Cyberscope

Audit Report

CloudBTCPresale

February 2024

SHA256 916b063e737d062f872616afc4c6acfdeed9e9efb90cde6ca931017711392508

Audited by © cyberscope

Table of Contents

Review

Testing Deploy	https://goerli.etherscan.io/address/0x9db907f7fa55c54854368fce0d00dd9372bdbc01

Audit Updates

Initial Audit	12 Feb 2024 https://github.com/cyberscope-io/audits/blob/main/cbtc/v1/audit.pdf
Corrected Phase 2	16 Feb 2024

Source Files

Filename	SHA256
contracts/CloudBTCPresale.sol	916b063e737d062f872616afc4c6acfdeed9e9efb90cde6ca931017711392508
@openzeppelin/contracts/utils/Context.sol	9c1cc43aa4a2bde5c7dea0d4830cd42c54813ff883e55c8d8f12e6189bf7f10a
@openzeppelin/contracts/utils/math/SafeMath.sol	6f78c3ed573a960297585ff30b926477aa2ac81297ecae48727de5102c9f7742
@openzeppelin/contracts/token/ERC20/IERC20.sol	6f2faae462e286e24e091d7718575179644dc60e79936ef0c92e2d1ab3ca3cee
@openzeppelin/contracts/security/ReentrancyGuard.sol	265194fe7821f95d4c3cf2c6c8ad3a86313a2df90e8ee3385e454abbb8c233f6
@openzeppelin/contracts/security/Pausable.sol	5a5498e08dcbab736f3399f8115629eb818b6a7171d83430aef3bc146630d16b
@openzeppelin/contracts/access/Ownable.sol	38578bd71c0a909840e67202db527cc6b4e6b437e0f39f0c909da32c1e30cb81

@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol	62d0c0c753b724d3450723960ca4d256a16613a8c50ca42755610a951b772c7d

Overview

The contract facilitates a presale event for a specific token, allowing participants to purchase tokens using either Ethereum (ETH) or Tether (USDT) as payment methods. The presale is divided into 9 stages, each with a predefined token supply and price, ensuring a gradual and strategic token distribution. As the presale progresses through these stages, the contract adjusts the token price and supply available for purchase, automatically transitioning to the next stage once the current stage's token allocation is fully sold. This design aims to incentivize early participation and manage the distribution pace effectively.

buyWithETH functionality

The `buyWithETH` functionality within the contract allows participants to purchase tokens using Ethereum (ETH) during a presale event, subject to minimum and maximum investment limits of 0.001 and 1 Ether. Upon sending ETH to the contract, a portion may be allocated as a referral reward if a valid referral is provided. The remaining ETH, after deducting the referral reward, is distributed among predefined wallets. The contract converts the ETH sent into its USDT equivalent based on the current ETH price, then calculates the number of tokens the participant can purchase at the current stage's price. If the purchase exceeds the available token supply for the current stage, the contract automatically advances to the next stage to fulfill the remaining order, ensuring a seamless transition and accurate token distribution. Finally, the `presaleToken` is transferred to the user, completing the purchase process.

buyWithUSDT functionality

The `buyWithUSDT` functionality enables participants to use Tether (USDT) for purchasing tokens during the presale event, adhering to set minimum and maximum transaction limits of 10 and 2500 USDT. Initially, the contract verifies that the sender has granted sufficient USDT allowance to the contract for the transaction. A portion of the USDT amount may be allocated as a referral reward if a valid referral is provided, fostering the growth of the presale's network. The remaining USDT, after subtracting the referral reward, is divided and sent to designated receiver addresses. The contract calculates the number of tokens purchasable based on the current stage's USDT price, ensuring participants receive the correct token amount for their USDT. Should the desired purchase exceed the available

supply for the current stage, the contract automatically progresses to the next stage, adjusting the token price accordingly to complete the transaction. This mechanism ensures a fair and orderly distribution of tokens across different presale stages, ultimately increasing the total USDT raised and the total tokens sold. Finally, the `presaleToken` is transferred to the participant, finalizing the purchase.

