Cyberscope

Audit Report

AIGPROJECT

October 2023

Repository	https://github.com/aigtkn/presale_widget_abhishek_01/blob/main/AiGold_Presale.txt
Commit	7e0feb2d7103abda968bde453228440c00b565d90fc8ffb430791582b3f8d58e
Audited by	© cyberscope

Table of Contents

Review

Repository	https://github.com/aigtkn/presale_widget_abhishek_01/blob/main/AiGold_Presale.txt
Commit	2992e09b2d34ff3419f70b0f8eb1ad3b10786cb0
Testing Deploy	https://goerli.etherscan.io/address/0xa961f4bc31dd228914a307023e10afbeac033a09

Audit Updates

Initial Audit	09 Aug 2023 https://github.com/cyberscope-io/audits/blob/main/aig/v1/presale.pdf
Corrected Phase 2	19 Aug 2023 https://github.com/cyberscope-io/audits/blob/main/aig/v2/presale.pdf
Corrected Phase 3	25 Oct 2023

Source Files

Filename	SHA256
TestPresale.sol	229f4a51b6c3bdd7bdc7acb63b53a3592574afc9f2ad1595442297b558bd958a

Overview

The "TestPresale" contract is designed to manage a token presale on the Ethereum blockchain. The contract is structured around a series of rounds, with a total of 12 rounds, each having a distinct token price and duration. Each round is designed to last for 12 days, with a predefined starting and ending price. The contract leverages the Chainlink price feed to obtain real-time price data and accepts both ETH and USDT as payment methods for token purchases. The total number of tokens available for sale is determined by the sum of tokens allocated for each round. The contract also incorporates safety checks to ensure that purchases are made within the valid timeframes of the ongoing round and that the presale adheres to the set parameters. Furthermore, the contract owner retains the capability to pause the presale and adjust critical sale parameters.

Out of Scope Address: The contract uses the `priceFeed` address to fetch real-time ETH prices. Specifically, the `latestRoundData` function utilizes this priceFeed to retrieve and compute the accurate ETH price. It's important to note that the integrity, functionality, and security associated with the priceFeed address remain out of the scope of this contract audit. Engaging with this address or depending on its data mandates prudence. A distinct audit or review is advised for the contract associated with that specific address.

Price Calculation

The presale allows users to purchase tokens using both ETH and USDT. When a user opts to buy with ETH, the `buyWithETH` function is invoked, which internally calls the `ethUSDHelper` function. This helper function leverages the Chainlink oracle to determine the current ETH to USD price, ensuring accurate conversion. It then calculates the equivalent USD amount for the provided ETH, taking into account the decimal differences between ETH and USDT, and returns the computed value. This ensures that users get the correct number of tokens for their ETH based on the current market rate.

Token Purchasing

The contract offers two primary methods for users to acquire tokens: through ETH (`buyWithETH`) and stablecoins like USDT (`buyWithUSDT`). Both purchasing functions are safeguarded by the validRoundCheck modifier, ensuring that:

- The presale isn't paused.
- The purchase occurs within a valid round, specifically between rounds 1 to 12.
- Tokens are still available for purchase, especially ensuring that if all tokens are sold out by the 12th round, no further purchases can be made.
- The purchase is made within the stipulated start and end times of the current round.

In the event any of these conditions aren't met, the transaction is reverted with an appropriate error message. The `buyWithETH` function further converts the provided ETH amount to its equivalent in USD, leveraging the `ethUSDHelper` function. All token purchases, regardless of the method, are internally handled by the `_buyTokens` function. Transactions are also protected against reentrancy attacks, ensuring the security of funds and the integrity of the sale process.

Token Purchase Mechanism

The internal `_buyTokens` function in the contract is intricately designed to manage the process of purchasing tokens, whether the user opts to buy with ETH or USD tokens. The function requires a minimum investment of 10 USD. Upon invoking the function, the current round and the user's details are fetched. The function then calculates the number of tokens the user can acquire based on the inputted USD amount and the prevailing rate for the current round. If the available tokens in the current round are insufficient to fulfill the user's purchase request, the function automatically transitions to the next round, adjusting the token price accordingly. This ensures that if a user's investment can buy tokens across different rounds, it does so efficiently. For instance, if only 100 tokens are left in the first round priced at 0.01 USD each, and a user wishes to invest 100 USD, they would buy the remaining tokens of the first round and the rest at the rate of the second round. A special consideration is given to the final round, the 12th round. If a user's purchase exhausts the tokens of this round, the function calculates the exact USD required and determines if there's any excess amount. This excess, in the case of ETH purchases, is refunded to the user. The function also incorporates checks to ensure that a user doesn't exceed the maximum wallet limit. Once the calculations are done, the function transfers the USD or ETH to the multiSig wallet. If there's any refund due, ETH buys, it's returned to the user. The function concludes by updating the user's total contributions and the total tokens they've bought, along with updating the overall metrics of tokens sold and USD raised. Every token purchase is also broadcasted as an event for transparency.

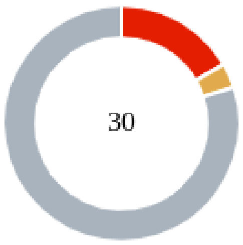
Start Claim

The `enableClaims` function is designed for the contract owner to initiate the claim process for tokens.

Claim

The `claim` function is designed to facilitate users in retrieving the tokens they have purchased during the presale. The claim functionality must be enabled, which is determined by the `claimEnabled` variable. If it's not set to 2, the function will revert, indicating that claims are not yet available. Once this primary condition is met, the function fetches the user's details based on the caller's address. It then checks if the user has previously claimed their tokens. If they have, the function reverts with a message indicating that the user has already claimed their tokens, ensuring that double-claims are prevented. If the user hasn't claimed before, the function calculates the total tokens the user is entitled to, which is equivalent to the total tokens they've bought. This amount is then marked as claimed to prevent future claims. The function then proceeds to transfer the tokens to the user. To ensure the transfer's success, the function checks the user's token balance before and after the transfer. If there's no increase in the balance post-transfer, the function reverts, signaling a failed claim. Every successful claim is broadcasted as an event, providing transparency and traceability for each claim transaction.

