Cyberscope

Audit Report

AIGPROJECT

Aug 2023

SHA256 7e0feb2d7103abda968bde453228440c00b565d90fc8ffb430791582b3f8d58e

Audited by © cyberscope

Table of Contents

Table of Contents **1**
Review **4**
Audit Updates 4
Source Files 4
Overview **5**
Initialization 5
Sale Phases 5
Price Calculation 6
Token Purchasing 6
Buy With ETH 6
Buy With USD 7
Start Claim 7
Claim 7
Findings Breakdown **8**
Diagnostics **9**
MI - Missing Implementation 11
Description 11
Recommendation 12
ZC - Zero Calculation 13
Description 13
Recommendation 13
PU - Potential Underflow 14
Description 14
Recommendation 14
UST - Uninitialized Sale Token 15
Description 15
Recommendation 15
MC - Missing Check 17
Description 17
Recommendation 18
IBC - Insufficient Balance Check 19
Description 19
Recommendation 19
CR - Code Repetition 20
Description 20
Recommendation 20
ZD - Zero Division 22
Description 22
Recommendation 22

DPI - Decimals Precision Inconsistency 23
Description 23
Recommendation 24
ST - Stops Transactions 25
Description 25
Recommendation 25
PRDI - Potential Reward Decimal Inconsistency 26
Description 26
Recommendation 26
RVA - Redundant Variable Assignment 27
Description 27
Recommendation 27
RCC - Redundant Conditional Check 28
Description 28
Recommendation 28
OCTD - Transfers Contract's Tokens 29
Description 29
Recommendation 29
RSW - Redundant Storage Writes 30
Description 30
Recommendation 30
MEE - Missing Events Emission 31
Description 31
Recommendation 31
IDI - Immutable Declaration Improvement 33
Description 33
Recommendation 33
L04 - Conformance to Solidity Naming Conventions 34
Description 34
Recommendation 34
L06 - Missing Events Access Control 36
Description 36
Recommendation 36
L07 - Missing Events Arithmetic 37
Description 37
Recommendation 37
L09 - Dead Code Elimination 38
Description 38
Recommendation 39
L11 - Unnecessary Boolean equality 40
Description 40
Recommendation 40

L13 - Divide before Multiply Operation 41
Description 41
Recommendation 41
L14 - Uninitialized Variables in Local Scope 42
Description 42
Recommendation 42
L16 - Validate Variable Setters 43
Description 43
Recommendation 43
L17 - Usage of Solidity Assembly 44
Description 44
Recommendation 44
L18 - Multiple Pragma Directives 45
Description 45
Recommendation 45
L19 - Stable Compiler Version 46
Description 46
Recommendation 46
L20 - Succeeded Transfer Check 47
Description 47
Recommendation 47
Functions Analysis **48**
Inheritance Graph **55**
Flow Graph **56**
Summary **57**
Disclaimer **58**
About Cyberscope **59**

Review

Repository	https://github.com/aigtkn/presale_widgetbuy1/blob/main/PRESALEWIDGETBUYBUTTONCONTRACT1.txt
Commit	9af5dfe14dcfb39a2fd20ba74ce94b61ef55e539
Testing Deploy	https://testnet.bscscan.com/address/0x3306a297113e5baa77b582cac83be57be9236b62

Audit Updates

Initial Audit	09 Aug 2023

Source Files

Filename	SHA256
contracts/AiGold_PreSale.sol	7e0feb2d7103abda968bde453228440c00b565d90fc8ffb430791582b3f8d58e

Overview

The "AiGold_PreSale" contract is designed to manage a presale event for a specific token on the Ethereum blockchain. It integrates with the Uniswap V2 router and has provisions for setting and updating sale prices, managing different sale steps, and handling token purchases with both ETH and stablecoins like USDT. The contract ensures that the presale operates within defined timeframes, adheres to a hard cap, and allows users to claim their purchased tokens after the sale. Additionally, the owner has the authority to pause the presale, adjust sale parameters, and withdraw tokens or ETH.

Out of Scope Address: The contract employs the `dataOracle` address `0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419` to fetch real-time ETH prices. Specifically, the `getETHLatestPrice` function utilizes this oracle to retrieve and compute the accurate ETH price. It's important to note that the integrity, functionality, and security associated with the dataOracle 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.