Roles

Owner

The owner can interact with the following functions:

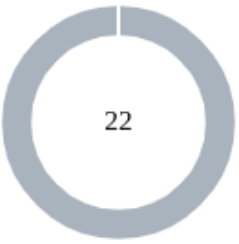
- function setPresaleStage
- function withdrawETH
- function withdrawUSDT
- function pausePresale
- function unpausePresale
- function updateReferralPercentage
- function distributeTokens
- function setStagePrice
- function setStageTokenSupply
- function setMinLimitETH
- function setMaxLimitETH
- function setMinLimitUSD
- function setMaxLimitUSD
- function setEthReceivers
- function setUsdtReceivers
- function setReferralPercentage
- function updateUsdtTokenAddress
- function updateEthPriceFeedAddress

Users

The users can interact with the following functions:

- function buyWithETH(address referral)
- function buyWithUSDT
- function getLatestETHPrice()

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	22

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	22	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	PBV	Percentage Boundaries Validation	Unresolved
	TSI	Tokens Sufficiency Insurance	Unresolved
	CCR	Contract Centralization Risk	Unresolved
	PTRP	Potential Transfer Revert Propagation	Unresolved
	DDP	Decimal Division Precision	Unresolved
	USS	Unvalidated Stage Setting	Unresolved
	ODM	Oracle Decimal Mismatch	Unresolved
	MPR	Misleading Price Representation	Unresolved
	SPM	Stage Price Mismatch	Unresolved
	DPI	Decimals Precision Inconsistency	Unresolved
	ALU	Array Length Usage	Unresolved
	RSW	Redundant Storage Writes	Unresolved
	MEE	Missing Events Emission	Unresolved
	MSC	Missing Sanity Check	Unresolved

●	RFU	Redundant Functionality Usage	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

PBV - Percentage Boundaries Validation

Criticality	Minor / Informative
Location	contracts/Presale.sol#L178,216,320
Status	Unresolved

Description

The contract is using the `referralReward` variable for calculations. However, this variable is used in multiplication operations and if `referralReward` is set to a value greater than `100`, it could lead to incorrect calculations, potentially causing unintended behavior or financial discrepancies within the contract's operations.

```
referralReward = msg.value.mul(referralPercentage).div(100); //
5% referral reward
...
referralReward = (usdtAmount *
(10**usdtDecimals)).mul(referralPercentage).div(100); // 5%
referral reward in USDT

...
    function updateReferralPercentage(uint256 val) public
onlyOwner {
        referralPercentage = val;
    }
```

Recommendation

It is recommended to ensure that the values of `claimPer` and `tgePer` cannot exceed `100`. This can be achieved by adding checks whenever these variables are set.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	contracts/Presale.sol#L130
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

```
   constructor(address _presaleTokenAddress, address
_usdtTokenAddress, address _ethPriceFeedAddress)
Ownable(msg.sender) {
        presaleToken = IERC20(_presaleTokenAddress);
    ...
    }
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the presale tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	contracts/Presale.sol#L285,312,316,325
Status	Unresolved

Description

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

Specifically, the owner has the ability to set the Presale Stage, pause and unpause the presale functionality, update the referral percentage reward, and distribute tokens to specific addresses. This concentration of power in the hands of a single entity introduces a significant centralization risk, potentially undermining the trust and security principles typically associated with decentralized systems. Such centralization may lead to scenarios where the contract owner could unilaterally make changes that affect all stakeholders, without consensus or oversight, potentially impacting the fairness and transparency of the presale and token distribution processes.