Findings Breakdown



●	Critical	5
●	Medium	1
●	Minor / Informative	24

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	5	0	0	0
●	Medium	1	0	0	0
●	Minor / Informative	24	0	0	0

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
🔴	CMCC	Contradictory Modifier Condition Checks	Unresolved
🔴	RVCB	Round Validation Check Bypass	Unresolved
🔴	IRV	Inaccurate Round Validation	Unresolved
🔴	IRC	Incorrect Refund Calculation	Unresolved
🔴	FTC	Flawed Token Calculation	Unresolved
🟠	IRM	Inconsistent Refund Mechanism	Unresolved
⚪	PRDI	Potential Reward Decimal Inconsistency	Unresolved
⚪	OCTD	Transfers Contract's Tokens	Unresolved
⚪	UVD	Unoptimized Variables Declaration	Unresolved
⚪	FO	Function Optimization	Unresolved
⚪	CCR	Contract Centralization Risk	Unresolved
⚪	IBC	Insufficient Balance Check	Unresolved
⚪	UST	Uninitialized Sale Token	Unresolved
⚪	RCC	Redundant Condition Checks	Unresolved

●	IBR	Inefficient Boolean Representation	Unresolved
●	RC	Repetitive Calculations	Unresolved
●	ODM	Oracle Decimal Mismatch	Unresolved
●	RSW	Redundant Storage Writes	Unresolved
●	ZD	Zero Division	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	DPI	Decimals Precision Inconsistency	Unresolved
●	RSK	Redundant Storage Keyword	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L18	Multiple Pragma Directives	Unresolved
●	L19	Stable Compiler Version	Unresolved

CMCC - Contradictory Modifier Condition Checks

Criticality	Critical
Location	TestPresale.sol#L974
Status	Unresolved

Description

The contract is designed with a `validRoundCheck` modifier to ensure certain conditions are met before proceeding with specific functions. However:

1. The condition `currentRound < 1 && currentRound > 12` checks if `currentRound` is simultaneously less than 1 and greater than 12, which is impossible. As a result, since the condition is contradictory and the `InvalidRound` error will never be triggered.
2. Similarly, the condition `block.timestamp < round.startTime && block.timestamp > round.endTime` checks if the current block timestamp is both before the start time and after the end time simultaneously. This is also impossible, meaning the `PresaleIsNotStartedOrAlreadyEnded` error will never be triggered.

As a result, this condition consists of logical inconsistencies in the conditions checked within this modifier.

```
    modifier validRoundCheck() {
        if (paused == 2) {
            revert PresaleIsPaused();
        }
        if (currentRound < 1 && currentRound > 12) {
            revert InvalidRound();
        }
        Round storage round = rounds[currentRound];
        uint256 availableTokens = round.totalTokens - round.totalSold;
        if (availableTokens == 0 && currentRound == 12) {
            revert PresaleIsOver();
        }
        if (
            block.timestamp < round.startTime && block.timestamp >
round.endTime
        ) {
            revert PresaleIsNotStartedOrAlreadyEnded();
        }

        _;
    }
```

Recommendation

It is recommended that the team revisits the code of the `validRoundCheck` modifier. Specifically, the `&&` operators in the conditions `currentRound < 1 && currentRound > 12` and `block.timestamp < round.startTime && block.timestamp > round.endTime` should be replaced with the `||` operators. This change will ensure that the conditions reflect the intended logic and that the appropriate reverts are triggered when necessary.

RVCB - Round Validation Check Bypass

Criticality	Critical
Location	TestPresale.sol#L897
Status	Unresolved

Description

The contract contains the `switchRound` function which aims to validate the round number. However, the function uses an if statement that checks if the parameter `_round` is both less than or equal to `currentRound` and greater than 12. Given the nature of logical operators, this condition will always evaluate to false, making the function never revert with the "InvalidRound" error message, even when it should. For example in case where `currentRound` equals 12 and `switchToNextRound` is invoked, then `_round` will equal 13, and `currentRound` will equal 12 within the `switchRound` function. As a result, the if statement will evaluate to false (since `13 <= 12`) and therefore the revert error will not be displayed as intended.

```
   function switchRound(uint256 _round) internal {
        if (_round <= currentRound && _round > 12) {
            revert InvalidRound();
        }
    ...
    }

    function switchToNextRound() external onlyOwner {
        currentRound = currentRound + 1;
        switchRound(currentRound);
    }
```

Recommendation

It is recommended to rectify the validation logic within the `switchRound` function. The contract should be adjusted to properly validate `_round` values, ensuring that the values are within the intended range of 1 to 12. This will ensure that the function operates correctly and remains consistent with the contract's design.

IRV - Inaccurate Round Validation

Criticality	Critical
Location	TestPresale.sol#L1253
Status	Unresolved

Description

The contract contains the `addTokensToSale` function which allows the owner to add tokens to a sale. However, the function reverts if `currentRound` is less than `13`. Given that the only valid values for `currentRound` within the contract's implementation range from 1 to 12, this means the function will always revert, making it unusable and defeating its purpose.

```
    function addTokensToSale(uint256 amount) external onlyOwner {
        if (currentRound < 13) {
            revert InvalidRound();
        }
        Round storage round = rounds[currentRound];
        round.totalTokens = round.totalTokens + amount;
        totalTokensForSale = totalTokensForSale + amount;
    }
```

Recommendation

It is recommended to adjust the validation logic within the `addTokensToSale` function. Specifically, the function should only revert if `currentRound` is greater than `12`. This ensures that the function operates correctly within the intended rounds and remains consistent with the contract's design.