Initialization

Upon the contract's deployment, it initializes the UniswapV2Router using a predefined address. It sets the base decimal for tokens as 10**18, and the sale price is set to 0.0035 USD. The hard cap for tokens is fixed at 1,350,000,000, which is also the total number of tokens available for presale. The sale starts immediately upon contract deployment and ends 90 days from then. A couple of addresses, namely dataOracle, dAddress, and the address for the USDT token are also defined. The first sale phase is started with startStep(1).

Sale Phases

The presale is divided into three phases or steps. For each step, the sale price is adjusted, as well as the hardcap size in USD, the total USD value for presale, and the in-sale USD value:

Step 1: Sale price is 0.0035 USD with a hardcap size of 3,000,000 USD. Step 2: Sale price is 0.0045 USD with a hardcap size of 3,500,000 USD. Step 3: Sale price is 0.0075 USD with a hardcap size of 4,000,000 USD.

Modify Steps and Values The contract owner has the capability to:

- Change the sale phase manually.
- Adjust the number of tokens in the sale (either add or subtract).
- Modify the hardcap size in USD (either add or subtract).
- Adjust the sale price and its subsequent value.
- Modify the total tokens available for the presale.
- Presale Control:pause or unpause the presale. When paused, no purchases can be made.

Price Calculation

The sale supports both ETH and stablecoin USDT for purchases. The `calculatePrice` function determines the necessary amount in USD for a specific amount of tokens. It ensures that buyers receive the correct price even when transitioning from one phase to another or when the hardcap is exceeded.

Token Purchasing

Users can buy tokens using either ETH `buyWithETH` or stablecoins like USDT `buyWithUSD` . The buying functions ensure that:

- The purchase is within the sale duration.
- The amount to buy is within the allowed range.
- Payments surpass the calculated price.
- If a user overpays in ETH, the excess is returned. Token purchases are emitted as events.

Buy With ETH

The `buyWithETH` function allows users to purchase a certain amount of tokens using ETH. After determining the required ETH for the desired token amount based on the current ETH to USD exchange rate, it checks the sent ETH against the required amount. If users overpay, they receive a refund for the excess. The function updates the sale status depending on the current phase and conditions, adjusting available tokens and their

equivalent USD value. Successful purchases result in the corresponding ETH being transferred to a designated address, with any surplus ETH being returned to the buyer.

Buy With USD

The `buyWithUSD` function allows users to purchase a certain amount of tokens using USD-based stablecoins, like USDT or USDC, under specific conditions and while the function isn't paused. If the sale is still active and hasn't reached its final phase, the function adjusts the available tokens for sale and their equivalent value. It then checks whether the user has provided the necessary permissions for the stablecoin transfer and attempts to deduct the required payment from the user's account, transferring it to a designated recipient. If the transaction is successful, a purchase event is emitted detailing the purchase specifics.

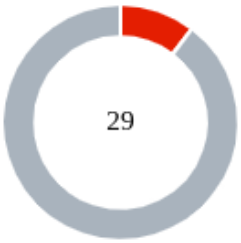
Start Claim

The `startClaim` function is designed for a contract owner to initiate a claim process for tokens. It takes three parameters: a starting time for the claim (_claimStart), the amount of tokens to be claimed (tokensAmount), and the address of the token to be claimed (_saleToken). For the function to proceed, it ensures that the provided start time is after both the endTime (presumably the end of a sale or event) and the current blockchain timestamp. It also checks that a valid token address is provided and that a claim hasn't been set previously. If these conditions are met, it updates the claim start time and token details in the contract's state. Lastly, it transfers the specified token amount from the calling user to the contract, requiring the user to have sufficient token balance.

Claim

The `claim` function allows users to claim tokens they're entitled to, provided certain conditions are met and the function isn't paused. Before tokens can be claimed, the function checks if the sale token has been set, ensures that the current time is past the designated `claimStart` time, and verifies that the caller hasn't claimed before. If the user meets these conditions and has a deposit greater than zero, their deposit amount is retrieved and then deleted from the contract's record. The specified amount of the sale token is then transferred to the user. Lastly, an event named `TokensClaimed` is emitted, logging the user's address, the amount they claimed, and the timestamp of the claim.