```
   function setPresaleStage(PresaleStage stage) public
onlyOwner {
        currentStage = stage;
        emit StageChanged(stage);
    }

    function pausePresale() public onlyOwner {
        _pause();
    }

    function unpausePresale() public onlyOwner {
        _unpause();
    }

    function updateReferralPercentage(uint256 val) public
onlyOwner {
        referralPercentage = val;
    }

    function distributeTokens(address[] calldata userAddresses,
uint256[] calldata tokenAmounts) external onlyOwner
nonReentrant whenNotPaused {
    ...
    }
```

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.

PTRP - Potential Transfer Revert Propagation

Criticality	Minor / Informative
Location	contracts/Presale.sol#L179
Status	Unresolved

Description

The contract sends funds to a `referral` address as part of the transfer flow. This address can either be a wallet address or a contract. If the address belongs to a contract then it may revert from incoming payment. As a result, the error will propagate to the token's contract and revert the transfer.

```
payable(referral).transfer(referralReward);
```

Recommendation

The contract should tolerate the potential revert from the underlying contracts when the interaction is part of the main transfer flow. This could be achieved by not allowing set contract addresses or by sending the funds in a non-revertable way.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	contracts/Presale.sol#L187,227,299,307
Status	Unresolved

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
        uint256 share = remainingETH.div(3);
         for (uint i = 0; i < ethReceivers.length; i++) {
             ethReceivers[i].transfer(share);
         }
    ...

    uint256 share = remainingUSDT.div(3);
         for (uint i = 0; i < usdtReceivers.length; i++) {
             usdtToken.transferFrom(msg.sender,
usdtReceivers[i], share);
         }

    ...
    function withdrawETH() external onlyOwner {
         uint256 balance = address(this).balance;
         uint256 share = balance.div(3);
         for (uint i = 0; i < ethReceivers.length; i++) {
             ethReceivers[i].transfer(share);
         }
    }
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

USS - Unvalidated Stage Setting

Criticality	Minor / Informative
Location	contracts/Presale.sol#L148,285
Status	Unresolved

Description

The contract contains the `setPresaleStage` function that allows the owner to set the current presale stage. However, this function lacks validation to ensure the stage value passed as a parameter is within accepted limits. Consequently, it is possible for an undefined or unintended presale stage to be set, potentially leading to unpredictable behavior or misuse of the contract functionality. This oversight introduces a risk to the contract's integrity and could affect the presale process's fairness and transparency.

```
    function initializeStageData() private {
        // Initialize stage prices in USDT and token supplies
        stagePrices[PresaleStage.Stage1] = 0.000800 ether;
        stagePrices[PresaleStage.Stage2] = 0.000880 ether;
        stagePrices[PresaleStage.Stage3] = 0.000968 ether;
        stagePrices[PresaleStage.Stage4] = 0.001065 ether;
        stagePrices[PresaleStage.Stage5] = 0.001171 ether;
        stagePrices[PresaleStage.Stage6] = 0.001288 ether;
        stagePrices[PresaleStage.Stage7] = 0.001417 ether;
        stagePrices[PresaleStage.Stage8] = 0.001559 ether;
        stagePrices[PresaleStage.Stage9] = 0.001715 ether;
        ...

    function setPresaleStage(PresaleStage stage) public
onlyOwner {
        currentStage = stage;
        emit StageChanged(stage);
    }
```

Recommendation

It is recommended to add an additional check within the `setPresaleStage` function to verify the state value before setting the new stage. This could involve explicitly checking that the passed `stage` parameter is within the range of defined `PresaleStage` enum values. Implementing such validation ensures that only valid stages can be set,

preventing any unintended contract states and enhancing the contract's security and reliability. This approach will safeguard the presale process against potential manipulation or errors stemming from invalid stage settings.

ODM - Oracle Decimal Mismatch

Criticality	Minor / Informative
Location	contracts/Presale.sol#L194
Status	Unresolved

Description

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