IRC - Incorrect Refund Calculation

Criticality	Critical
Location	TestPresale.sol#L1114
Status	Unresolved

Description

The contract contains a refund mechanism to return the extra USDT amount that users overpaid in the last round. To track this overpay, the contract uses the `extraUsd` variable. However, the contract calculates `extraUsd` as `_usdAmount - extraUsd` . Given that `extraUsd` is initialized to zero, this calculation will always result in `extraUsd` being equal to `_usdAmount` . This means that the contract will always consider the entire `_usdAmount` as surplus, which does not reflect a refund functionality.

```
   uint256 usdRequired;
        uint256 extraUsd;
        if (currentRound == 12 && outputTokens == availableTokens) {
            usdRequired =
                (availableTokens * round.currentPrice) /
                10 ** TOKEN_DECIMALS;
            if (usdRequired > 0) {
                extraUsd = _usdAmount - extraUsd;
                _usdAmount = usdRequired;
                outputTokens = availableTokens;
            }
        }
```

Recommendation

It is recommended to rectify the logic of the refund mechanism. If the intended functionality is to return the surplus USDT paid by the user, the contract should compute the surplus as `extraUsd = _usdAmount - usdRequired` . This will ensure that only the actual surplus amount is considered as extraUsd, making the refund mechanism accurate and fair.

FTC - Flawed Token Calculation

Criticality	Critical
Location	TestPresale.sol#L1299
Status	Unresolved

Description

The contract is utilizing the `getTokenAmount` function to determine the number of tokens that can be acquired with a specified amount of USDT. However, when `currentRound` is equal to 12, regardless of the USDT amount provided, the function returns the remaining tokens of the 12th round. This means that an individual could potentially acquire all the unsold tokens of the 12th round with a minimal USDT amount, which could lead to unintended consequences and potential exploitation.

```
    function getTokenAmount(
        uint256 _usdAmount
    ) public view returns (uint256 tokensOutput) {
        ...
        if (totalAvailableTokens >= tokensAtCurrentPrice) {
            return tokensAtCurrentPrice;
        } else if (
            totalAvailableTokens < tokensAtCurrentPrice && currentRound
< 12
        ) {
            uint256 usdUsed = (totalAvailableTokens *
round.currentPrice) /
                10 ** TOKEN_DECIMALS;
            uint256 usdLeft = _usdAmount - usdUsed;
            uint256 tokensFromNewRound = (usdLeft * 10 **
TOKEN_DECIMALS) /
                round.nextPrice;
            return tokensFromNewRound + totalAvailableTokens;
        } else {
            return round.totalTokens - round.totalSold;
        }
    }
```

Recommendation

It is recommended to re-evaluate the logic within the `getTokenAmount` function, particularly when currentRound equals `12`. The contract should accurately compute the number of tokens based on the provided USDT amount for the last round, ensuring fairness and preventing potential misuse.

IRM - Inconsistent Refund Mechanism

Criticality	Medium
Location	TestPresale.sol#L1107
Status	Unresolved

Description

The contract contains a refund functionality that calculates the `extraUsd` amount, which represents the surplus funds that can be returned to the user. However, the contract only processes the refund for the `msg.sender` if they have purchased tokens using native tokens like ETH. Consequently, users who acquire tokens with USDT will not benefit from the refund mechanism, leading to potential loss of funds and user dissatisfaction.

```
       uint256 usdRequired;
        uint256 extraUsd;
        if (currentRound == 12 && outputTokens == availableTokens) {
            usdRequired =
                (availableTokens * round.currentPrice) /
                10 ** TOKEN_DECIMALS;
            if (usdRequired > 0) {
                extraUsd = _usdAmount - extraUsd;
                _usdAmount = usdRequired;
                outputTokens = availableTokens;
            }
        }

        if (user.totalTokensBought + outputTokens > MAX_WALLET_CAP) {
            revert MaxWalletCapReached();
        }

        if (value) {
            uint256 multiSigBalanceBefore = USDT.balanceOf(multiSig);
            USDT.safeTransferFrom(msg.sender, multiSig, _usdAmount);
            uint256 multiSigBalanceAfter = USDT.balanceOf(multiSig);

            if (multiSigBalanceAfter <= multiSigBalanceBefore) {
                revert USDPaymentFailed();
            }
        }
        if (!value) {
            uint256 refund = 0;
            uint256 ethRequired = msg.value;
            if (extraUsd > 0) {
                uint256 oneEth = (getLatestPrice() * 1e10);
                ethRequired = (_usdAmount * 1e12) / oneEth;
                refund = msg.value - ethRequired;
            }

            (bool success, ) = payable(multiSig).call{value:
ethRequired}("");
            if (!success) {
                revert ETHTransferFailed();
            }
            if (refund > 0) {
                (bool sent, ) = payable(msg.sender).call{value:
refund}("");
                if (!sent) {
                    revert ETHRefundFailed();
                }
            }
```

Recommendation

It is recommended to re-evaluate the code implementation. If the intended functionality is to offer a refund every time a user overpays, then the contract should consistently refund users regardless of whether they made purchases with native tokens or USDT. This ensures fairness and prevents potential grievances from users who transact with USDT.

PRDI - Potential Reward Decimal Inconsistency

Criticality	Minor / Informative
Location	TestPresale.sol#L1050
Status	Unresolved

Description

The contract contains the `claim` function that allows users to retrieve their `token` rewards. However, an inherent assumption in the function is that the `token` has the same decimal precision as the tokens in `availableToClaim` variable. This assumption can lead to discrepancies in the amount of `token` transferred to users, if the `token` and the tokens in `availableToClaim` have different decimal configurations. Such an oversight can result in users either receiving more or fewer tokens than they should, potentially leading to financial discrepancies and undermining the trustworthiness of the contract.

```
    function claim() external nonReentrant {
        if (claimEnabled != 2) {
            revert ClaimsAreNotAvailableYet();
        }
        User storage user = users[msg.sender];
        if (user.totalTokensClaimed > 0) {
            revert AlreadyClaimed();
        }
        uint256 availableToClaim = user.totalTokensBought;
        user.totalTokensClaimed = availableToClaim;
        uint256 balanceBefore = token.balanceOf(msg.sender);
        token.safeTransfer(msg.sender, availableToClaim);
        uint256 balanceAfter = token.balanceOf(msg.sender);
        if (balanceAfter <= balanceBefore) {
            revert TokenClaimFailed();
        }
        emit TokensClaimed(msg.sender, availableToClaim);
    }
```

