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/0xdbf86f039d1d95c6e388279042afc0812b825537 |

## Audit Updates

| | |
|---|---|
| **Initial Audit** | 28 Jun 2024<br>https://github.com/cyberscope-io/audits/blob/main/tea-fi/v1/audit.pdf |
| **Corrected Phase 2** | 08 Jul 2024 |

## Source Files

| Filename | SHA256 |
|---|---|
| **contracts/Quoter.sol** | 22f965c5b759f17ea457f649b01a5dc49d5f32587c2f419eb997534ad23739c0 |
| **contracts/Presale.sol** | 4f09807a0093034d8573d18ee198d62f4343605c9d9abce9975dfcc114f8e9c5 |
| **contracts/token/PresaleToken.sol** | a08442cc6b1082ca14fdca3bb6c39bf4cc2e39e74003a93d26ad502cd6aa6a5a |
| **contracts/token/IPresaleToken.sol** | 1bf9f5f519be18d268900de2b64ea75fff9d3c954cbee38bfb9fb943112ae77b |
| **@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** | ddff96777a834b51a08fec26c69bb6ca2d01d150a3142b3fdd8942e07921636a |
| **@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 :

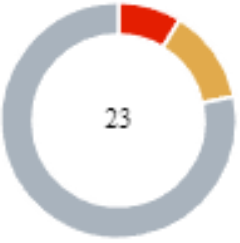
**Quoter Contract**

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 | 2 |
| Medium | 3 |
| Minor / Informative | 18 |

| Severity | Unresolved | Acknowledged | Resolved | Other |
|---|---|---|---|---|
| Critical | 2 | 0 | 0 | 0 |
| Medium | 3 | 0 | 0 | 0 |
| Minor / Informative | 16 | 2 | 0 | 0 |

# Diagnostics

Critical Medium Minor / Informative

| Severity | Code | Description | Status |
|---|---|---|---|
| ● | IDC | Inaccurate Decimal Compression | Unresolved |
| ● | IDH | Incorrect Decimal Handling | Unresolved |
| ● | ITM | Immediate Token Minting | Unresolved |
| ● | MT | Mints Tokens | Unresolved |
| ● | OMS | Oracle Manipulation Susceptibility | Unresolved |
| ● | CCR | Contract Centralization Risk | Unresolved |
| ● | IRS | Incomplete Referral System | Acknowledged |
| ● | ITAC | Inconsistent Token Availability Check | Unresolved |
| ● | IMU | Incorrect Modifier Usage | Unresolved |
| ● | IPQC | Incorrect Price Quote Calculation | Unresolved |
| ● | IDTU | Inefficient Data Types Usage | Unresolved |
| ● | IPU | Inefficient Parameter Usage | Unresolved |
| ● | MCV | Missing Constructor Validation | Unresolved |
| ● | MDC | Missing Decimal Checks | Unresolved |

| | | | |
|---|---|---|---|
| ● | MTI | Missing Token Initialization | Unresolved |
| ● | PDS | Potential Duplicate Salt | Unresolved |
| ● | PTAI | Potential Transfer Amount Inconsistency | Unresolved |
| ● | REE | Redundant Event Emission | Unresolved |
| ● | USM | Unused Struct Members | Acknowledged |
| ● | L04 | Conformance to Solidity Naming Conventions | Unresolved |
| ● | L11 | Unnecessary Boolean equality | Unresolved |
| ● | L16 | Validate Variable Setters | Unresolved |
| ● | L19 | Stable Compiler Version | Unresolved |

# IDC - Inaccurate Decimal Compression

| | |
|---|---|
| **Criticality** | Critical |
| **Location** | contracts/Presale.sol#L733 |
| **Status** | Unresolved |