```
    function getLatestETHPrice() public view returns (uint256) {
        (,int price,,,) = ethPriceFeed.latestRoundData();
        require(price > 0, "Invalid ETH price");
        return uint256(price);
    }
```

Recommendation

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

MPR - Misleading Price Representation

Criticality	Minor / Informative
Location	contracts/Presale.sol#L195,232
Status	Unresolved

Description

The contract declares the `currentPriceInUSDT` variable, which, by its naming, suggests it stores the price of tokens in USDT. However, the `stagePrices` for each presale stage, which are used to set `currentPriceInUSDT`, are denominated in ether (ETH) rather than USDT. This discrepancy between the variable's name and the unit of currency used for the price values is misleading. It could cause confusion leading to incorrect assumptions about the contract's pricing mechanism and potentially impacting the contract's integration with external systems or its overall functionality.

```
        uint256 currentPriceInUSDT = stagePrices[currentStage];
// Price per token in USDT for current stage
```

Recommendation

It is recommended to ensure consistent and clear naming conventions and units of measurement throughout the contract. If `currentPriceInUSDT` is intended to represent prices in USDT, the stagePrices should be set in USDT units, not in ether. Alternatively, if the contract's logic is to remain as is, renaming `currentPriceInUSDT` to reflect that it is denominated in ether would help clarify the actual functionality.

SPM - Stage Price Mismatch

Criticality	Minor / Informative
Location	contracts/Presale.sol#L242,273
Status	Unresolved

Description

The contract contains the processPurchase function designed to handle the purchase of tokens. This function checks if the available tokens in the current stage are sufficient for the purchase. If not, it advances to the next stage using the checkAndAdvanceStage function. However, a critical oversight is observed where tokens from the next stage are sold at the price of the previous stage. This discrepancy allows users to purchase tokens at a lower price than intended for the new stage, potentially leading to financial discrepancies and undermining the presale's pricing structure. As a result, depending on the ETH price at the current time of buying the tokens, the user could buy tokens from the next round with the price of the previous round.

```
    function processPurchase(uint256 tokensToBuy, uint256
value, string memory currency) private {
        uint256 availableTokens =
(stageTokenSupplies[currentStage] * baseDecimals) -
tokensSold[currentStage];

        if (tokensToBuy <= availableTokens) {
            // Purchase can be fulfilled within the current
stage
            tokensSold[currentStage] += tokensToBuy;
            presaleToken.transfer(msg.sender, tokensToBuy);
            emit TokensPurchased(msg.sender, tokensToBuy,
value, currency);
        } else {
            // Purchase spans to the next stage
            require(currentStage < PresaleStage.Stage9, "Not
enough tokens available");

            // Fulfill part of the purchase with the current
stage's remaining tokens
            tokensSold[currentStage] =
stageTokenSupplies[currentStage] * baseDecimals;
            uint256 stageValue = availableTokens *
stagePrices[currentStage] / baseDecimals;
            presaleToken.transfer(msg.sender, availableTokens);
            emit TokensPurchased(msg.sender, availableTokens,
stageValue, currency);

            // Move to the next stage and fulfill the remaining
part of the purchase
            checkAndAdvanceStage();
            uint256 remainingTokensToBuy = tokensToBuy -
availableTokens;
            uint256 remainingValue = (value > stageValue) ?
value - stageValue : 0;
            tokensSold[currentStage] += remainingTokensToBuy;
            presaleToken.transfer(msg.sender,
remainingTokensToBuy);
            emit TokensPurchased(msg.sender,
remainingTokensToBuy, remainingValue, currency);
        }

        // Check and advance stage after processing the
purchase
        checkAndAdvanceStage();
    }

    function checkAndAdvanceStage() private {
        if (tokensSold[currentStage] >=
stageTokenSupplies[currentStage] * baseDecimals) {
```