Recommendation

It is recommended to explicitly handle the decimal differences between the `token` and the tokens in `availableToClaim` variable. One approach is to fetch the decimal values of both tokens using the ERC20 decimals() function and then adjust the claimable amount accordingly. This ensures that users receive the correct amount of `token` proportional to their deposits, regardless of the decimal differences between the tokens.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	TestPresale.sol#L433,450
Status	Unresolved

Description

The contract owner has the authority to claim all the balance of the contract. Specifically the contract is designed to allow the owner to set a new token address using the `setToken` function and subsequently claim any ERC20 tokens (other than the currently set token) using the `claimOtherERC20` function. The contract owner can strategically invoke the `setToken` function to change the token address and then invoke the `claimOtherERC20` function to claim all the tokens of the previous address. This means that the owner can potentially drain all the tokens from the contract, which poses a significant security risk.

```
    function setToken(address _token) external onlyOwner {
        if (_token == address(0)) {
            revert ZeroAddressNotAllowed();
        }
        token = IERC20(_token);
    }

    function claimOtherERC20(
        address _oToken,
        uint256 amount
    ) external onlyOwner {
        if (_oToken == address(token)) {
            revert CannotClaimNativeTokens();
        }
        IERC20 otherToken = IERC20(_oToken);
        otherToken.safeTransfer(owner(), amount);
    }
```

Recommendation

It is recommended to implement additional safeguards to prevent the owner from claiming the tokens. The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

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

Permanent Solution:

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

UVD - Unoptimized Variables Declaration

Criticality	Minor / Informative
Location	TestPresale.sol#L1107
Status	Unresolved

Description

The contract is designed with certain variables that are declared but not always utilized. Specifically, the variables `usdRequired` and `extraUsd` are declared inside the `_buyTokens` function. However, these variables are actually used only within the `if` statement that checks `if (currentRound == 12 && outputTokens == availableTokens)`. If this condition is not met, these variables remain unutilized, leading to unnecessary gas consumption.

```
uint256 usdRequired;
uint256 extraUsd;
    if (currentRound == 12 && outputTokens == availableTokens) {
        usdRequired =
            (availableTokens * round.currentPrice) /
            10 ** TOKEN_DECIMALS;
        if (usdRequired > 0) {
            extraUsd = _usdAmount - extraUsd;
            _usdAmount = usdRequired;
            outputTokens = availableTokens;
        }
```

Recommendation

It is recommended to move the declaration of the `usdRequired` and `extraUsd` variables inside the if segment where they are used. This ensures that they are only declared when they are needed, optimizing gas usage and improving code clarity.

FO - Function Optimization

Criticality	Minor / Informative
Location	TestPresale.sol#L897
Status	Unresolved

Description

The contract is attempting to set parameters for 12 different rounds. However, a significant portion of the parameters for each round are identical to the previous rounds. Furthermore, the `switchRound` function uses 12 separate if conditions. This means that even if one if condition is satisfied, all subsequent if conditions will still be checked, leading to increased gas consumption.

```
function switchRound(uint256 _round) internal {
        if (_round <= currentRound && _round > 12) {
            revert InvalidRound();
        }
        if (_round == 1) {
            Round storage round = rounds[_round];
            round.currentPrice = 3500;
            round.nextPrice = 4500;
            round.totalTokens = tokensPerRound;
            round.totalSold = 0;
            round.startTime = block.timestamp;
            round.endTime = block.timestamp + 12 days;
            currentRound = _round;
        }

        if (_round == 2) {
            Round storage round = rounds[_round];
            round.currentPrice = 4500;
            round.nextPrice = 5500;
            round.totalTokens = tokensPerRound;
            round.totalSold = 0;
            round.startTime = block.timestamp;
            round.endTime = block.timestamp + 12 days;
            currentRound = _round;
        }
...
}
```

Recommendation

It is recommended to optimize the `switchRound` function. Instead of using multiple independent `if` conditions, an `if-else if` structure should be employed. This ensures that once a condition is met, the subsequent conditions are not unnecessarily checked, thus saving gas. Additionally, given the repetitive nature of the round parameters, a more modular approach or a loop-based mechanism might be considered to set round parameters, further enhancing efficiency and readability.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	TestPresale.sol#L1169,1173,1187
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.

The owner possesses significant authority over critical functionalities. Specifically, the owner has the authority to set and transfer the sale token to the contract, set the presale price to any desired value, modify the end date for the current round, and enable the claim functionality. Such a concentration of power in the hands of a single entity or individual poses a centralization risk.

```
   function enableClaims() external onlyOwner {
        if (claimEnabled == 2) {
            revert ClaimsAreEnabledAlready();
        }
        claimEnabled = 2;
    }

    function setEndDateForCurrentRound(uint256 endDate) external
onlyOwner {
        Round storage round = rounds[currentRound];
        if (endDate < block.timestamp) {
            revert CannotSetDateInPast();
        }
        round.endTime = endDate;
    }

    function setPrice(
        uint256 _newPrice,
        uint256 _nextPrice
    ) external onlyOwner {
        Round storage round = rounds[currentRound];
        round.currentPrice = _newPrice;
        round.nextPrice = _nextPrice;
    }
```

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.

