minor	ITAC	Inconsistent Token Availability Check	Unresolved
Minor	IPQC	Incorrect Price Quote Calculation	Acknowledged
Minor	MT	Mints Tokens	Acknowledged
Minor	PTAI	Potential Transfer Amount Inconsistency	Acknowledged
Minor	USM	Unused Struct Members	Acknowledged

OMS - Oracle Manipulation Susceptibility

Criticality	Medium
Location	contracts/Presale.sol#L420
Status	Acknowledged

Description

The function `getQuoteUniswapV2` is used to obtain price quotes for swaps through Uniswap V2. However, these quotes can be easily manipulated because they rely on Uniswap's on-chain price feeds, which are vulnerable to price manipulation attacks. This can occur through flash loan attacks, where an attacker borrows a large amount of tokens to manipulate the price within a single transaction, or through trading in low liquidity pools, where significant trades can disproportionately affect the price. As a result, the price quotes returned by these functions may be inaccurate, potentially leading to financial loss for users relying on them for transactions.

```
return
    uniswapV2Router.getAmountsOut(amountsIn,
paymentToken.path)[
        paymentToken.path.length - 1
    ];
```

Recommendation

It is recommended to mitigate this risk by employing strategies such as using a Time-Weighted Average Price (TWAP) instead of the current spot price from Uniswap. TWAP averages the price over a specified period, reducing the impact of short-term price manipulation. Another approach is to integrate more robust on-chain price oracles like Chainlink, which aggregate prices from multiple sources to provide a more reliable price feed. Additionally, implementing liquidity checks to ensure that the trading pairs have sufficient liquidity to withstand potential manipulation can further enhance the security and reliability of the price quotes used in the contract.

Team Update

The team has acknowledged that this is not a security issue and states: *Here is the token list to be used: ETH (WETH), USDC, DAI, WBTC, and USDT, which are blue chips and highly liquid assets with deep liquidity pools on Uniswap V2. This high liquidity makes price manipulation difficult and costly. These tokens are also widely accepted and used in the cryptocurrency market, therefore stable and accurate pricing is guaranteed. Any attempted price manipulation on Uniswap is quickly corrected by arbitrageurs who monitor and exploit price discrepancies across exchanges, reinforcing the accuracy of prices. Finally, the flash loan attack is not possible, as affecting the price in oracle the hacker can buy more Presale tokens, however, these presale tokens will not be listed anywhere and they don't have any market value, so the attacker will not be able to return the flash loan by the end of the transaction.*

ITM - Immediate Token Minting

Criticality	Minor / Informative
Location	contracts/Presale.sol#L797
Status	Acknowledged

Description

The Presale contract mints and transfers tokens to the buyer immediately upon purchase. This approach allows users to instantly receive and potentially sell their presale tokens at a different price on secondary markets before the presale event concludes. This can lead to market manipulation and undermine the intended price stability and distribution strategy of the presale.

```
function _buyExactTokens(
        uint8 optionId,
        uint32 referrerId,
        address tokenSell,
        address sender,
        uint256 tokenSellPrice,
        uint256 buyAmount,
        uint256 payAmount,
        uint256 amountInUsd
    ) private {
        Option storage option = saleOptions[optionId];
        Referral storage referrer = referrals[referrerId];

        // update the referrer statistics
        unchecked {
            referrer.sold += buyAmount;
            referrer.soldInUsd += amountInUsd;

            // inside of the contract there is out of scope to
check
            // if referral was added, it is done off-chain
            ++referrer.referrals;
        }

        // No need to check because of first check in top
        unchecked {
            totalSold += buyAmount;
            totalSoldInUsd += amountInUsd;
            option.sold += buyAmount;
            option.soldInUsd += amountInUsd;
        }

        // mint just a presale token which doesn't have any
market value before claim
        IPresaleToken(option.presaleToken).mint(sender,
buyAmount);

        emit BuyTokens(
            sender,
            tokenSell,
            option.presaleToken,
            optionId,
            referrerId,
            amountInUsd,
            payAmount,
            tokenSellPrice,
            option.price,
            buyAmount
        );
    }
```

Recommendation

To mitigate this issue, it is recommended to implement a delayed token distribution mechanism. Instead of minting and transferring tokens immediately, the contract should store the purchase details and distribute the tokens only after the presale event ends. This can be achieved by introducing a finalizePresale function, which the owner can call to trigger the distribution of tokens to all buyers. This approach ensures that tokens are distributed fairly and in line with the presale's goals, preventing premature trading and price manipulation.