## Description

The `_calculatePayAmount` function in the Presale contract contains an inconsistency in the comments and calculations related to decimal compression. The comment in the function suggests a formula for calculating the number of decimals for USD-pegged amounts, but the formula provided is incorrect. The comment states that `decimalsUSD = 18 - 2 - (18 - 6)`, which results in 4, not 6 as indicated. Furthermore, the calculation that follows adds 12 to the `DELTA_PRESALE_TOKEN_PRICE_SCALE_DIVISOR`, likely resulting from subtracting 18 from the 6 mentioned in the comment. This inconsistency indicates a misunderstanding in the decimal handling logic, which may lead to incorrect calculations for USD-pegged token amounts.

```
function _calculatePayAmount(
        uint8 optionId,
        address tokenSell,
        uint256 buyAmount
    ) private view returns (
        uint256 payAmount,
        uint256 optionTokenAmountInUsd,
        uint256 tokenSellPrice
    ) {
        Option memory option = saleOptions[optionId];

        uint8 presaleTokenDecimals =
_getDecimals(option.presaleToken);
        uint8 tokenSellDecimals = _getDecimals(tokenSell);
        uint8 delta = _getDecimalsDelta(presaleTokenDecimals,
tokenSellDecimals);
        uint256 presaleTokenAmountInUsd;

        unchecked {
            presaleTokenAmountInUsd = buyAmount * option.price;
        }

        // for stats price pegged to USD uses 6 decimals only.
        // As presale token price is 18 decimal - USD pegged
amound
        // should be compressed to 6 dicimals.
        //
        // formula: decimalsUSD = presaleTokenDecimals -
optionTokenPriceDecimals - delta =>
        // decimalsUSD = 18 - 2 - (18 - 6) => decimalsUsd = 6
        uint256 payAmountInUsdPegged = _compressToDecimals(
            DELTA_PRESALE_TOKEN_PRICE_SCALE_DIVISOR + 12,
            presaleTokenAmountInUsd
        );


        if(salePaymentTokens[tokenSell].peggedToUsd == true) {
            uint256 payAmountSub = _compressToDecimals(
                delta +
DELTA_PRESALE_TOKEN_PRICE_SCALE_DIVISOR,
                presaleTokenAmountInUsd
            );

            return (
                payAmountSub,
                payAmountInUsdPegged,
                1
            );
        }
```

```
        PaymentTokenType memory saleToken =
salePaymentTokens[tokenSell];
        uint256 lastInPathQuoteAmount = 10**tokenSellDecimals;
        uint256 quoteAmountInUsd = inputPriceQuote(tokenSell,
lastInPathQuoteAmount);
        uint8 lastInPathTokenDecimals =
_getTokenOutDecimals(saleToken);
        uint8 deltaForQuotedAmount = _getDecimalsDelta(
            lastInPathTokenDecimals,
            DELTA_PRESALE_TOKEN_PRICE_SCALE_DIVISOR
        );



        uint256 expandedPrice = _expandToDecimals(
            deltaForQuotedAmount,
            presaleTokenAmountInUsd
        );

        uint256 compressedPayAmount = _compressToDecimals(
            delta,
            expandedPrice / quoteAmountInUsd
        );


        return (
            compressedPayAmount,
            payAmountInUsdPegged,
            quoteAmountInUsd
        );
    }
```

## Recommendation

It is recommended to thoroughly review and correct the logic and comments related to decimal compression in the `_calculatePayAmount` function. Ensure that the formula for calculating the USD-pegged decimals is accurate and aligns with the intended logic. The comments should clearly and correctly explain the purpose and result of the calculations. Additionally, consider revising the calculations to avoid adding arbitrary constants, ensuring that all operations are mathematically sound and based on clear, consistent logic. This will help prevent potential errors in token amount calculations and improve the overall reliability of the contract.

# IDH - Incorrect Decimal Handling

| | |
|---|---|
| **Criticality** | Critical |
| **Location** | contracts/Presale.sol#L935 |
| **Status** | Unresolved |

## Description

The Presale contract uses a function `_getDecimalsDelta` to calculate the difference in decimals between two tokens. This function currently returns the absolute difference between the decimals, which does not account for whether the first token has more or fewer decimals than the second. This can lead to incorrect scaling of token amounts in subsequent calculations. For instance, if the presale token has 18 decimals and the other token has 24 decimals, the function will return 6. Conversely, if the presale token has 18 decimals and the other token has 12 decimals, it will also return 6. This lack of distinction can cause incorrect handling of decimal scaling, leading to potential inaccuracies in the amount of tokens received or paid during transactions. This issue is particularly relevant since there are tokens with more than 18 decimals.

```
function _getDecimalsDelta(
        uint8 decimals0,
        uint8 decimals1
    ) private pure returns(uint8) {
        if(decimals0 == decimals1) {
            return 0;
        }

        uint8 delta;

        unchecked {
            delta =
                decimals0 > decimals1 ?
                decimals0 - decimals1 :
                decimals1 - decimals0
            ;
        }
        return delta;
    }
```

## Recommendation

To ensure accurate and reliable calculations, the function should be modified to return a value that indicates both the magnitude and the direction of the difference between token decimals. This can be achieved by returning a signed integer. By making this change, the contract will be able to correctly handle the scaling up or down of token amounts based on whether the presale token has more or fewer decimals than the other token. This will enhance the accuracy and reliability of the contract’s calculations.

## ITM - Immediate Token Minting

| | |
|---|---|
| **Criticality** | Medium |
| **Location** | contracts/Presale.sol#L838 |
| **Status** | Unresolved |

### 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];

        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;
        }


        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.