IBC - Insufficient Balance Check

Criticality	Minor / Informative
Location	TestPresale.sol#L1253,1262
Status	Unresolved

Description

The contract contrains the `addTokensToSale` and `removeTokensFromSale` functions, which allows the owner to modify the value of `totalTokens` . However, these functions currently lacks essential checks to verify the feasibility of the change. If the value of `totalTokens` is increased, the contract does not ascertain if it possesses an adequate balance to cover this enhancement. The consequence is that it could result in a scenario where the contract promises more tokens for presale than it physically possesses, potentially leading to failures or inconsistencies when users attempt to purchase these tokens.

```
   function addTokensToSale(uint256 amount) external onlyOwner {
        if (currentRound < 13) {
            revert InvalidRound();
        }
        Round storage round = rounds[currentRound];
        round.totalTokens = round.totalTokens + amount;
        totalTokensForSale = totalTokensForSale + amount;
    }

    /// @dev remove tokens to current sale round
    /// @param amount: token amount to remove
    function removeTokensFromSale(uint256 amount) external onlyOwner {
        Round storage round = rounds[currentRound];
        if (amount > round.totalTokens - round.totalSold) {
            revert AmountMustBeLessThanAvailableTokens();
        }
        round.totalTokens = round.totalTokens - amount;
        totalTokensForSale = totalTokensForSale - amount;
    }
```

Recommendation

It is recommended to incorporate a verification mechanism within the `addTokensToSale` and `removeTokensFromSale` functions to ascertain the contract's token balance before allowing any adjustments to the `totalTokens` value. Specifically, when there's an increment in `totalTokens`, the contract should check if it has enough tokens to cover the new total. This can be achieved by querying the token balance of the contract and comparing it against the proposed `totalTokens` value. Only if the balance suffices should the change be permitted; otherwise, the operation should be rejected.

UST - Uninitialized Sale Token

Criticality	Minor / Informative
Location	TestPresale.sol#L1164,1198
Status	Unresolved

Description

The contract doesn't initialize the sale token during the setup of the presale state through the `enableClaims`. As a result the contract cannot ensure that it will have a sufficient balance of the `token` when users start claim. While the `setToken` function allows the owner to set the `token`, it can be done after the presale phase has ended. This means that there is no inherent guarantee in the contract structure itself to ensure the sale token's presence, leading to potential disruptions in the token sale process.

```
   function enableClaims() external onlyOwner {
        if (claimEnabled == 2) {
            revert ClaimsAreEnabledAlready();
        }
        claimEnabled = 2;
    }

     function setToken(address _token) external onlyOwner {
        if (_token == address(0)) {
            revert ZeroAddressNotAllowed();
        }
        token = IERC20(_token);
    }
```

Recommendation

It is recommended to modify the contract structure to transfer the `token` to the contract during its initialization phase, ensuring that the necessary tokens are available right from the start of the presale. This approach will reduce the risk of insufficiency and ensure a smoother user experience by preventing interruptions in the token purchasing process. It also reduces the reliance on external actors (e.g., the owner) to take subsequent actions for the contract to function as intended.

RCC - Redundant Condition Checks

Criticality	Minor / Informative
Location	TestPresale.sol#L1090,1124
Status	Unresolved

Description

The contract contains multiple if statements that check for conditions which are direct opposites of conditions already verified. Specifically, the code checks if `availableTokens` is greater than or equal to `outputTokens` and then immediately checks if `availableTokens` is less than `outputTokens`. Similarly, it checks for the `value` and then immediately checks for the negation of the same `value` variable. These redundant checks not only make the code less readable but also increase gas consumption, leading to inefficiencies.

```
      if (availableTokens >= outputTokens) {
            round.totalSold = round.totalSold + outputTokens;
        }

        if (availableTokens < outputTokens) {
        ...
        }
    ...
        if (value) {
            uint256 multiSigBalanceBefore = USDT.balanceOf(multiSig);
            USDT.safeTransferFrom(msg.sender, multiSig, _usdAmount);
            uint256 multiSigBalanceAfter = USDT.balanceOf(multiSig);

            if (multiSigBalanceAfter <= multiSigBalanceBefore) {
                revert USDPaymentFailed();
            }
        }
        if (!value) {
        ...
        }
```

Recommendation

It is recommended to streamline the code by using if-else constructs where contradictory conditions are being checked. This will enhance code readability, reduce potential points of failure, and optimize gas usage.

IBR - Inefficient Boolean Representation

Criticality	Minor / Informative
Location	TestPresale.sol#L846
Status	Unresolved

Description

The contract is utilizing the `claimEnabled` and `paused` variables as `uint256` data types to represent boolean states. Specifically, the value 1 is used to represent false and the value 2 is used to represent true. This approach is not only counterintuitive but also inefficient in terms of gas consumption and can lead to potential errors in the future.

```
    /// @notice claim enabled status, 1 = false, 2 = true
     uint256 private claimEnabled = 1;
     /// @notice presale status if paused or not 1 = false, 2 = true;
     uint256 private paused = 1;
```

Recommendation

It is recommended to refactor these variables to use the `bool` data type. This will make the code more readable, reduce potential points of failure, and optimize gas usage. The associated code segments that interact with these variables should also be adjusted to reflect boolean use.

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	TestPresale.sol#L882,1087,1283
Status	Unresolved

Description

The contract contains methods with multiple occurrences of the same calculation being performed. The calculation is repeated without utilizing a variable to store its result, which leads to redundant code, hinders code readability, and increases gas consumption. Each repetition of the calculation requires computational resources and can impact the performance of the contract, especially if the calculation is resource-intensive.

Specifically the value of the `availableTokens` is calculated inside the `_buyTokens` and `getTokenAmount` functions and `validRoundCheck` modifier.

```
uint256 availableTokens = round.totalTokens - round.totalSold;
```

Recommendation

To address this finding and enhance the efficiency and maintainability of the contract, it is recommended to refactor the code by assigning the calculation result to a variable once and then utilizing that variable throughout the method. By storing the calculation result in a variable, the contract eliminates the need for redundant calculations and optimizes code execution.