```
            if (currentStage == PresaleStage.Stage9) {
                _pause();
                emit PresaleEnded();
            } else {
                currentStage =
PresaleStage(uint256(currentStage) + 1);
                emit PresaleAdvanced(currentStage);
            }
        }
    }
```

`

Recommendation

It is recommended to revise the `processPurchase` function to ensure that when advancing to the next stage, the remaining tokens are sold at the correct price of the new stage. This could involve recalculating the `remainingValue` based on the `stagePrices` of the newly advanced stage before proceeding with the sale of additional tokens. Implementing this change will align the token sale process with the intended pricing strategy across different stages, ensuring fairness and maintaining the integrity of the presale event.

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	contracts/Presale.sol#L193,198,233,243,256
Status	Unresolved

Description

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

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

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

```
uint256 usdtAmount = (msg.value * ethPriceInUSDT * (10 ** 10));
...
processPurchase(tokensToBuy, (usdtAmount / baseDecimals),
"ETH");
...
uint256 tokensToBuy = (adjustedUsdtAmount / currentPriceInUSDT)
* baseDecimals; // Direct division for token amount calculation
...
uint256 availableTokens = (stageTokenSupplies[currentStage] *
baseDecimals) - tokensSold[currentStage];
....
tokensSold[currentStage] = stageTokenSupplies[currentStage] *
baseDecimals;
uint256 stageValue = availableTokens *
stagePrices[currentStage] / baseDecimals;
```

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.

ALU - Array Length Usage

Criticality	Minor / Informative
Location	contracts/Presale.sol#L187,227,299,307
Status	Unresolved

Description

The contract is designed to distribute a balance among addresses stored in the `ethReceivers` and `usdtReceivers` arrays, which both are intended to contain three addresses. The distribution logic calculates each recipient's `share` by dividing the total value by a hardcoded value of 3. This division approach does not account for the actual length of the `ethReceivers` and `usdtReceivers` arrays. This oversight can result in either an excess or shortage of distributed funds, depending on the current length of the array, potentially causing financial discrepancies and undermining the fairness and reliability of the distribution mechanism.

```
uint256 share = remainingETH.div(3);
for (uint i = 0; i < ethReceivers.length; i++) {

...

uint256 share = remainingUSDT.div(3);
for (uint i = 0; i < usdtReceivers.length; i++) {
```

Recommendation

It is recommended to dynamically calculate the share amount by using the length of the `ethReceivers` and `usdtReceivers` arrays instead of a hardcoded value. This can be achieved by modifying the share calculation to `uint256 share = balance.div(ethReceivers.length);`. By adopting this approach, the contract ensures that the total balance is always evenly divided among the current number of addresses in the arrays, regardless of the length. This adjustment enhances the contract's flexibility and accuracy in fund distribution, aligning it with the principle of dynamic array handling in smart contract development.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	contracts/Presale.sol#L347,352
Status	Unresolved

Description

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

```
   function setStagePrice(PresaleStage stage, uint256 price)
public onlyOwner {
        stagePrices[stage] = price;
    }

    // Function to update stage token supplies
    function setStageTokenSupply(PresaleStage stage, uint256
supply) public onlyOwner {
        stageTokenSupplies[stage] = supply;
    }
```

Recommendation

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

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	contracts/Presale.sol#L346,357
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 to update stage prices
    function setStagePrice(PresaleStage stage, uint256 price)
public onlyOwner {
        stagePrices[stage] = price;
    }

    // Function to update stage token supplies
    function setStageTokenSupply(PresaleStage stage, uint256
supply) public onlyOwner {
        stageTokenSupplies[stage] = supply;
    }

    // Functions to update min and max limits for ETH and USD
    function setMinLimitETH(uint256 _minLimitETH) public
onlyOwner {
        minLimitETH = _minLimitETH;
    }

    function setMaxLimitETH(uint256 _maxLimitETH) public
onlyOwner {
        maxLimitETH = _maxLimitETH;
    }

    function setMinLimitUSD(uint256 _minLimitUSD) public
onlyOwner {
        minLimitUSD = _minLimitUSD;
    }

    function setMaxLimitUSD(uint256 _maxLimitUSD) public
onlyOwner {
        maxLimitUSD = _maxLimitUSD;
    }
```

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.