Findings Breakdown



Critical	3
Medium	0
Minor / Informative	26

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	3	0	0	0
Medium	0	0	0	0
Minor / Informative	26	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	MI	Missing Implementation	Unresolved
●	ZC	Zero Calculation	Unresolved
●	PU	Potential Underflow	Unresolved
●	UST	Uninitialized Sale Token	Unresolved
●	MC	Missing Check	Unresolved
●	IBC	Insufficient Balance Check	Unresolved
●	CR	Code Repetition	Unresolved
●	ZD	Zero Division	Unresolved
●	DPI	Decimals Precision Inconsistency	Unresolved
●	ST	Stops Transactions	Unresolved
●	PRDI	Potential Reward Decimal Inconsistency	Unresolved
●	RVA	Redundant Variable Assignment	Unresolved
●	RCC	Redundant Conditional Check	Unresolved
●	OCTD	Transfers Contract's Tokens	Unresolved

●	RSW	Redundant Storage Writes	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L06	Missing Events Access Control	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L11	Unnecessary Boolean equality	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
●	L20	Succeeded Transfer Check	Unresolved

MI - Missing Implementation

Criticality	Critical
Location	contracts/AiGold_PreSale.sol#L1255
Status	Unresolved

Description

The contract is designed to allow the purchase of tokens with USD through the `buyWithUSD` function. However, there is no safeguard to restrict the value of `purchaseToken`. The only valid implementation available in the function is for when `purchaseToken` equals 0. However, the function contains a condition that also checks if `purchaseToken` is 1, but there's no subsequent code that properly handles this case. As a result, if the function is called with `purchaseToken` set to 1, the contract will not perform the expected actions, which could lead to potential disruptions in contract operations or unintended behavior.

```
 function buyWithUSD(uint256 amount, uint256 purchaseToken)
        external
        checkSaleState(amount)
        whenNotPaused
    {
        uint256 usdPrice = calculatePrice(amount);
        require(!(usdPrice == 0 && currentStep == 3),"Presale it's over,
sorry!");
        if (purchaseToken == 0 || purchaseToken == 1) usdPrice =
usdPrice; //USDT and USDC have 6 decimals

        if (usdPrice > inSaleUSDvalue) {
            uint256 upfrontSaleUSDvalue = usdPrice - inSaleUSDvalue;
            startStep(currentStep + 1);
            inSale -= amount;
            inSaleUSDvalue -= upfrontSaleUSDvalue;
        } else if (usdPrice == inSaleUSDvalue && currentStep == 3) {
            amount = usdPrice / salePrice;
            inSale -= amount;
            inSaleUSDvalue -= usdPrice;
        } else {
            inSale -= amount;
            inSaleUSDvalue -= usdPrice;
        }
        userDeposits[_msgSender()] += (amount * (10**18));
        IERC20 tokenInterface;
        if (purchaseToken == 0) tokenInterface = IERC20(USDTtoken);

    ...
}
```

Recommendation

It is recommended to introduce proper checks to validate the `purchaseToken` value at the beginning of the function. This ensures that only acceptable values are processed. If the intention is for the contract to also handle cases where `purchaseToken` equals 1, then the code should be supplemented with the necessary logic to cater to this scenario. Otherwise, it would be prudent to explicitly restrict `purchaseToken` to only allowed values, e.g., 0, and revert transactions that do not meet this criterion. Implementing these measures will bolster the contract's robustness and protect against potential misuse or unexpected behavior.

ZC - Zero Calculation

Criticality	Critical
Location	contracts/AiGold_PreSale.sol#L1133
Status	Unresolved

Description

The contract contains the `settotalTokensForPresale` function which is intended to adjust the `inSale` and `totalTokensForPresale` variables. However, the calculation for `diffTokensale` subtracts the variable `totalTokensForPresale` from itself, through the intermediary variable `prevTotalTokensForPresale`. This results in `diffTokensale` always having a value of zero, regardless of the input or previous state of the contract. As a consequence, the `inSale` variable will never be increased, effectively making this function only capable of setting the `totalTokensForPresale` and not achieving the likely intended effect of also adjusting `inSale`.

```
    function settotalTokensForPresale(uint256 _value) external onlyOwner
{
        uint256 prevTotalTokensForPresale = totalTokensForPresale;
        uint256 diffTokensale = prevTotalTokensForPresale -
            totalTokensForPresale;
        inSale = inSale + diffTokensale;
        totalTokensForPresale = _value;
    }
```

Recommendation

It is recommended to refactor the `settotalTokensForPresale` function to correct the calculation of `diffTokensale` and ensure it accurately captures the difference in tokens for the presale. This change will ensure that the `inSale` variable is adjusted correctly in response to changes in the total tokens.