Refactoring the code to assign the calculation result to a variable has several benefits. It improves code readability by making the purpose and intent of the calculation explicit. It also reduces code redundancy, making the method more concise, easier to maintain, and gas effective. Additionally, by performing the calculation once and reusing the variable, the contract improves performance by avoiding unnecessary computations

ODM - Oracle Decimal Mismatch

Criticality	Minor / Informative
Location	TestPresale.sol#L1321
Status	Unresolved

Description

The contract relies on data retrieved from an external Oracle to make critical calculations. However, the contract does not include a verification step to align the decimal precision of the retrieved data with the precision expected by the contract's internal calculations. This mismatch in decimal precision can introduce substantial errors in calculations involving decimal values.

```
    function getLatestPrice() public view returns (uint256) {
        (, int256 price, , , ) = priceFeed.latestRoundData();
        return uint256(price);
    }
```

Recommendation

The team is advised to retrieve the decimals precision from the Oracle API in order to proceed with the appropriate adjustments to the internal decimals representation.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	TestPresale.sol#L1187
Status	Unresolved

Description

The contract modifies the state of the following variables without checking if their current value is the same as the one given as an argument. As a result, the contract performs redundant storage writes, when the provided parameter matches the current state of the variables, leading to unnecessary gas consumption and inefficiencies in contract execution.

```
    function setPrice(
        uint256 _newPrice,
        uint256 _nextPrice
    ) external onlyOwner {
        Round storage round = rounds[currentRound];
        round.currentPrice = _newPrice;
        round.nextPrice = _nextPrice;
    }
```

Recommendation

The team is advised to implement additional checks within to prevent redundant storage writes when the provided argument matches the current state of the variables. By incorporating statements to compare the new values with the existing values before proceeding with any state modification, the contract can avoid unnecessary storage operations, thereby optimizing gas usage.

ZD - Zero Division

Criticality	Minor / Informative
Location	TestPresale.sol#L1187,1284,1295
Status	Unresolved

Description

The contract is using variables that may be set to zero as denominators. This can lead to unpredictable and potentially harmful results, such as a transaction revert.

The owner could sets the values of `currentPrice` and `nextPrice` to zero by invoking the `setPrice` function. Specifically, when these values are set to zero, the calculations will attempt to divide by zero, leading to unpredictable behavior. A division by zero will cause a transaction to revert, potentially breaking expected contract functionality and causing user interactions to fail.

```
   function setPrice(
        uint256 _newPrice,
        uint256 _nextPrice
    ) external onlyOwner {
        Round storage round = rounds[currentRound];
        round.currentPrice = _newPrice;
        round.nextPrice = _nextPrice;
    }

    uint256 tokensAtCurrentPrice = (_usdAmount * 10 ** TOKEN_DECIMALS) /
            round.currentPrice;
    ...
    uint256 tokensFromNewRound = (usdLeft * 10 ** TOKEN_DECIMALS) /
                round.nextPrice;
```

Recommendation

It is recommended to incorporate checks within the setSalePrice function to prevent the salePrice variable from being set to zero. Additionally, it is important to handle division by zero appropriately in the code to avoid unintended behavior and to ensure the reliability and safety of the contract. The contract should ensure that the divisor is always non-zero before performing a division operation. It should prevent the variables to be set to zero, or should not allow the execution of the corresponding statements.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	TestPresale.sol#L1164,1173,1187,1243
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
    function enableClaims() external onlyOwner {
        if (claimEnabled == 2) {
            revert ClaimsAreEnabledAlready();
        }
        claimEnabled = 2;
    }

    function setEndDateForCurrentRound(uint256 endDate) external
onlyOwner {
        Round storage round = rounds[currentRound];
        if (endDate < block.timestamp) {
            revert CannotSetDateInPast();
        }
        round.endTime = endDate;
    }

    function setPrice(
        uint256 _newPrice,
        uint256 _nextPrice
    ) external onlyOwner {
        Round storage round = rounds[currentRound];
        round.currentPrice = _newPrice;
        round.nextPrice = _nextPrice;
    }

    function togglePauseStatus() external onlyOwner {
        if (paused == 1) {
            paused = 2;
        } else if (paused == 2) {
            paused = 1;
        }
    }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	TestPresale.sol#L1137,1312
Status	Unresolved

Description

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

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

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

```
uint256 oneEth = (getLatestPrice() * 1e10);
ethRequired = (_usdAmount * 1e12) / oneEth;
...
uint256 pricePerWei = (perEthPrice * 1e10) / 10 ** TOKEN_DECIMALS;
usdAmount = (amount * pricePerWei) / 1e12;
```

Recommendation

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

The following example depicts 3 tokens with different decimals precision.

ERC20	Decimals
Token 1	6
Token 2	9
Token 3	18

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

RSK - Redundant Storage Keyword

Criticality	Minor / Informative
Location	TestPresale.sol#L1282,1328,1338,1344
Status	Unresolved

Description

The contract uses the `storage` keyword in a view function. The `storage` keyword is used to persist data on the contract's storage. View functions are functions that do not modify the state of the contract and do not perform any actions that cost gas (such as sending a transaction). As a result, the use of the `storage` keyword in view functions is redundant.

```
Round storage round
```

Recommendation

It is generally considered good practice to avoid using the `storage` keyword in view functions because it is unnecessary and can make the code less readable.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	TestPresale.sol#L552,824,826,829,897,1037,1188,1189,1198,1207,1216,1280
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.

```
function DOMAIN_SEPARATOR() external view returns (bytes32);
uint256 private constant tokensPerRound = 112_500_000 * 10 **
TOKEN_DECIMALS
uint256 private constant totalRounds = 12
IERC20 public immutable USDT
uint256 _round
uint256 _usdAmount
uint256 _newPrice
uint256 _nextPrice
address _token
address _multisig
address _oToken
```

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.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	TestPresale.sol#L105,342,396,405,436,697,706,721,755
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function _reentrancyGuardEntered() internal view returns (bool) {
        return _status == ENTERED;
    }

function sendValue(address payable recipient, uint256 amount) internal {
        if (address(this).balance < amount) {
            revert AddressInsufficientBalance(address(this));
        }

        (bool success, ) = recipient.call{value: amount}("");
        if (!success) {
            revert FailedInnerCall();
        }
    }

