Cyberscope

Audit Report

AIGPROJECT

November 2023

SHA256 8fc86014d7735acfbdb87d97612dd39be3b9f33e1414211e9095e6586565ef28

Audited by © cyberscope

Table of Contents

Review

Testing Deploy	https://goerli.etherscan.io/address/0x70d87aff261ffad995e2a863255cda7257c803ff

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 https://github.com/cyberscope-io/audits/blob/main/aig/v3/presale.pdf
Corrected Phase 4	30 Oct 2023 https://github.com/cyberscope-io/audits/blob/main/aig/v4/presale.pdf
Corrected Phase 5	02 Nov 2023

Source Files

Filename	SHA256
AiGoldPresale.sol	8fc86014d7735acfbdb87d97612dd39be3b9f33e1414211e9095e6586565ef28

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 14 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, 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.

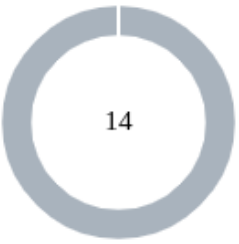
Start Claim

The `enableClaims` function is designed for a contract owner to initiate a 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 true, 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	0
Medium	0
Minor / Informative	14

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	1	13	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	RCC	Redundant Condition Check	Unresolved
●	PRDI	Potential Reward Decimal Inconsistency	Acknowledged
●	DPI	Decimals Precision Inconsistency	Acknowledged
●	UST	Uninitialized Sale Token	Acknowledged
●	IBC	Insufficient Balance Check	Acknowledged
●	RC	Repetitive Calculations	Acknowledged
●	CCR	Contract Centralization Risk	Acknowledged
●	OCTD	Transfers Contract's Tokens	Acknowledged
●	IPF	Inconsistent Price Functionality	Acknowledged
●	RSK	Redundant Storage Keyword	Acknowledged
●	L04	Conformance to Solidity Naming Conventions	Acknowledged
●	L09	Dead Code Elimination	Acknowledged
●	L17	Usage of Solidity Assembly	Acknowledged
●	L18	Multiple Pragma Directives	Acknowledged

RCC - Redundant Condition Check

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

Description

The contract uses the The contract checks the value of `totalAvailableTokens` against `tokensAtCurrentPrice`. The `if` statement checks if `totalAvailableTokens` is greater than or equal to `tokensAtCurrentPrice`. Following this, the `else if` statement that checks the contrary condition, (if `totalAvailableTokens` is less than `tokensAtCurrentPrice`). Given that the first condition already checks for the opposite scenario, the `else if` statement becomes redundant. If the first condition is not met, it is implicitly understood that `totalAvailableTokens` is less than `tokensAtCurrentPrice`.

```
if (totalAvailableTokens >= tokensAtCurrentPrice) {
    return tokensAtCurrentPrice;
} else if (
    totalAvailableTokens < tokensAtCurrentPrice
)
```

Recommendation

It is recommended to simplify the code by using the `else` keyword instead of the `else if` statement. This will make the code more concise and easier to understand, as the else condition will inherently handle the scenario where `totalAvailableTokens` is less than `tokensAtCurrentPrice`.

PRDI - Potential Reward Decimal Inconsistency

Criticality	Minor / Informative
Location	AiGoldPresale.sol#L1045
Status	Acknowledged

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) {
            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 != availableToClaim) {
            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.

Team Update

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

User avalaible tokens are in correct decimals, so can't claim more than that

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	TestPresale.sol#L1138,1343
Status	Acknowledged

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.

Specifically, the contract use the constant value of `6` for `USDT_DECIMALS` instead of retrieving the actual decimal number from the corresponding `USDT` address. This approach, while simpler, can lead to inaccuracies and potential issues, especially if the actual decimal value of the token changes or if there's a need to integrate with other systems or contracts that rely on accurate decimal representation.

```
uint256 multiplier = BASE_DECIMALS + priceFeedDecimals - USDT_DECIMALS;
...
uint256 divisor  = BASE_DECIMALS + priceFeedDecimals - USDT_DECIMALS;
```

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. Additionally, it's recommended to dynamically retrieve the decimals value from the `USD` contract address rather than hard-coding it.

UST - Uninitialized Sale Token

Criticality	Minor / Informative
Location	AiGoldPresale.sol#L1170,1216
Status	Acknowledged

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) {
            revert ClaimsAreEnabledAlready();
        }

        claimEnabled = true;
        emit ClaimEnabled(true);
    }

     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.