## MT - Mints Tokens

| | |
|---|---|
| **Criticality** | Medium |
| **Location** | contracts/token/PresaleToken.sol#L57 |
| **Status** | Unresolved |

### 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.

## OMS - Oracle Manipulation Susceptibility

| | |
|---|---|
| **Criticality** | Medium |
| **Location** | contracts/Quoter.sol#L25 |
| **Status** | Unresolved |

## 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.

```
function getQuoteUniswapV2(
    uint256 amountsIn,
    address[] memory path
) internal view returns(uint256) {
    return UNISWAP_V2_ROUTER.getAmountsOut(amountsIn,
path)[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.

## CCR - Contract Centralization Risk

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Presale.sol#L554,559,569,592,606,656,672,695<br>contracts/token/PresaleToken.sol#L44,57 |
| **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 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,) = MULTISIG_WALLET.call{value:
balance}("");

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

        emit Withdraw(MULTISIG_WALLET, 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);
    }

function pause() external onlyOwner {
        _pause();
    }

function unpause() external onlyOwner {
        _unpause();
    }

...
```

## 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.

## IRS - Incomplete Referral System

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Presale.sol#L164,838 |
| **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 The number of referrals made by the user
        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 buyAmount,
        uint256 payAmount,
        uint256 amountInUsd,
        Option storage option
    ) private {
        Referral storage referrer = referrals[referrerId];

        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;
        }


        ...
    }
```

## 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.

## ITAC - Inconsistent Token Availability Check

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Presale.sol#L282,648,661 |
| **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
?
            TOKENS_AVAILABLE_FOR_PRESALE / saleOptionsCount :
            TOKENS_AVAILABLE_FOR_PRESALE;

        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.

# IMU - Incorrect Modifier Usage

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Presale.sol#L275,473 |
| **Status** | Unresolved |

## Description

The `checkTokensRemain` modifier is designed to ensure that there are sufficient tokens available for a transaction by checking if the total tokens sold plus the transaction amount exceed the tokens available for presale. This modifier is correctly used in the `buyExactPresaleTokens` function. However, in the `buyExactPresaleTokensETH` function, the modifier uses `msg.value` to determine the transaction amount. Using `msg.value` is inappropriate here because `msg.value` represents the amount of Ether sent in the transaction, not the number of tokens being purchased. This discrepancy can lead to incorrect checks and potentially allow transactions that should not be permitted if the tokens available for presale are insufficient.

```
modifier checkTokensRemain(uint256 amount) {
        if (totalSold + amount > TOKENS_AVAILABLE_FOR_PRESALE)
{
            revert NotEnoughTokensLeft(amount,
TOKENS_AVAILABLE_FOR_PRESALE - totalSold);
        }
        _;
    }

function buyExactPresaleTokensETH(
        uint8 optionId,
        uint32 referrerId,
        uint256 buyAmount
    )
    external
    payable
    whenNotPaused
    whenOptionCreated(optionId)
    whenTokenAllowed(address(0))
    checkTokensRemain(msg.value) {
        // get receive token price first
        (uint256 payAmount, uint256 optionTokenAmountInUsd,
Option storage option) = _calculatePayAmount(
            optionId,
            address(0),
            buyAmount
        );

        address sender = _msgSender();
        uint256 transferValue = msg.value;

        ...
}
```

## Recommendation

It is recommended to modify the `buyExactPresaleTokensETH` function to use the correct amount of tokens being purchased for the `checkTokensRemain` modifier. This will ensure that the token availability checks are accurate and consistent with the intended logic of the presale contract. The logic should be revised to ensure the correct variable representing the token amount is passed to the modifier, preventing any potential issues related to incorrect token availability checks.

## IPQC - Incorrect Price Quote Calculation

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Presale.sol#L406 |
| **Status** | Unresolved |

## 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 == false) {
            return 0;
        } else if(paymentToken.peggedToUsd == true) {
            return 1;
        }

        return getQuoteUniswapV2(amountsIn, paymentToken.path);
    }
```

## 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.

## IDTU - Inefficient Data Types Usage

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Presale.sol#L138,164,198 |
| **Status** | Unresolved |

### Description

The `Presale` contract uses `uint8` data types for several variables, including `saleOptionsCount`, and members of the `Option` struct such as `tgeAmount`, `leftoverVesting`, and `price`. While using smaller data types like `uint8` can potentially save storage space by packing multiple values into a single 32-byte storage slot, it can also lead to higher gas usage. This is because the Ethereum Virtual Machine (EVM) operates on 32-byte words, and when dealing with elements smaller than 32 bytes, additional operations are required to handle the smaller sizes, leading to increased gas consumption. In scenarios where these variables are not read or written together frequently, the benefit of packing may be outweighed by the extra gas cost incurred from these additional operations.

```
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;
    }

uint8 public saleOptionsCount;
```

## Recommendation

It is recommended to use `uint256` instead of smaller data types like `uint8` for variables such as `saleOptionsCount` and struct members. This change will align with the EVM's native word size and can lead to more efficient gas usage during contract execution. The potential savings in storage should be carefully weighed against the gas cost implications to determine the best approach for optimizing the contract's performance and cost.