PU - Potential Underflow

Criticality	Critical
Location	contracts/AiGold_PreSale.sol#L1214,1255
Status	Unresolved

Description

The contract contains the `buyWithETH` and `buyWithUSD` functions. Within these functions, the contract consistently decrements the values of `inSale` by the given `amount` and `inSaleUSDvalue` by the calculated `upfrontSaleUSDvalue` or `usdPrice`. However, if the value of `upfrontSaleUSDvalue` or `usdPrice` surpasses `inSaleUSDvalue` or if the `amount` is larger than the existing value of `inSale`, then the subsequent subtractions would result in an underflow, which can have unintended consequences and could compromise the integrity of the contract's internal state.

For instance, if external functions like `removeHardcapsizeUSD` and `removeTokensInSale` are invoked, they could potentially reduce `inSaleUSDvalue` and `inSale`, making these values very small. This condition can set the stage for underflows during subsequent calculations within the purchase functions.

```
    inSaleUSDvalue -= upfrontSaleUSDvalue;
     inSale -= amount;
     inSaleUSDvalue -= usdPrice;
```

Recommendation

It is recommended to introduce checks prior to decreasing the `inSale` and `inSaleUSDvalue` values. By verifying that the subtraction operation will not lead to negative results, the contract can prevent potential underflows.

UST - Uninitialized Sale Token

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1326
Status	Unresolved

Description

The contract doesn't initialize the sale token during the setup of the presale state. As a result the contract cannot ensure that it will have a sufficient balance of the `saleToken` when users start claim. While the `startClaim` function allows the owner to set the `saleToken`, it is done after the presale phase has commenced. 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 startClaim(
        uint256 _claimStart,
        uint256 tokensAmount,
        address _saleToken
    ) external onlyOwner {
        require(
            _claimStart > endTime && _claimStart > block.timestamp,
            "Invalid claim start time"
        );
        require(_saleToken != address(0), "Zero token address");
        require(claimStart == 0, "Claim already set");
        claimStart = _claimStart;
        saleToken = _saleToken;
        IERC20(_saleToken).transferFrom(
            _msgSender(),
            address(this),
            tokensAmount
        );
    }
```

Recommendation

It is recommended to modify the contract structure to transfer the `saleToken` 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.

MC - Missing Check

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1108
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

For example if `inSaleUSDvalue` is set to a disproportionately large amount, it could lead to unexpected behavior or even vulnerabilities within the contract, affecting its integrity and the safety of funds.

```
 function addTokensInSale(uint256 _value) external onlyOwner {
    inSale = inSale + _value;
    }

    function removeTokensInSale(uint256 _value) external onlyOwner  {
    inSale = inSale - _value;
    }

    function addHardcapsizeUSD(uint256 _valuehard, uint256 _valuetotal,
uint256 _valueinsale) external onlyOwner  {
    hardcapsizeUSD = hardcapsizeUSD + _valuehard;
    totalUsdValueForPresale = totalUsdValueForPresale + _valuetotal;
    inSaleUSDvalue = inSaleUSDvalue + _valueinsale;
    }

    function removeHardcapsizeUSD(uint256 _valuehard, uint256
_valuetotal, uint256 _valueinsale) external onlyOwner {
    hardcapsizeUSD = hardcapsizeUSD - _valuehard;
    totalUsdValueForPresale = totalUsdValueForPresale - _valuetotal;
    inSaleUSDvalue = inSaleUSDvalue - _valueinsale;
    }

    function setSalePrice(uint256 _value, uint256 _valuenext) external
onlyOwner {
        salePrice = _value;
        nextPrice = _valuenext;
    }
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