MSC - Missing Sanity Check

Criticality	Minor / Informative
Location	contracts/Presale.sol#L356,365
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.

Specifically, the contract is designed to allow the contract owner to dynamically adjust the minimum and maximum limits for both ETH and USD. This functionality is provided through four distinct functions, the `setMinLimitETH`, `setMaxLimitETH`, `setMinLimitUSD`, and `setMaxLimitUSD`. However, a critical oversight in the contract's design is the absence of validation checks to ensure that the minimum limit values set for both ETH and USD are strictly less than their respective maximum limit values. This lack of validation could potentially allow for the configuration of a minimum limit that is greater than or equal to the maximum limit, leading to logical inconsistencies and preventing users from performing transactions within the intended limits.

```
    // Functions to update min and max limits for ETH and USD
    function setMinLimitETH(uint256 _minLimitETH) public
onlyOwner {
        minLimitETH = _minLimitETH;
    }

    function setMaxLimitETH(uint256 _maxLimitETH) public
onlyOwner {
        maxLimitETH = _maxLimitETH;
    }

    function setMinLimitUSD(uint256 _minLimitUSD) public
onlyOwner {
        minLimitUSD = _minLimitUSD;
    }

    function setMaxLimitUSD(uint256 _maxLimitUSD) public
onlyOwner {
        maxLimitUSD = _maxLimitUSD;
```

```
    }
```

Recommendation

It is recommended to introduce additional validation logic within the `setMinLimitETH`, `setMaxLimitETH`, `setMinLimitUSD`, and `setMaxLimitUSD` functions to verify that the new minimum limit values being set are less than the current maximum limit values. Similarly, when setting new maximum limit values, it should be verified that they are greater than the current minimum limit values. This can be achieved by adding require statements that compare the new minimum values against the existing maximum values and vice versa, ensuring that the contract enforces the logical constraint that minimum values must always be less than maximum values. Implementing this recommendation will enhance the contract's robustness and prevent potential operational issues related to limit configurations.

RFU - Redundant Functionality Usage

Criticality	Minor / Informative
Location	contracts/Presale.sol#L320,383
Status	Unresolved

Description

The contract is found to contain two separate functions, `updateReferralPercentage` and `setReferralPercentage`, that both serve the identical purpose of updating the referralPercentage variable. This redundancy not only makes the contract code less efficient but also introduces potential confusion for contract maintainers or developers interacting with the contract, as it is unclear which function should be used for updating the referral percentage. Both functions are marked as public and restricted to the contract owner, indicating that their intended use is for contract administration. However, the presence of two functions for the same action does not adhere to best practices in contract development, which emphasize code simplicity and clarity.

```
   function updateReferralPercentage(uint256 val) public
onlyOwner {
        referralPercentage = val;
    }
    // Function to update referral percentage
    function setReferralPercentage(uint256 _referralPercentage)
public onlyOwner {
        referralPercentage = _referralPercentage;
    }
```

Recommendation

It is recommended to remove one of the functions to streamline the contract's functionality and reduce potential confusion. Consolidating the functionality into a single function will make the contract more maintainable and easier to understand for other developers or auditors. This can be achieved by determining which of the two functions is preferred based on naming conventions, usage patterns, or other criteria, and then removing the other.

Additionally, ensuring that the remaining function has clear documentation on its purpose and usage will further enhance the contract's clarity and ease of use.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	contracts/CloudBTCPresale.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/v0.8.16/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	contracts/CloudBTCPresale.sol#L136
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.

```
baseDecimals
```

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/CloudBTCPresale.sol#L357,361,365,369,374,379,384,389,395
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.

```
uint256 _minLimitETH
uint256 _maxLimitETH
uint256 _minLimitUSD
uint256 _maxLimitUSD
address payable[3] memory _ethReceivers
address[3] memory _usdtReceivers
uint256 _referralPercentage
address _newUsdtTokenAddress
address _newEthPriceFeedAddress
```

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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/CloudBTCPresale.sol#L233
Status	Unresolved

Description

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

```
uint256 tokensToBuy = (adjustedUsdtAmount / currentPriceInUSDT)
* baseDecimals
```

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/CloudBTCPresale.sol#L228
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.

```
bool success
```

Recommendation

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

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	contracts/CloudBTCPresale.sol#L2,11
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 ^0.8.20;
```

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/CloudBTCPresale.sol#L248,257,265,308,338
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.