# IPU - Inefficient Parameter Usage

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Quoter.sol#L25 |
| **Status** | Unresolved |

## Description

The function `getQuoteUniswapV2` of the Quoter contract utilize the memory keyword for their path parameters. This function is declared as internal and is subsequently invoked by public functions in the `Presale` contract. When public functions are called externally, their parameters are stored in calldata. Using calldata instead of memory in internal functions can reduce gas consumption, as it avoids the need to copy data from calldata to memory.

```
function getQuoteUniswapV2(
        uint256 amountsIn,
        address[] memory path
    ) internal view returns(uint256) {
        return UNISWAP_V2_ROUTER.getAmountsOut(amountsIn,
path)[path.length - 1];
    }
```

## Recommendation

To optimize gas efficiency, it is recommended to change the parameter type of path in the `getQuoteUniswapV2` function from memory to calldata. This change will allow the contract to handle external calls more efficiently by directly accessing data from calldata without unnecessary copying operations. Making this adjustment will contribute to lower transaction costs and improved overall performance of the contract.

# MCV - Missing Constructor Validation

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Presale.sol#L234 |
| **Status** | Unresolved |

## Description

The constructor of the `Presale` contract initializes the `saleOptions` without performing any validation checks to ensure that the options are created correctly. This lack of validation can lead to incorrect or unintended configuration of the presale options, which cannot be modified later due to the absence of functions to modify them post-deployment. In contrast, the `createNewOption` function, which allows the owner to create new presale options, includes comprehensive checks to ensure the validity of the options. These checks prevent issues such as duplicate option creation and incorrect option configuration.

```
constructor(
        address _multisigWallet,
        IUniswapV2Router02 _uniswapRouterV2,
        uint256 tokensAvailableForPresale,
        Option[] memory options
    ) Ownable(msg.sender) Quoter(_uniswapRouterV2) {
        MULTISIG_WALLET = _multisigWallet;
        TOKENS_AVAILABLE_FOR_PRESALE =
tokensAvailableForPresale;

        // init options
        for(uint8 i; i < options.length;) {
            saleOptions[i] = options[i];
            unchecked { ++i; }
        }

        saleOptionsCount = uint8(options.length);
        _pause();
    }
```

## Recommendation

It is recommended to incorporate similar validation checks in the constructor as those used in the `createNewOption` function. Alternatively, the constructor can utilize the `createNewOption` function to ensure that the presale options are validated correctly during initialization. Implementing these checks will ensure that the presale options are configured correctly and consistently, preventing potential issues and ensuring the integrity of the contract's setup.

# MDC - Missing Decimal Checks

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Presale.sol#L569 |
| **Status** | Unresolved |

## Description

The function `addPaymentToken` lacks checks to verify the decimals of tokens when `peggedToUsd` is set to true and when handling tokens in the provided path. Specifically, when `peggedToUsd` is true, it should be ensured that the token has 6 decimals. Similarly, in other parts of the contract, there are calculations involving token decimals that do not verify the actual decimals of the involved tokens. This can lead to incorrect calculations and potential errors during token transfers and conversions.

```
function addPaymentToken(
        bool peggedToUsd,
        address token,
        address[] memory path
    )
    external
    onlyOwner {
        if (salePaymentTokens[token].allowed == true) {
            revert PaymentTokenAlreadyAuthorized(token);
        }
        if(peggedToUsd == true) {
            salePaymentTokens[token] = PaymentTokenType(true,
true, new address[](0));
        } else {
            salePaymentTokens[token] = PaymentTokenType(false,
true, path);
        }

        emit AddPaymentToken(token);
    }
```

## Recommendation

Introduce checks to ensure that tokens pegged to USD have 6 decimals when peggedToUsd is set to true. Additionally, verify the decimals of all tokens in the path to ensure consistency and correctness in calculations. Implement these checks within the addPaymentToken function and other relevant parts of the contract where token decimals are used. This will ensure accurate token conversions and prevent potential issues arising from incorrect decimal assumptions.