Team Update

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

Token isn't deployed yet and owner'll be setting it once presale completes

IBC - Insufficient Balance Check

Criticality	Minor / Informative
Location	AiGoldPresale.sol#L1277,1284
Status	Acknowledged

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.

```
    /// @dev add tokens to current sale round
    /// @param amount: token amount to add
    function addTokensToSale(uint256 amount) external onlyOwner {
        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.

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	AiGoldPresale.sol#L1086,1304
Status	Acknowledged

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

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	AiGoldPresale.sol#L1170,1181,1196
Status	Acknowledged

Description

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

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) {
            revert ClaimsAreEnabledAlready();
        }

        claimEnabled = true;
        emit ClaimEnabled(true);
    }

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

    function setPrice(
        uint256 _newPrice,
        uint256 _nextPrice
    ) external onlyOwner {
        if(_newPrice == 0 || _newPrice == 0){
            revert PriceCantNotBeZero();
        }

        ...
    }
```

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.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	AiGoldPresale.sol#L1216,1237
Status	Acknowledged

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 othertkn,
        uint256 amount
    ) external onlyOwner {
        if (othertkn == address(token)) {
            revert CannotClaimNativeTokens();
        }
        IERC20 otherToken = IERC20(othertkn);
        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.

Team Update

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

Use of this is, to claim other erc20 tokens if accidently sent by users.

IPF - Inconsistent Price Functionality

Criticality	Minor / Informative
Location	AiGoldPresale.sol#L810,1203
Status	Acknowledged

Description

The contract stores the price variables within each round by storing two distinct values, `currentPrice` and `nextPrice`. This design allows the contract owner to set these prices arbitrarily by invoking the `setPrice` function. The use of separate variables for current and next prices within the same round could lead to inconsistencies and confusion. The logic within the `setPrice` function contains checks against the price being zero and against setting a price that matches the current or next price already recorded for the round. As a result maintaining two separate variables for pricing within the contract can introduce confusion and inconsistencies in the system's pricing structure.

```
  struct Round {
        uint256 currentPrice;
        uint256 nextPrice;
        uint256 totalTokens;
        uint256 totalSold;
        uint256 startTime;
        uint256 endTime;
    }

    function setPrice(
        uint256 _newPrice,
        uint256 _nextPrice
    ) external onlyOwner {
        if(_newPrice == 0 || _nextPrice == 0){
            revert PriceCantNotBeZero();
        }

        Round storage round = rounds[currentRound];

        /// one of value can stay same if owner want to update only one
value
        if(_newPrice == round.currentPrice && _nextPrice ==
round.nextPrice){
            revert ValuesAlreadyExists();
        }
        round.currentPrice = _newPrice;
        round.nextPrice = _nextPrice;

        emit PriceUpdated(_newPrice, _nextPrice);
 }
```

Recommendation

It is recommended to streamline the pricing structure within each round. By refactoring the contract to use a single price variable, it would reduce complexity and storage needs, while also eliminating the risk of price inconsistencies. This change would not only simplify the contract's logic but also could potentially save on gas costs and enhance contract security.

Team Update

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

The round.currentPrice and round.nextPrice is part of contract design

RSK - Redundant Storage Keyword

Criticality	Minor / Informative
Location	AiGoldPresale.sol#L1309
Status	Acknowledged

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	AiGoldPresale.sol#L552,832,834,837,912,1032,1079,1204,1205,1225,1234,1307
Status	Acknowledged

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
```

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	AiGoldPresale.sol#L105,342,396,405,436,697,706,721,755
Status	Acknowledged

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.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	AiGoldPresale.sol#L452
Status	Acknowledged

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	AiGoldPresale.sol#L4,28,115,142,244,305,467,560,642,650,768
Status	Acknowledged

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 0.8.21;
pragma solidity ^0.8.0;
pragma solidity ^0.8.20;
```

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.