Team Update

The team has acknowledged that this is not a security issue and states: *The immediately minted tokens are presale tokens that do not hold any intrinsic tokenomic value. While these tokens can be traded, their primary purpose is to allow holders to claim their real TEA tokens later. Immediate token distribution also rewards early participants, offering them the chance to benefit from initial trading opportunities and providing an incentive for early investment. Overall, these tokens don’t have any market value by default, they will be exchanged to real TEA tokens during the claim phase.*

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	contracts/Presale.sol#L531,536,544,567,579,626,640,659 contracts/token/PresaleToken.sol#L39,51
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 contract owners have the authority to make changes to key variables that heavily impact the functionality of the contracts. Furthermore, the contract owner of the Presale contract can withdraw any amount of tokens and Ether. Lastly, they can pause and unpause the purchase of tokens in the presale.

```
function withdraw(address token) external onlyOwner {
        uint256 balance;
        if (token == address(0)) {
            balance = address(this).balance;
            (bool succeed, ) = multisigWallet.call{value:
balance}("");

            if (!succeed) {
                revert WithdrawFailed();
            }
        } else {
            balance = IERC20(token).balanceOf(address(this));
            IERC20(token).safeTransfer(multisigWallet,
balance);
        }

        emit Withdraw(multisigWallet, token, balance);
    }

function deleteOption(
        uint8 optionId
    ) external onlyOwner checkIfOptionInitialized(optionId) {
        // no need to check because of modifier check
        unchecked {
            --saleOptionsCount;
        }

        delete saleOptions[optionId];
        emit OptionDeleted(optionId);
    }

...
```

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: *Although the deployer is initially considered the owner when the Presale contract is deployed, ownership is subsequently transferred to a multisig wallet. This wallet has more than three owners and requires more than two signatures to authorize any transaction. For the Presale Tokens contract, the token owner is the owner of the Presale contract. These measures ensure that centralization risks are minimal in both scenarios, as control is distributed among multiple stakeholders.*

IRS - Incomplete Referral System

Criticality	Minor / Informative
Location	contracts/Presale.sol#L145,797
Status	Acknowledged

Description

The referral system in the Presale contract appears to be incomplete and ineffective. The `Referral` struct is defined to track the number of referrals made by a user, the amount of tokens sold through these referrals, and the total USD equivalent of these tokens. However, there are several issues with the current implementation. There is no functionality to add referrers to the system. Even if users buy the presale token, they are not iadded as referrers. The contract does not provide any mechanism to register a user as a referrer or to validate the existence of a referrer before making a purchase. Furthermore, users can call the `buyExactPresaleTokens` and `buyExactPresaleTokensETH` functions without providing a `referrerId` or by providing different `referrerId` values each time. This lack of consistency and validation makes the referral system unreliable. Lastly, the referral tracking is limited to incrementing the referrals count and updating the `sold` and `soldInUsd` fields within the `_buyExactTokens` function. However, this does not ensure that the referrer benefits from the referral.

```
struct Referral {

        /// @dev Number of purchases
        uint16 referrals;
        /// @dev The amount of tokens sold through referrals
        uint256 sold;
        /// @dev The total amount of USD equivalent of tokens
sold through referrals
        uint256 soldInUsd;
    }


function _buyExactTokens(
        uint8 optionId,
        uint32 referrerId,
        address tokenSell,
        address sender,
        uint256 tokenSellPrice,
        uint256 buyAmount,
        uint256 payAmount,
        uint256 amountInUsd
    ) private {
        Option storage option = saleOptions[optionId];
        Referral storage referrer = referrals[referrerId];

        // update the referrer statistics
        unchecked {
            referrer.sold += buyAmount;
            referrer.soldInUsd += amountInUsd;

            // inside of the contract there is out of scope to
check
            // if referral was added, it is done off-chain
            ++referrer.referrals;
        }

        // No need to check because of first check in top
        unchecked {
            totalSold += buyAmount;
            totalSoldInUsd += amountInUsd;
            option.sold += buyAmount;
            option.soldInUsd += amountInUsd;
        }

        // mint just a presale token which doesn't have any
market value before claim
        IPresaleToken(option.presaleToken).mint(sender,
buyAmount);

        emit BuyTokens(
```

```
            sender,
            tokenSell,
            option.presaleToken,
            optionId,
            referrerId,
            amountInUsd,
            payAmount,
            tokenSellPrice,
            option.price,
            buyAmount
        );
    }
```