## MTI - Missing Token Initialization

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Presale.sol#L491 |
| **Status** | Unresolved |

### Description

The function `buyExactPresaleTokensETH` assumes that `address(0)` is present in the salePaymentTokens list and represents WETH. However, the constructor does not initialize `address(0)` in the `salePaymentTokens` list, leading to potential issues during execution. This oversight can cause the function to fail when users attempt to purchase presale tokens using ETH, as the required checks and configurations for `address(0)` are not set up.

```
function buyExactPresaleTokensETH(
        uint8 optionId,
        uint32 referrerId,
        uint256 buyAmount
    )
    external
    payable
    whenNotPaused
    checkIfOptionInitialized(optionId)
    checkIfTokenAllowed(address(0))
    checkTokensRemain(optionId, buyAmount)
    nonReentrant {
        uint256 buyAmountCopy = buyAmount;

        // get receive token price first
        (
            uint256 payAmount,
            uint256 optionTokenAmountInUsd,
            uint256 tokenSellPrice
        ) = _calculatePayAmount(
            optionId,
            address(0),
            buyAmountCopy
        );


        address sender = _msgSender();

        {
            uint256 transferValue = msg.value;
            if(transferValue < payAmount) {
                revert InsufficientFunds();
            } else if(transferValue > payAmount) {
                // refund exceeded funds
                (bool succeed,) = payable(sender).call{value:
transferValue - payAmount}("");
                require(succeed, "Failed to withdraw Ether");
            }
        }


        _buyExactTokens(
            optionId,
            referrerId,
            address(0),
            sender,
            tokenSellPrice,
            buyAmountCopy,
            payAmount,
            optionTokenAmountInUsd
```

```
        );
    }
```

## Recommendation

Ensure that `address(0)`, representing WETH, is properly initialized in the salePaymentTokens list during the contract's initialization. This can be done in the constructor or through a dedicated setup function. Proper initialization will ensure that all necessary checks and configurations for WETH are in place, allowing the function buyExactPresaleTokensETH to operate correctly. Implementing this change will prevent potential failures and ensure that users can reliably use ETH to participate in the presale.

# PDS - Potential Duplicate Salt

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Presale.sol#L695,814 |
| **Status** | Unresolved |

## Description

The function initOptions contains a for loop that calls `_deployPresaleTokenForOption` to deploy presale tokens. If the same `optionId` is provided multiple times within the same block, the salt generated for the `create2` opcode will be identical, leading to potential conflicts or failures in token deployment. This issue arises because the salt is derived from the `optionId` and the current block number, which remains constant within the same block.

```
function initOptions(uint8[] memory optionIds)
    external
    onlyOwner {
        address operator = address(this);
        address owner = owner();

        for(uint256 i; i < optionIds.length;) {
            uint8 optionId = optionIds[i];

            if(saleOptions[optionId].presaleToken ==
address(0)) {
                address presaleToken =
_deployPresaleTokenForOption(
                    optionId,
                    owner,
                    operator
                );

                saleOptions[optionId].presaleToken =
presaleToken;
            }

            unchecked { ++i; }
        }
    }

function _deployPresaleTokenForOption(
        uint8 optionId,
        address owner,
        address operator
    )
    private
    returns(address) {
        bytes32 salt = keccak256(abi.encodePacked(optionId,
block.number));
        address presaleToken = address(new PresaleToken{salt:
salt}(owner, operator));

        emit PresaleTokenCreated(presaleToken);
        return presaleToken;
    }
```

## Recommendation

It is recommended to implement additional checks or unique identifiers to ensure that each `optionId` is processed only once per block. This can be achieved by validating the uniqueness of `optionId` before deploying the presale token or by modifying the salt generation logic to include an additional unique component. This will prevent the deployment of multiple presale tokens with the same `optionId` in the same block, ensuring the integrity and uniqueness of each deployment.

## PTAI - Potential Transfer Amount Inconsistency

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Presale.sol#L467 |
| **Status** | Unresolved |

## Description

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

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

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

```
IERC20(tokenSellCopy).safeTransferFrom(
                sender,
                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
```

# REE - Redundant Event Emission

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/token/PresaleToken.sol#L57 |
| **Status** | Unresolved |

## Description

The `mint` function emits custom `Minted` event, in addition to the standard `Transfer` event emitted by the ERC20 `_mint` function from OpenZeppelin's implementation. This leads to redundant event emissions for the same actions, which can cause unnecessary clutter in the event logs.

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

        _mint(to, amount);

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

## Recommendation

It is recommended to remove the redundant `Minted` event emission from the `mint` function. The standard `Transfer` events emitted by the `_mint` function are sufficient to indicate the minting of tokens. This change will simplify the event logs associated with emitting multiple events for the same actions.