IBC - Insufficient Balance Check

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1133
Status	Unresolved

Description

The contract contrains the `settotalTokensForPresale` function, which allows the owner to modify the value of `totalTokensForPresale`. However, this function currently lacks essential checks to verify the feasibility of the change. If the value of `totalTokensForPresale` 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 settotalTokensForPresale(uint256 _value) external onlyOwner
{
        uint256 prevTotalTokensForPresale = totalTokensForPresale;
        uint256 diffTokensale = prevTotalTokensForPresale -
            totalTokensForPresale;
        inSale = inSale + diffTokensale;
        totalTokensForPresale = _value;
    }
```

Recommendation

It is recommended to incorporate a verification mechanism within the `settotalTokensForPresale` function to ascertain the contract's token balance before allowing any adjustments to the `totalTokensForPresale` value. Specifically, when there's an increment in `totalTokensForPresale`, 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 `totalTokensForPresale` value. Only if the balance suffices should the change be permitted; otherwise, the operation should be rejected.

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

Specifically, the `buyWithETH` and `buyWithUSD` functions contain repetitive code segments.

```
uint256 usdPrice = calculatePrice(amount);
require(!(usdPrice == 0 && currentStep == 3),"Presale it's over,
sorry!");
...
if (usdPrice > inSaleUSDvalue) {
    uint256 upfrontSaleUSDvalue = usdPrice - inSaleUSDvalue;
    startStep(currentStep + 1);
    inSale -= amount;
    inSaleUSDvalue -= upfrontSaleUSDvalue;
} else if (usdPrice == inSaleUSDvalue && currentStep == 3) {
    amount = usdPrice / salePrice;
    inSale -= amount;
    inSaleUSDvalue -= usdPrice;
} else {
    inSale -= amount;
    inSaleUSDvalue -= usdPrice;
}
...
userDeposits[_msgSender()] += (amount * (10**18));
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever

possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

ZD - Zero Division

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1128,1166,1235,1270
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 value of `salePrice` to zero by invoking the `setSalePrice` function. Specifically, when the `salePrice` is set to zero, the computation for `currentStepAmount` and `amount` variables 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 setSalePrice(uint256 _value, uint256 _valuenext) external
onlyOwner {
        salePrice = _value;
        nextPrice = _valuenext;
    }
    ...
    currentStepAmount =
        (hardcapsizeUSD * (10**18) - totalSoldUSD) /
        salePrice;
    ...
    amount = usdPrice / salePrice;
```

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.

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1255
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.

ERC20 tokens can have varying decimal precision, typically ranging from 0 to 18, but not strictly limited to these. Relying on fixed decimal calculations instead of the actual decimal precision defined in the token's contract can lead to rounding errors, unintended behaviors, and in some cases, financial discrepancies.

```
        uint256 totalSoldUSD = (totalUsdValueForPresale * (10**18)) -

         currentStepAmount =
                     (hardcapsizeUSD * (10**18) - totalSoldUSD) /
                     salePrice;

         return (hardcapsizeUSD * (10**18) - totalSoldUSD);
             inSaleUSDvalue;

         uint256 ETHAmount = (usdPrice * (10**18)) / getETHLatestPrice();
```

Recommendation

To avoid these issues, it is important to carefully review the implementation of the decimals field of the underlying tokens.It is recommended to retrieve and utilize the decimals() function from the ERC20 token's interface to determine the correct decimal precision for calculations. 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.

ST - Stops Transactions

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1141
Status	Unresolved

Description

The contract owner has the authority to stop the presale buys for all users including the owner.

```
    function pause() external onlyOwner {
        _pause();
        isPresalePaused = true;
    }

    function unpause() external onlyOwner  {
        _unpause();
        isPresalePaused = false;
    }
```

Recommendation

The team could consider to remove the `pause` functionality from the contract if it doesn't play a crucial role. Additionally, 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. Some suggestions are:

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-sign wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.
- Renouncing the ownership will eliminate the threats but it is non-reversible.

PRDI - Potential Reward Decimal Inconsistency

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1346
Status	Unresolved

Description

The contract contains the `claim` function that allows users to retrieve their `saleToken` rewards. However, an inherent assumption in the function is that the `saleToken` has the same decimal precision as the tokens in `userDeposits`. This assumption can lead to discrepancies in the amount of `saleToken` transferred to users, if the `saleToken` and the tokens in `userDeposits` 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 whenNotPaused {
        require(saleToken != address(0), "Sale token not added");
        require(block.timestamp >= claimStart, "Claim has not started
yet");
        require(!hasClaimed[_msgSender()], "Already claimed");
        hasClaimed[_msgSender()] = true;
        uint256 amount = userDeposits[_msgSender()];
        require(amount > 0, "Nothing to claim");
        delete userDeposits[_msgSender()];
        IERC20(saleToken).transfer(_msgSender(), amount);
        emit TokensClaimed(_msgSender(), amount, block.timestamp);
    }
```

Recommendation

It is recommended to explicitly handle the decimal differences between the `saleToken` and the tokens in `userDeposits`. 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 `saleToken` proportional to their deposits, regardless of the decimal differences between the tokens.