```
presaleToken.transfer(msg.sender, tokensToBuy)
presaleToken.transfer(msg.sender, availableTokens)
presaleToken.transfer(msg.sender, remainingTokensToBuy)
usdtToken.transfer(usdtReceivers[i], share)
presaleToken.transfer(userAddress, tokensToBuy)
```

Recommendation

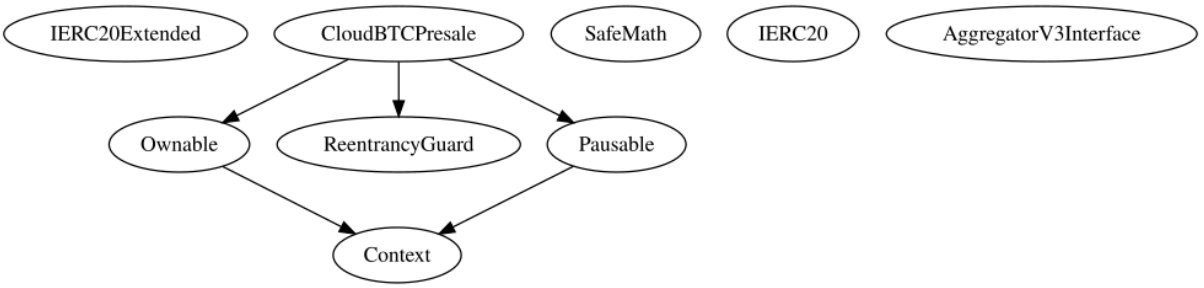
The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

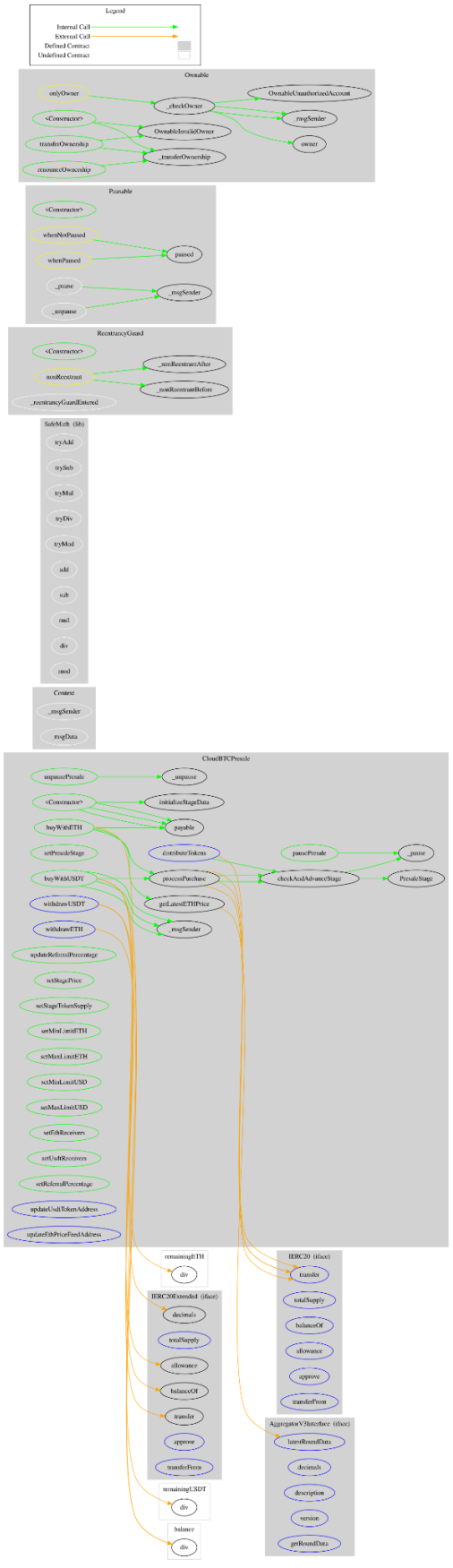
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IERC20Extended	Interface			
	decimals	External		-
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
CloudBTCPresale	Implementation	Ownable, ReentrancyGuard, Pausable		
		Public	✓	Ownable
	initializeStageData	Private	✓	
	buyWithETH	Public	Payable	nonReentrant whenNotPaused
	buyWithUSDT	Public	✓	nonReentrant whenNotPaused
	processPurchase	Private	✓	

	checkAndAdvanceStage	Private	✓	
	setPresaleStage	Public	✓	onlyOwner
	getLatestETHPrice	Public		-
	withdrawETH	External	✓	onlyOwner
	withdrawUSDT	External	✓	onlyOwner
	pausePresale	Public	✓	onlyOwner
	unpausePresale	Public	✓	onlyOwner
	updateReferralPercentage	Public	✓	onlyOwner
	distributeTokens	External	✓	onlyOwner nonReentrant whenNotPaused
	setStagePrice	Public	✓	onlyOwner
	setStageTokenSupply	Public	✓	onlyOwner
	setMinLimitETH	Public	✓	onlyOwner
	setMaxLimitETH	Public	✓	onlyOwner
	setMinLimitUSD	Public	✓	onlyOwner
	setMaxLimitUSD	Public	✓	onlyOwner
	setEthReceivers	Public	✓	onlyOwner
	setUsdtReceivers	Public	✓	onlyOwner
	setReferralPercentage	Public	✓	onlyOwner
	updateUsdtTokenAddress	External	✓	onlyOwner