## 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.

## L04 - Conformance to Solidity Naming Conventions

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Quoter.sol#L17<br>contracts/Presale.sol#L204,207 |
| **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.

```
IUniswapV2Router02 public immutable UNISWAP_V2_ROUTER
address public immutable MULTISIG_WALLET
uint256 public immutable TOKENS_AVAILABLE_FOR_PRESALE
```

### 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/v0.8.17/style-guide.html#naming-convention.

## L11 - Unnecessary Boolean equality

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/Presale.sol#L273,322,356,367,415,417,576,579,765 |
| **Status** | Unresolved |

### Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
salePaymentTokens[token].allowed != true
saleToken.allowed == false
saleToken.peggedToUsd != true
paymentToken.allowed == false
paymentToken.peggedToUsd == true
salePaymentTokens[token].allowed == true
peggedToUsd == true
salePaymentTokens[tokenSell].peggedToUsd == true
```

### Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

## L16 - Validate Variable Setters

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/token/PresaleToken.sol#L29<br>contracts/Presale.sol#L240,526 |
| **Status** | Unresolved |

### Description

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

```
operator = _operator
MULTISIG_WALLET = _multisigWallet
(bool succeed,) = payable(sender).call{value: transferValue -
payAmount}("")
```

### Recommendation

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

## L19 - Stable Compiler Version

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | contracts/token/PresaleToken.sol#L2<br>contracts/token/IPresaleToken.sol#L2<br>contracts/Quoter.sol#L2<br>contracts/Presale.sol#L2 |
| **Status** | Unresolved |

### Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.25;
```

### Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

# Functions Analysis

| Contract | Type | Bases | | |
|---|---|---|---|---|
| | **Function Name** | **Visibility** | **Mutability** | **Modifiers** |
| | | | | |
| **IUniswapV2Router02** | Interface | | | |
| | getAmountsOut | External | | - |
| | | | | |
| **Quoter** | Implementation | | | |
| | | Public | ✓ | - |
| | getQuoteUniswapV2 | Internal | | |
| | | | | |
| **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<br>checkIfOptionInitialized<br>checkIfTokenAllowed<br>checkTokensRemain<br>nonReentrant |
| | pause | External | ✓ | onlyOwner |
| | unpause | External | ✓ | onlyOwner |
| | addPaymentToken | External | ✓ | onlyOwner |
| | removePaymentToken | External | ✓ | onlyOwner<br>checkIfTokenAllowed |
| | createNewOption | External | ✓ | onlyOwner |
| | deleteOption | External | ✓ | onlyOwner<br>checkIfOptionInitialized |
| | withdraw | External | ✓ | onlyOwner |
| | initOptions | External | ✓ | onlyOwner |
| | _calculatePayAmount | Private | | |
| | _deployPresaleTokenForOption | Private | ✓ | |
| | _buyExactTokens | Private | ✓ | |
| | _getTokenOutDecimals | Private | | |
| | _getDecimals | Private | | |
| | _compressToDecimals | Private | | |
| | _expandToDecimals | Private | | |
| | _getDecimalsDelta | Private | | |
| | | | | |
| **PresaleToken** | Implementation | ERC20,<br>Ownable, | | |

| | | | | |
|---|---|---|---|---|
| | | IPresaleToken | | |
| | | Public | ✓ | Ownable |
| | isOperator | External | | - |
| | transferOperator | External | ✓ | onlyOwner |
| | mint | External | ✓ | - |
| | | | | |
| **IPresaleToken** | Interface | | | |
| | operator | External | | - |
| | isOperator | External | | - |
| | transferOperator | External | ✓ | - |
| | mint | External | ✓ | - |
| | | | | |
| **ReentrancyGuard** | Implementation | | | |
| | | Public | ✓ | - |
| | _nonReentrantBefore | Private | ✓ | |
| | _nonReentrantAfter | Private | ✓ | |
| | _reentrancyGuardEntered | Internal | | |
| | | | | |
| **Pausable** | Implementation | Context | | |
| | | Public | ✓ | - |
| | paused | Public | | - |
| | _requireNotPaused | Internal | | |
| | _requirePaused | Internal | | |