Recommendation

To improve the referral system, it is recommended to implement a comprehensive and reliable mechanism for managing referrers. This should include functions to add and validate referrers, ensuring that users are registered as referrers before they can refer others. Additionally, the referral tracking logic should be enhanced to ensure consistency and accuracy. By addressing these issues, the referral system can become a valuable and effective feature of the presale contract, encouraging more users to participate and refer others.

Team Update

The team has acknowledged that this is not a security issue and states: *The referral logic is handled front-end/back-end, as on-chain there is only statistics.*

ITAC - Inconsistent Token Availability Check

Criticality	Minor / Informative
Location	contracts/Presale.sol#L272
Status	Unresolved

Description

The `checkTokensRemain` modifier aims to ensure that sufficient tokens are available for a transaction by comparing the total tokens sold against the available tokens for each presale option. However, the calculation depends on the `saleOptionsCount` variable, which can be dynamically increased or decreased by the contract owner through the createNewOption and deleteOption functions. This dynamic adjustment can lead to inconsistencies in token availability checks. For instance, if a user tries to buy tokens when the total available per option is calculated based on a higher `saleOptionsCount`, the transaction might revert due to insufficient tokens. However, if the owner subsequently deletes an option, reducing the `saleOptionsCount`, the same transaction might succeed as the denominator decreases, making more tokens available per option. Conversely, creating new options increases the `saleOptionsCount`, potentially reducing the available tokens per option and leading to failed transactions that might have succeeded previously.

```
modifier checkTokensRemain(uint8 optionId, uint256 amount) {
        uint256 totalAvailablePerOption = saleOptionsCount != 1
            ? tokensAvailableForPresale / saleOptionsCount
            : tokensAvailableForPresale;

        uint256 soldAmount = saleOptions[optionId].sold +
amount;

        if (soldAmount > totalAvailablePerOption) {
            revert NotEnoughTokensLeftPerOption(
                soldAmount,
                totalAvailablePerOption
            );
        }

        _;
    }
```

Recommendation

To ensure consistent and reliable token availability checks, it is recommended to decouple the token availability logic from the dynamic `saleOptionsCount` variable. One approach could be to set a fixed cap on the total tokens available per option when the option is created and ensure that this cap remains unchanged regardless of the total number of options. This approach will ensure that token availability checks are accurate and independent of changes in the number of presale options, thereby providing a more predictable and fair experience for users. Additionally, the logic for creating and deleting options should be thoroughly reviewed to prevent potential manipulation of token availability through changes in the `saleOptionsCount` variable.

IPQC - Incorrect Price Quote Calculation

Criticality	Minor / Informative
Location	contracts/Presale.sol#L408
Status	Acknowledged