RVA - Redundant Variable Assignment

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1262
Status	Unresolved

Description

The contract contains the variable `usdPrice` which is being set to its own value. This operation is redundant as it has no impact on the code's logic or the state of the contract. Such assignments can introduce confusion and make the code less readable for developers and auditors.

```
usdPrice = usdPrice;
```

Recommendation

It is recommended to remove the redundant assignment of the variable. Cleaning up such unnecessary operations will not only make the codebase cleaner and more readable but also reduces the overall gas consumption for any transaction invoking this part of the code.

RCC - Redundant Conditional Check

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1262
Status	Unresolved

Description

The contract contains a conditional check which check whether `purchaseToken` is 0 or 1. However, no specific actions or logic are executed within the body of the 'if' statement. This creates unnecessary complexity and can lead to confusion when interpreting the contract's behavior.

```
        if (purchaseToken == 0 || purchaseToken == 1) usdPrice =
usdPrice; //USDT and USDC have 6 decimals
```

Recommendation

It is recommended to remove the redundant conditional check. This will help in improving the readability and efficiency of the contract while reducing potential points of failure or misinterpretation.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1399
Status	Unresolved

Description

The contract owner has the authority to claim all the balance of the contract. The owner may take advantage of it by calling the `withdrawTokens` function.

```
   function withdrawTokens(address token, uint256 amount) external
onlyOwner   {
        IERC20(token).transfer(dAddress, amount);
    }
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions. Some suggestions are:

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-sign wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.
- Renouncing the ownership will eliminate the threats but it is non-reversible.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1128
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The contract updates the values of `salePrice` and `nextPrice` varriables even if their current state is the same as the the one passed as an argument. As a result, the contract performs redundant storage writes.

```
   function setSalePrice(uint256 _value, uint256 _valuenext) external
onlyOwner {
        salePrice = _value;
        nextPrice = _valuenext;
    }
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1128,1386,1394
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 setSalePrice(uint256 _value, uint256 _valuenext) external
onlyOwner {
        salePrice = _value;
        nextPrice = _valuenext;
    }

    function changehardcapSize(uint256 _hardcapSize) external onlyOwner
{
        require(
            _hardcapSize > 0 && _hardcapSize != hardcapSize,
            "Invalid hardcapSize size"
        );
        hardcapSize = _hardcapSize;
    }

    function changeMinimumBuyAmount(uint256 _amount) external onlyOwner
{
        require(_amount > 0 && _amount != minimumBuyAmount, "Invalid
amount");
        minimumBuyAmount = _amount;
    }
```

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.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1062,1064,1072,1074,1075
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
uniswapV2Router
baseDecimals
dataOracle
routerAddress
USDTtoken
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L659,878,880,911,1015,1037,1104,1108,1112,1116,1122,1128,1133,1151,1327,1329,1358,1365,1382,1386,1394
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 WETH() external pure returns (address);
function DOMAIN_SEPARATOR() external view returns (bytes32);
function PERMIT_TYPEHASH() external pure returns (bytes32);
function MINIMUM_LIQUIDITY() external pure returns (uint256);

...
```

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.

L06 - Missing Events Access Control

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1383
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task. There are functions that have no event emitted, so it is difficult to track off-chain changes.

```
dAddress = _dAddress
```

Recommendation

To avoid this issue, it's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1118,1124,1337,1362
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
totalUsdValueForPresale = totalUsdValueForPresale + _valuetotal
totalUsdValueForPresale = totalUsdValueForPresale - _valuetotal
claimStart = _claimStart
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L75,335,359,390,403,422,442,466,485,502,520,537,1184
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 isContract(address account) internal view returns (bool) {
        // This method relies on extcodesize/address.code.length, which
returns 0
        // for contracts in construction, since the code is only stored
at the end
        // of the constructor execution.

        return account.code.length > 0;
    }

...
```

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.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1170,1180
Status	Unresolved

Description

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

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

```
require(isPresalePaused != true, "presale paused")
```

Recommendation

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

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1166,1171,1235,1242,1270,1277
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.