	updateEthPriceFeedAddress	External	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Ownable
onlyOwner
_checkOwner
OwnableUnauthorizedAccount
<Constructor>
_msgSender
OwnableInvalidOwner
transferOwnership
owner
_transferOwnership
renounceOwnership
Pausable
<Constructor>
whenNotPaused
paused
whenPaused
_pause
_msgSender
_unpause
ReentrancyGuard
<Constructor>
nonReentrant
_nonReentrantAfter
_nonReentrantBefore
_reentrancyGuardEntered
SafeMath (lib)
tryAdd
trySub
tryMul
tryDiv
tryMod
add
sub
mul
div
mod
Context
_msgSender
_msgData
CloudBTCPresale
unpausePresale
_unpause
<Constructor>
initializeStageData
buyWithETH
payable
setPresaleStage
distributeTokens
pausePresale
_pause
buyWithUSDT
processPurchase
checkAndAdvanceStage
PresaleStage
withdrawUSDT
getLatestETHPrice
withdrawETH
_msgSender
updateReferralPercentage
setStagePrice
setStageTokenSupply
setMinLimitETH
setMaxLimitETH
setMinLimitUSD
setMaxLimitUSD
setEthReceivers
setUsdtReceivers
setReferralPercentage
updateUsdtTokenAddress
updateEthPriceFeedAddress
remainingETH
div
IERC20Extended (iface)
decimals
totalSupply
allowance
balanceOf
transfer
approve
transferFrom
remainingUSDT
div
balance
div
IERC20 (iface)
transfer
totalSupply
balanceOf
allowance
approve
transferFrom
AggregatorV3Interface (iface)
latestRoundData
decimals
description
version
getRoundData

Summary

CloudBTC contract implements a presale mechanism allowing participants to purchase tokens using either Ethereum (ETH) or Tether (USDT). 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