...
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	TestPresale.sol#L1110,1115,1138,1312,1315
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 pricePerWei = (perEthPrice * 1e10) / 10 ** TOKEN_DECIMALS
usdAmount = (amount * pricePerWei) / 1e12
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	TestPresale.sol#L1108
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
uint256 extraUsd
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	TestPresale.sol#L275
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.

```
_pendingOwner = newOwner
```

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.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	TestPresale.sol#L452
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	TestPresale.sol#L4,28,115,142,244,305,467,560,642,650,768
Status	Unresolved

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
pragma solidity 0.8.19;
pragma solidity ^0.8.0;
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	TestPresale.sol#L4
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 ^0.8.0;
```

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**
AggregatorV3Interface	Interface			
	decimals	External		-
	description	External		-
	version	External		-
	getRoundData	External		-
	latestRoundData	External		-
ReentrancyGuard	Implementation			
		Public	✓	-
	_nonReentrantBefore	Private	✓	
	_nonReentrantAfter	Private	✓	
	_reentrancyGuardEntered	Internal		
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		
Ownable	Implementation	Context		
		Public	✓	-

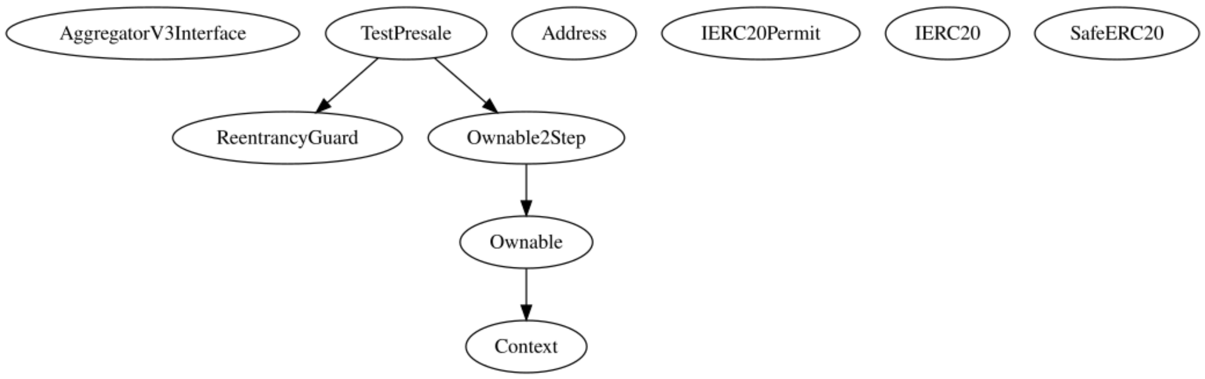
	owner	Public		-
	_checkOwner	Internal		
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
	_transferOwnership	Internal	✓	
Ownable2Step	Implementation	Ownable		
	pendingOwner	Public		-
	transferOwnership	Public	✓	onlyOwner
	_transferOwnership	Internal	✓	
	acceptOwnership	Public	✓	-
Address	Library			
	sendValue	Internal	✓	
	functionCall	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionStaticCall	Internal		
	functionDelegateCall	Internal	✓	
	verifyCallResultFromTarget	Internal		
	verifyCallResult	Internal		
	_revert	Private		
IERC20Permit	Interface			

	permit	External	✓	-
	nonces	External		-
	DOMAIN_SEPARATOR	External		-
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
SafeERC20	Library			
	safeTransfer	Internal	✓	
	safeTransferFrom	Internal	✓	
	safeIncreaseAllowance	Internal	✓	
	safeDecreaseAllowance	Internal	✓	
	forceApprove	Internal	✓	
	_callOptionalReturn	Private	✓	
	_callOptionalReturnBool	Private	✓	
TestPresale	Implementation	Ownable2Step, ReentrancyGuard		

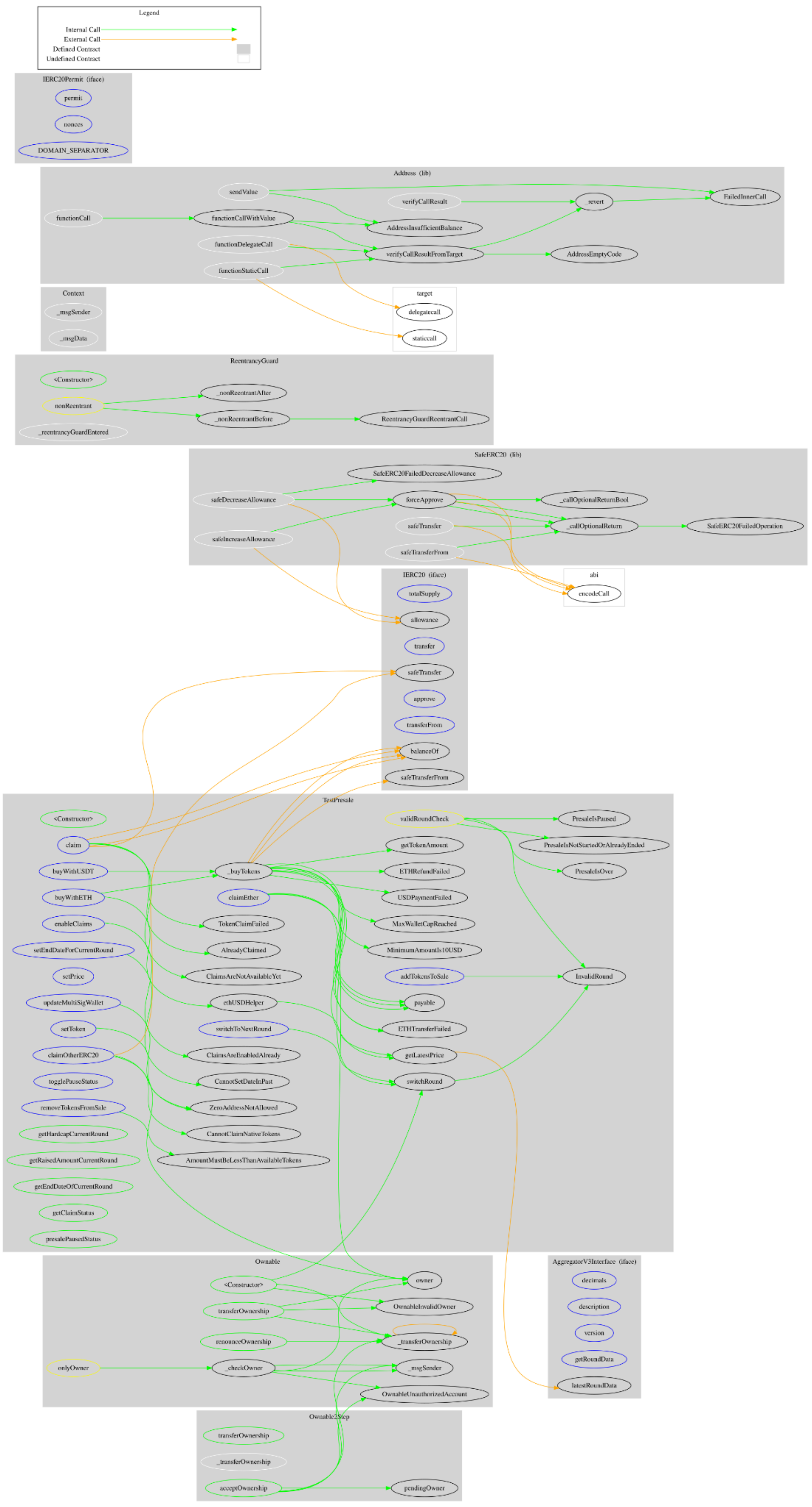
		Public	✓	Ownable
	switchRound	Internal	✓	
	buyWithUSDT	External	✓	validRoundCheck nonReentrant
	buyWithETH	External	Payable	validRoundCheck nonReentrant
	claim	External	✓	nonReentrant
	_buyTokens	Private	✓	
	enableClaims	External	✓	onlyOwner
	setEndDateForCurrentRound	External	✓	onlyOwner
	setPrice	External	✓	onlyOwner
	setToken	External	✓	onlyOwner
	updateMultiSigWallet	External	✓	onlyOwner
	claimOtherERC20	External	✓	onlyOwner
	switchToNextRound	External	✓	onlyOwner
	claimEther	External	✓	onlyOwner
	togglePauseStatus	External	✓	onlyOwner
	addTokensToSale	External	✓	onlyOwner
	removeTokensFromSale	External	✓	onlyOwner
	getTokenAmount	Public		-
	ethUSDHelper	Public		-