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		

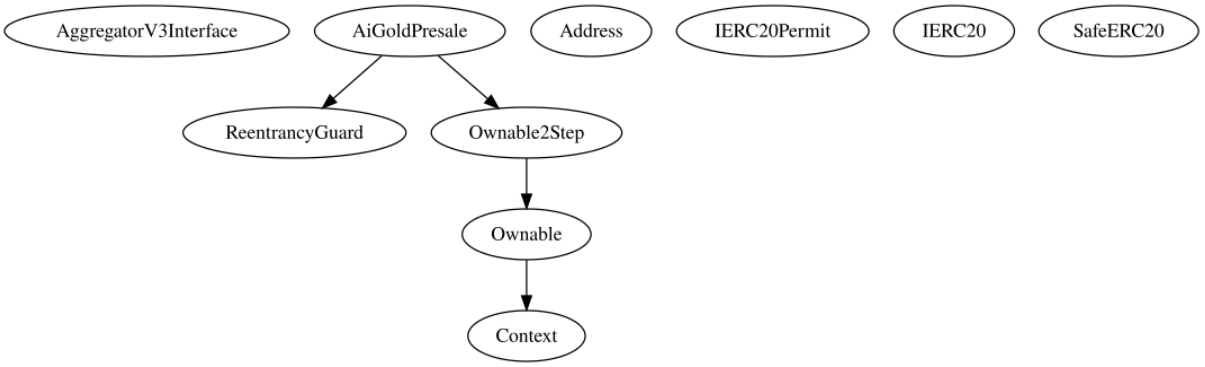
		Public	✓	-
	owner	Public		-
	_checkOwner	Internal		
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
	_transferOwnership	Internal	✓	
Ownable2Step	Implementation	Ownable		
	pendingOwner	Public		-
	transferOwnership	Public	✓	onlyOwner
	_transferOwnership	Internal	✓	
	acceptOwnership	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	✓	
AiGoldPresale	Implementation	Ownable2Step,		