```
amount = usdPrice / salePrice
userDeposits[_msgSender()] += (amount * (10**18))
```

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	contracts/AiGold_PreSale.sol#L1278
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.

```
IERC20 tokenInterface
```

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	contracts/AiGold_PreSale.sol#L1383
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.

```
dAddress = _dAddress
```

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	contracts/AiGold_PreSale.sol#L549
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	contracts/AiGold_PreSale.sol#L5,84,110,198,303,564,654
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 ^0.8.0;
pragma solidity ^0.8.1;
pragma solidity ^0.8.6;
```

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	contracts/AiGold_PreSale.sol#L5,84,110,198,303,564,654
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;
pragma solidity ^0.8.1;
pragma solidity ^0.8.6;
```

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.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	contracts/AiGold_PreSale.sol#L1339,1354,1400
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
IERC20(_saleToken).transferFrom(
            _msgSender(),
            address(this),
            tokensAmount
        )
IERC20(saleToken).transfer(_msgSender(), amount)
IERC20(token).transfer(dAddress, amount)
```

Recommendation

The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
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	✓	

Pausable	Implementation	Context		
		Public	✓	-
	paused	Public		-
	_requireNotPaused	Internal		
	_requirePaused	Internal		
	_pause	Internal	✓	whenNotPaused
	_unpause	Internal	✓	whenPaused
Address	Library			
	isContract	Internal		
	sendValue	Internal	✓	
	functionCall	Internal	✓	
	functionCall	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionStaticCall	Internal		
	functionStaticCall	Internal		
	functionDelegateCall	Internal	✓	
	functionDelegateCall	Internal	✓	
	verifyCallResult	Internal		
IERC20	Interface			
	totalSupply	External		-

	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
IUniswapV2Router01	Interface			
	factory	External		-
	WETH	External		-
	addLiquidity	External	✓	-
	addLiquidityETH	External	Payable	-
	removeLiquidity	External	✓	-
	removeLiquidityETH	External	✓	-
	removeLiquidityWithPermit	External	✓	-
	removeLiquidityETHWithPermit	External	✓	-
	swapExactTokensForTokens	External	✓	-
	swapTokensForExactTokens	External	✓	-
	swapExactETHForTokens	External	Payable	-
	swapTokensForExactETH	External	✓	-
	swapExactTokensForETH	External	✓	-
	swapETHForExactTokens	External	Payable	-
	quote	External		-
	getAmountOut	External		-

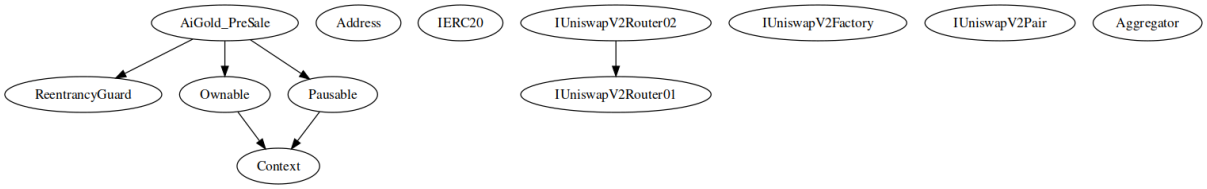
	getAmountIn	External		-
	getAmountsOut	External		-
	getAmountsIn	External		-
IUniswapV2Factory	Interface			
	feeTo	External		-
	feeToSetter	External		-
	getPair	External		-
	allPairs	External		-
	allPairsLength	External		-
	createPair	External	✓	-
	setFeeTo	External	✓	-
	setFeeToSetter	External	✓	-
IUniswapV2Pair	Interface			
	name	External		-
	symbol	External		-
	decimals	External		-
	totalSupply	External		-
	balanceOf	External		-
	allowance	External		-
	approve	External	✓	-
	transfer	External	✓	-