	getLatestPrice	Public		-
	getHardcapCurrentRound	Public		-
	getRaisedAmountCurrentRound	Public		-

	getEndDateOfCurrentRound	Public		-
	getClaimStatus	Public		-
	presalePausedStatus	Public		-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IERC20Permit (iface)
permit
nonces
DOMAIN_SEPARATOR
Address (lib)
sendValue
functionCall
functionCallWithValue
functionDelegateCall
functionStaticCall
verifyCallResult
AddressInsufficientBalance
verifyCallResultFromTarget
revert
AddressEmptyCode
FailedInnerCall
Context
_msgSender
_msgData
target
delegatecall
staticcall
ReentrancyGuard
<Constructor>
nonReentrant
_reentrancyGuardEntered
_nonReentrantAfter
_nonReentrantBefore
ReentrancyGuardReentrantCall
SafeERC20 (lib)
SafeERC20FailedDecreaseAllowance
safeDecreaseAllowance
safeIncreaseAllowance
forceApprove
safeTransfer
safeTransferFrom
_callOptionalReturnBool
_callOptionalReturn
SafeERC20FailedOperation
abi
encodeCall
IERC20 (iface)
totalSupply
allowance
transfer
safeTransfer
approve
transferFrom
balanceOf
safeTransferFrom
TestPresale
<Constructor>
claim
buyWithUSDT
buyWithETH
enableClaims
setEndDateForCurrentRound
setPrice
updateMultiSigWallet
setToken
claimOtherERC20
togglePauseStatus
removeTokensFromSale
getHardcapCurrentRound
getRaisedAmountCurrentRound
getEndDateOfCurrentRound
getClaimStatus
presalePausedStatus
_buyTokens
claimEther
TokenClaimFailed
AlreadyClaimed
ClaimsAreNotAvailableYet
ethUSDHelper
switchToNextRound
ClaimsAreEnabledAlready
CannotSetDateInPast
ZeroAddressNotAllowed
CannotClaimNativeTokens
AmountMustBeLessThanAvailableTokens
validRoundCheck
getTokenAmount
ETHRefundFailed
USDPaymentFailed
MaxWalletCapReached
MinimumAmountIs10USD
addTokensToSale
payable
ETHTransferFailed
getLatestPrice
switchRound
PresaleIsPaused
PresaleIsNotStartedOrAlreadyEnded
PresaleIsOver
InvalidRound
Ownable
<Constructor>
transferOwnership
renounceOwnership
onlyOwner
_checkOwner
owner
OwnableInvalidOwner
_transferOwnership
_msgSender
OwnableUnauthorizedAccount
AggregatorV3Interface (iface)
decimals
description
version
getRoundData
latestRoundData
Ownable2Step
transferOwnership
_transferOwnership
acceptOwnership
pendingOwner

Summary

AIGPROJECT contract implements a token and rewards mechanism. The audit scope is to check for security vulnerabilities, validate the business logic and propose potential optimizations. The contract is missing the fundamental principles of a Solidity smart contract regarding gas consumption, code readability, and data structures. According to the previously mentioned issues, the contract cannot be assumed that it is in a production-ready state. Given these issues, it is not advisable to assume that the contract is in a production-ready state. The development team is strongly encouraged to re-evaluate the business logic and Solidity guidelines to ensure that the contract adheres to established best practices and security measures. It is recommended that the team review the contract's gas consumption and optimize it accordingly to minimize costs and improve the contract's efficiency. The code's readability should also be improved by simplifying function definitions and using descriptive variable names, as this will enhance the contract's auditability and maintenance.

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