Description

The `inputPriceQuote` function is designed to retrieve the price quote for a given input amount of tokens. However, the current implementation does not correctly handle stablecoins. When the payment token is a stablecoin, the function incorrectly returns 1 instead of the actual amount of stablecoins corresponding to the given `amountsIn`. For stablecoins, the function should return the input amount adjusted by the decimal places of the stablecoin. For instance, if `amountsIn` is given in USD with 6 decimals and the payment token is a stablecoin with 6 decimals, the return value should be `amountsIn`. If the payment token is a stablecoin with 18 decimals, the return value should be `amountsIn` adjusted by the difference between decimals.

```
function inputPriceQuote(
        address token,
        uint256 amountsIn
    ) public view returns (uint256) {
        PaymentTokenType memory paymentToken =
salePaymentTokens[token];

        if (!paymentToken.allowed) {
            return 0;
        } else if (paymentToken.peggedToUsd) {
            return 1;
        }

        return
            uniswapV2Router.getAmountsOut(amountsIn,
paymentToken.path)[
                paymentToken.path.length - 1
            ];
    }
```

Recommendation

The function should be updated to correctly handle stablecoins by adjusting the return value based on the decimal places of the stablecoin. Instead of returning a fixed value of 1 for stablecoins pegged to USD, the function should return the actual amount of stablecoins that correspond to the given amountsIn. This adjustment ensures that the price quote accurately reflects the value of stablecoins and prevents potential miscalculations in transactions involving stablecoins. Implementing these changes will improve the accuracy and reliability of the price quote calculations for stablecoins in the presale contract.

Team Update

The team has acknowledged that this is not a security issue and states: *This logic is used and handled only in the front-end, which shows $1 for approved stablecoins.*

MT - Mints Tokens

Criticality	Minor / Informative
Location	contracts/token/PresaleToken.sol#L51
Status	Acknowledged

Description

The operator of the PresaleToken that is a role managed by the contract owner has the authority to mint tokens. They may take advantage of it by calling the `mint` function. As a result, the contract tokens will be highly inflated.

```
function mint(address to, uint256 amount) external {
    if(_msgSender() != operator) {
        revert CallerIsNotOperator();
    }

    _mint(to, amount);

    emit Minted(to, amount);
}
```

Recommendation

The team should carefully manage the private keys of the owner’s account and the operator's. 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: *Presale tokens are exclusively deployed within the Presale smart contract in initOptions function after deployment. Upon deployment, the Presale contract itself is designated as the operator, thus tokens are minted strictly according to the core business logic (which requires the payment before minting). Moreover, the presale token owner is a multisig wallet, making unauthorized transfers of the operator role impossible and further safeguarding against token inflation.*

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	contracts/Presale.sol#L457
Status	Acknowledged

Description

The `safeTransferFrom()` function is used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
IERC20(tokenSell).safeTransferFrom(
    _msgSender(),
    address(this),
    payAmount
);
```

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: *To address this issue, we want to clarify that our contract utilizes USDT, WBTC, USDC, DAI, and native ETH as payment tokens. These tokens do not charge any transfer fees. Further on, we ensure that all token transfers will be conducted with standard ERC20 tokens that do not include any additional fees or taxes. Our commitment to using fee-free ERC20 tokens maintains the integrity and predictability of token transfers within the system.*

USM - Unused Struct Members

Criticality	Minor / Informative
Location	contracts/token/PresaleToken.sol#L141,145
Status	Acknowledged

Description

The `Option` struct in the `Presale` contract includes two members, `tgeAmount` and `leftoverVesting`, which are not utilized anywhere in the contract. The presence of these unused members can lead to confusion, suggesting incomplete implementation or future features that were not fully integrated. Additionally, unused variables occupy unnecessary storage space and can clutter the codebase, reducing its readability and maintainability.

```
struct Option {
        /// @dev The percentage of tokens available at the time
of TGE (Token Generation Event)
        /// NOTE The param is used outside of this contract
        uint8 tgeAmount;

        /// @dev The percentage of tokens that will be vested
over time after TGE
        /// NOTE The param is used outside of this contract
        uint8 leftoverVesting;

        /// @dev The price per token in the presale
        uint8 price;

        /// @dev The address of the token being sold in the
presale
        address presaleToken;

        /// @dev The amount of tokens that have been sold so
far
        uint256 sold;