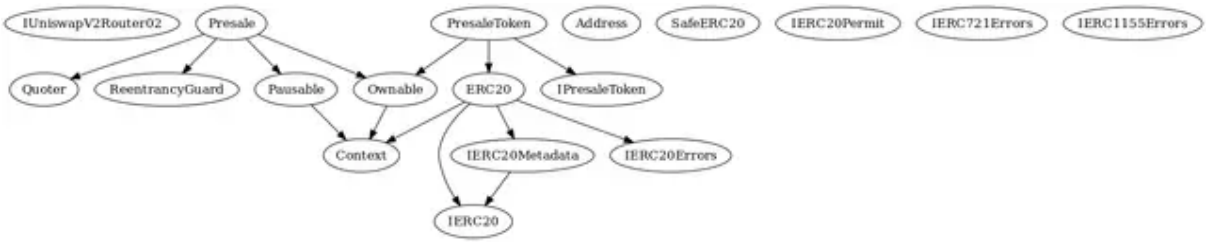
| | | | | |
|---|---|---|---|---|
| | _pause | Internal | ✓ | whenNotPaused |
| | _unpause | Internal | ✓ | whenPaused |
| | | | | |
| **Context** | Implementation | | | |
| | _msgSender | Internal | | |
| | _msgData | Internal | | |
| | _contextSuffixLength | Internal | | |
| | | | | |
| **Address** | Library | | | |
| | sendValue | Internal | ✓ | |
| | functionCall | Internal | ✓ | |
| | functionCallWithValue | Internal | ✓ | |
| | functionStaticCall | Internal | | |
| | functionDelegateCall | Internal | ✓ | |
| | verifyCallResultFromTarget | Internal | | |
| | verifyCallResult | Internal | | |
| | _revert | Private | | |
| | | | | |
| **IERC20** | Interface | | | |
| | totalSupply | External | | - |
| | balanceOf | External | | - |
| | transfer | External | ✓ | - |
| | allowance | External | | - |

| | | | | |
|---|---|---|---|---|
| | approve | External | ✓ | - |
| | transferFrom | External | ✓ | - |
| | | | | |
| **ERC20** | Implementation | Context, IERC20, IERC20Metadata, IERC20Errors | | |
| | | Public | ✓ | - |
| | name | Public | | - |
| | symbol | Public | | - |
| | decimals | Public | | - |
| | totalSupply | Public | | - |
| | balanceOf | Public | | - |
| | transfer | Public | ✓ | - |
| | allowance | Public | | - |
| | approve | Public | ✓ | - |
| | transferFrom | Public | ✓ | - |
| | _transfer | Internal | ✓ | |
| | _update | Internal | ✓ | |
| | _mint | Internal | ✓ | |
| | _burn | Internal | ✓ | |
| | _approve | Internal | ✓ | |
| | _approve | Internal | ✓ | |
| | _spendAllowance | Internal | ✓ | |
| | | | | |

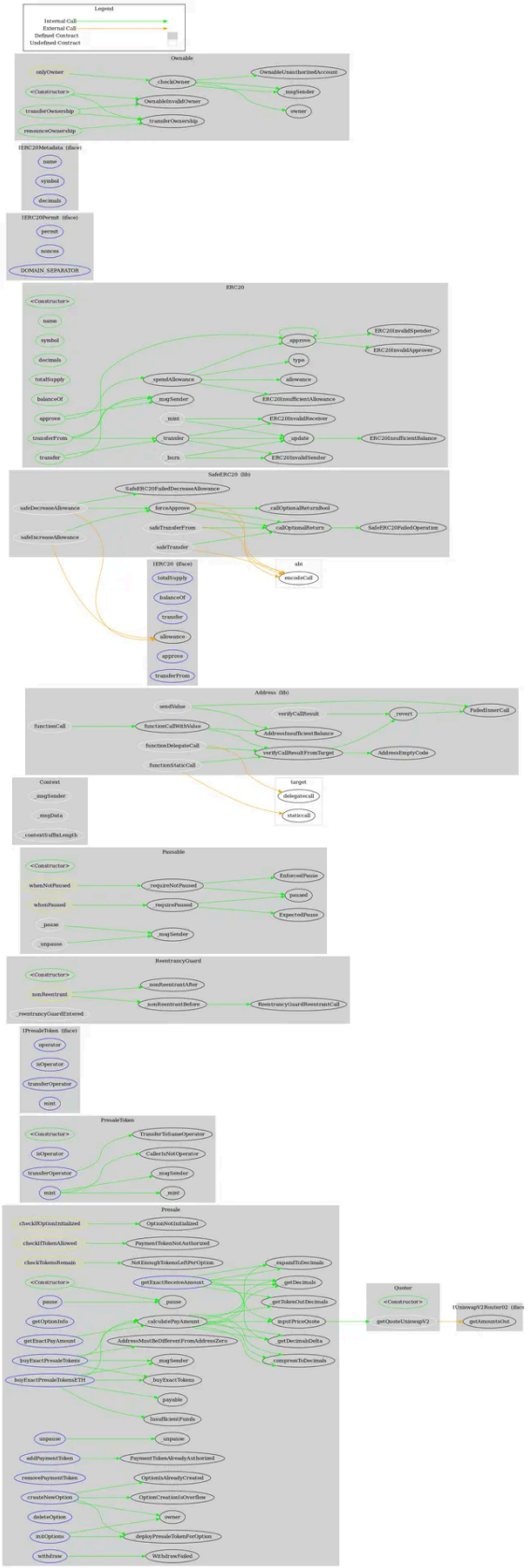
| | | | | |
|---|---|---|---|---|
| **SafeERC20** | Library | | | |
| | safeTransfer | Internal | ✓ | |
| | safeTransferFrom | Internal | ✓ | |
| | safeIncreaseAllowance | Internal | ✓ | |
| | safeDecreaseAllowance | Internal | ✓ | |
| | forceApprove | Internal | ✓ | |
| | _callOptionalReturn | Private | ✓ | |
| | _callOptionalReturnBool | Private | ✓ | |
| | | | | |
| **IERC20Permit** | Interface | | | |
| | permit | External | ✓ | - |
| | nonces | External | | - |
| | DOMAIN_SEPARATOR | External | | - |
| | | | | |
| **IERC20Metadata** | Interface | IERC20 | | |
| | name | External | | - |
| | symbol | External | | - |
| | decimals | External | | - |
| | | | | |
| **IERC20Errors** | Interface | | | |
| | | | | |
| **IERC721Errors** | Interface | | | |
| | | | | |