	transferFrom	External	✓	-
	DOMAIN_SEPARATOR	External		-
	PERMIT_TYPEHASH	External		-
	nonces	External		-
	permit	External	✓	-
	MINIMUM_LIQUIDITY	External		-
	factory	External		-
	token0	External		-
	token1	External		-
	getReserves	External		-
	price0CumulativeLast	External		-
	price1CumulativeLast	External		-
	kLast	External		-
	mint	External	✓	-
	burn	External	✓	-
	swap	External	✓	-
	skim	External	✓	-
	sync	External	✓	-
	initialize	External	✓	-
IUniswapV2Router02	Interface	IUniswapV2Router01		
	removeLiquidityETHSupportingFeeOnTransferTokens	External	✓	-
	removeLiquidityETHWithPermitSupportingFeeOnTransferTokens	External	✓	-

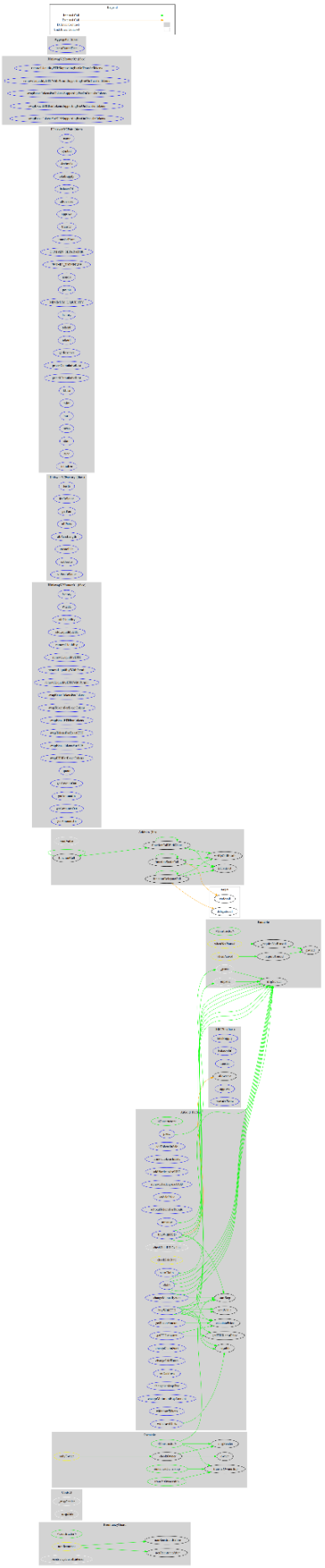
	swapExactTokensForTokensSupportingFeeOnTransferTokens	External	✓	-
	swapExactETHForTokensSupportingFeeOnTransferTokens	External	Payable	-
	swapExactTokensForETHSupportingFeeOnTransferTokens	External	✓	-
Aggregator	Interface			
	latestRoundData	External		-
AiGold_PreSale	Implementation	ReentrancyGuard, Ownable, Pausable		
		Public	✓	-
	startStep	Internal	✓	
	changeManuallyStep	External	✓	onlyOwner
	addTokensInSale	External	✓	onlyOwner
	removeTokensInSale	External	✓	onlyOwner
	addHardcapsizeUSD	External	✓	onlyOwner
	removeHardcapsizeUSD	External	✓	onlyOwner
	setSalePrice	External	✓	onlyOwner
	settotalTokensForPresale	External	✓	onlyOwner
	pause	External	✓	onlyOwner
	unpause	External	✓	onlyOwner
	calculatePrice	Internal		
	checkSoldUSDvalue	Internal		
	getETHLatestPrice	Public		-

	sendValue	Internal	✓	
	buyWithETH	External	Payable	checkSaleState whenNotPaused nonReentrant
	buyWithUSD	External	✓	checkSaleState whenNotPaused
	getETHAmount	External		-
	getTokenAmount	External		-
	startClaim	External	✓	onlyOwner
	claim	External	✓	whenNotPaused
	changeClaimStart	External	✓	onlyOwner
	changeSaleTimes	External	✓	onlyOwner
	setDaddress	External	✓	onlyOwner
	changehardcapSize	External	✓	onlyOwner
	changeMinimumBuyAmount	External	✓	onlyOwner
	withdrawTokens	External	✓	onlyOwner
	withdrawETHs	External	✓	onlyOwner

Inheritance Graph



Flow Graph



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