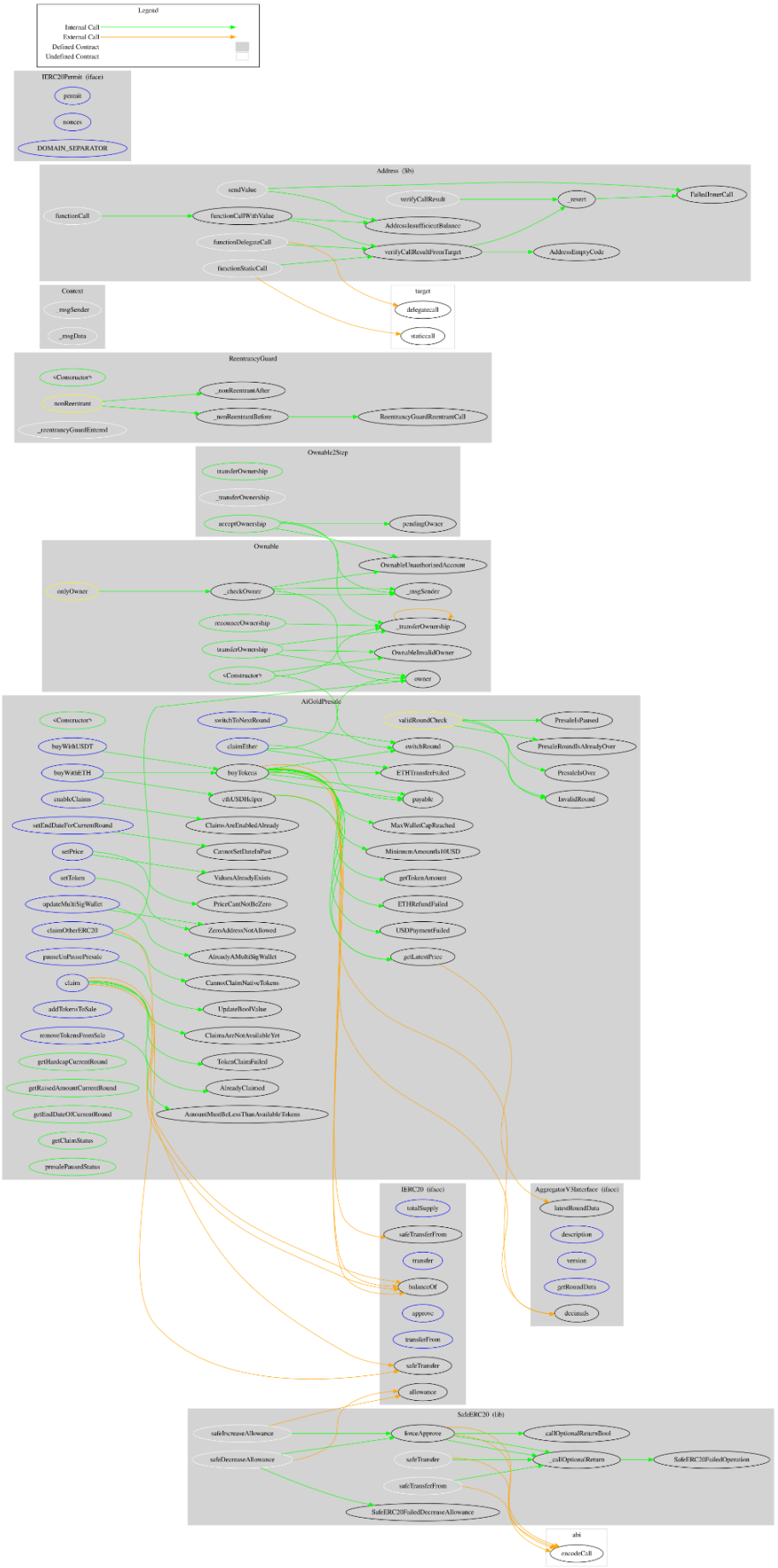
		ReentrancyGuard		
		Public	✓	Ownable
	switchRound	Internal	✓	
	buyWithUSDT	External	✓	validRoundCheck nonReentrant
	buyWithETH	External	Payable	validRoundCheck nonReentrant
	claim	External	✓	nonReentrant
	buyTokens	Private	✓	
	enableClaims	External	✓	onlyOwner
	setEndDateForCurrentRound	External	✓	onlyOwner
	setPrice	External	✓	onlyOwner
	setToken	External	✓	onlyOwner
	updateMultiSigWallet	External	✓	onlyOwner
	claimOtherERC20	External	✓	onlyOwner
	switchToNextRound	External	✓	onlyOwner
	claimEther	External	✓	onlyOwner
	pauseUnPausePresale	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
verifyCallResult
revert
FailedInnerCall
functionCall
functionCallWithValue
AddressInsufficientBalance
functionDelegateCall
verifyCallResultFromTarget
AddressEmptyCode
functionStaticCall
Context
_msgSender
_msgData
target
delegatecall
staticcall
ReentrancyGuard
<Constructor>
nonReentrant
_nonReentrantAfter
_nonReentrantBefore
ReentrancyGuardReentrantCall
_reentrancyGuardEntered
Ownable2Step
transferOwnership
_transferOwnership
acceptOwnership
pendingOwner
Ownable
OwnableUnauthorizedAccount
onlyOwner
_checkOwner
renounceOwnership
OwnableInvalidOwner
owner
AIGoldPresale
switchToNextRound
validRoundCheck
PresaleIsPaused
claimEther
switchRound
PresaleRoundIsAlreadyOver
buyWithUSDT
buyWithETH
buyTokens
ETHTransferFailed
PresaleIsOver
enableClaims
ethUSDHelper
payable
InvalidRound
setEndDateForCurrentRound
ClaimsAreEnabledAlready
MaxWalletCapReached
setPrice
CannotSetDateInPast
MinimumAmountIs10USD
setToken
ValuesAlreadyExists
getTokenAmount
updateMultiSigWallet
PriceCantNotBeZero
ETHRefundFailed
claimOtherERC20
ZeroAddressNotAllowed
USDPaymentFailed
pauseUnPausePresale
AlreadyAMultiSigWallet
getLatestPrice
claim
CannotClaimNativeTokens
addTokensToSale
UpdateBoolValue
removeTokensFromSale
ClaimsAreNotAvailableYet
getHardcapCurrentRound
TokenClaimFailed
getRaisedAmountCurrentRound
AlreadyClaimed
getEndDateOfCurrentRound
AmountMustBeLessThanAvailableTokens
getClaimStatus
presalePausedStatus
IERC20 (iface)
totalSupply
safeTransferFrom
transfer
balanceOf
approve
transferFrom
safeTransfer
allowance
AggregatorV3Interface (iface)
latestRoundData
description
version
getRoundData
decimals
SafeERC20 (lib)
safeIncreaseAllowance
forceApprove
callOptionalReturnBool
safeDecreaseAllowance
safeTransfer
_callOptionalReturn
SafeERC20FailedOperation
safeTransferFrom
SafeERC20FailedDecreaseAllowance
abi
encodeCall

Summary

AIGPROJECT Presale contract implements a financial 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