| | | | | |
|---|---|---|---|---|
| **IERC1155Errors** | Interface | | | |
| | | | | |
| **Ownable** | Implementation | Context | | |
| | | Public | ✓ | - |
| | owner | Public | | - |
| | _checkOwner | Internal | | |
| | renounceOwnership | Public | ✓ | onlyOwner |
| | transferOwnership | Public | ✓ | onlyOwner |
| | _transferOwnership | Internal | ✓ | |

# Inheritance Graph



# Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Ownable
onlyOwner
checkOwner
OwnableUnauthorizedAccount
<Constructor>
OwnableInvalidOwner
msgSender
transferOwnership
transferOwnership
owner
renounceOwnership
IERC20Metadata (iface)
name
symbol
decimals
IERC20Permit (iface)
permit
nonces
DOMAIN_SEPARATOR
ERC20
<Constructor>
name
symbol
decimals
totalSupply
balanceOf
approve
transferFrom
transfer
approve
ERC20InvalidSpender
ERC20InvalidApprover
type
spendAllowance
allowance
msgSender
ERC20InsufficientAllowance
mint
ERC20InvalidReceiver
transfer
update
ERC20InsufficientBalance
burn
ERC20InvalidSender
SafeERC20 (lib)
SafeERC20FailedDecreaseAllowance
safeDecreaseAllowance
forceApprove
callOptionalReturnBool
safeTransferFrom
callOptionalReturn
SafeERC20FailedOperation
safeIncreaseAllowance
safeTransfer
abi
encodeCall
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
Address (lib)
sendValue
verifyCallResult
revert
FailedInnerCall
functionCall
functionCallWithValue
AddressInsufficientBalance
functionDelegateCall
verifyCallResultFromTarget
AddressEmptyCode
functionStaticCall
target
delegatecall
staticcall
Context
_msgSender
_msgData
_contextSuffixLength
Pausable
<Constructor>
whenNotPaused
_requireNotPaused
EnforcedPause
paused
whenPaused
_requirePaused
ExpectedPause
_pause
_unpause
msgSender
ReentrancyGuard
<Constructor>
nonReentrant
_nonReentrantAfter
_nonReentrantBefore
ReentrancyGuardReentrantCall
_reentrancyGuardEntered
IPresaleToken (iface)
operator
isOperator
transferOperator
mint
PresaleToken
<Constructor>
TransferToSameOperator
isOperator
CallerIsNotOperator
transferOperator
msgSender
mint
mint
Presale
checkIfOptionInitialized
OptionNotInitialized
checkIfTokensAllowed
PaymentTokensNotAuthorized
checkTokensRemain
NotEnoughTokensLeftPerOption
expandToDecimals
<Constructor>
getExactReceiveAmount
getDecimals
pause
pause
getTokensOutDecimals
getOptionInfo
calculatePayAmount
inputPriceQuote
getExactPayAmount
AddressMustBeDifferentFromAddressZero
getDecimalsDelta
buyExactPresaleTokens
msgSender
compressToDecimals
buyExactPresaleTokensETH
buyExactTokens
payable
InsufficientFunds
unpause
unpause
addPaymentToken
PaymentTokenAlreadyAuthorized
removePaymentToken
OptionIsAlreadyCreated
createNewOption
OptionCreationIsOverflow
deleteOption
owner
initOptions
deployPresaleTokenForOption
withdraw
WithdrawFailed
Quoter
<Constructor>
getQuoteUniswapV2
IUniswapV2Router02 (iface)
getAmountsOut

# Summary

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

# 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