        /// @dev The total amount of USD equivalent of tokens
sold
        uint256 soldInUsd;
    }
```

Recommendation

It is recommended to either remove the `tgeAmount` and `leftoverVesting` members from the `Option` struct if they are not intended to be used, or implement their functionality if they are meant to play a role in the presale process. This will help in maintaining a clean and efficient codebase, reducing potential confusion, and ensuring that all struct members serve a purpose in the contract.

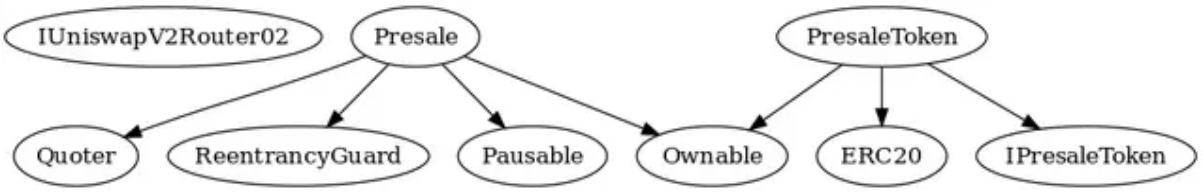
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IUniswapV2Router02	Interface			
	getAmountsOut	External		-
	getAmountsIn	External		-
Quoter	Implementation			
		Public	✓	-
Presale	Implementation	Ownable, ReentrancyGuard, Pausable, Quoter		
		Public	✓	Ownable Quoter
	getExactPayAmount	External		-
	getExactReceiveAmount	External		-
	getOptionInfo	External		-
	inputPriceQuote	Public		-
	buyExactPresaleTokens	External	✓	whenNotPaused checkIfOptionInitialized checkIfTokenAllowed checkTokensRemain nonReentrant

	buyExactPresaleTokensETH	External	Payable	whenNotPaused checkIfOptionInitialized checkIfTokenAllowed checkTokensRemain nonReentrant
	pause	External	✓	onlyOwner
	unpause	External	✓	onlyOwner
	addPaymentToken	External	✓	onlyOwner
	removePaymentToken	External	✓	onlyOwner checkIfTokenAllowed
	createNewOption	External	✓	onlyOwner
	deleteOption	External	✓	onlyOwner checkIfOptionInitialized
	withdraw	External	✓	onlyOwner
	initOptions	External	✓	onlyOwner
	_calculatePayAmount	Private		
	_deployPresaleTokenForOption	Private	✓	
	_buyExactTokens	Private	✓	
	_getDecimals	Private		
	_expandToDecimals	Private		
PresaleToken	Implementation	ERC20, Ownable, IPresaleToken		
		Public	✓	Ownable
	isOperator	External		-

	transferOperator	External	✓	onlyOwner
	mint	External	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IPresaleToken (iface)
operator
isOperator
transferOperator
mint
PresaleToken
<Constructor>
InvalidOperatorAddress
isOperator
TransferToSameOperator
transferOperator
_msgSender
mint
_mint
CallerIsNotOperator
Quoter
<Constructor>
Presale
checkIfOptionInitialized
OptionNotInitialized
checkIfTokenAllowed
PaymentTokenNotAuthorized
checkTokensRemain
NotEnoughTokensLeftPerOption
pause
_pause
getExactReceiveAmount
InvalidMultisigWalletAddress
<Constructor>
OptionIsAlreadyCreated
getOptionInfo
OptionCreationIsOverflow
_getDecimals
IERC20Metadata
createNewOption
owner
initOptions
_deployPresaleTokenForOption
AddressMustBeDifferentFromAddressZero
getExactPayAmount
_calculatePayAmount
_expandToDecimals
buyExactPresaleTokens
_buyExactTokens
buyExactPresaleTokensETH
_msgSender
payable
unpause
InsufficientFunds
removePaymentToken
_unpause
addPaymentToken
PaymentTokenAlreadyAuthorized
deleteOption
withdraw
IERC20
inputPriceQuote
WithdrawFailed
IUniswapV2Router02 (iface)
getAmountsOut
getAmountsIn

Summary

Tea-Fi contract implements a presale mechanism. This audit investigates security issues, business logic concerns and potential improvements.

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

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

https://www.cyberscope.io