Cyberscope

Audit Report

EBM Presale

February 2025

Network	BSC
Address	0x61Ba8A1A403441d450dc7cE95757F57f3E3BA237
Audited by	© cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope’s smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Review

Explorer	https://bscscan.com/address/0x61ba8a1a403441d450dc7ce95757f57f3e3ba237

Audit Updates

Initial Audit	06 Feb 2025

Source Files

Filename	SHA256
contracts/EBMPresale.sol	c0f384df4b7a15fcce7bd1e825e8bb0883f6818e0585bd3c1f4f7fa8f8698ce0

Overview

EBMPresale Contract

The EBMPresale contract is a token presale contract that allows users to purchase tokens using BNB, USDT, or USDC at different stages. It manages stages of token sales, each with a different price, supply, and bonus/referral structure. The contract also handles referral rewards, bonus distributions, and stage transitions based on sales targets.

Features

- Multiple sale stages with predefined token prices and targets.
- Purchasing functions for users to buy tokens using BNB, USDT, or USDC.
- Referral system, where referrers receive a percentage of the buyer’s payment.
- Bonus system, where buyers get extra tokens based on their contribution.
- Automatic stage transitions when targets are met.
- Admin controls for managing sale parameters and unsold tokens.

Roles

Admin

The admin can interact with the following functions:

- setAmountSpentInUsd(uint256 _newamount)
- getUnsoldTokens()

Users

users can interact with the following functions:

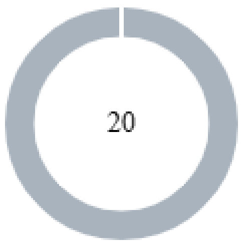
- buyTokensWithBNB(address referrer)
- buyTokensWithUSDT(uint256 usdtAmount, address referrer)
- buyTokensWithUSDC(uint256 usdcAmount, address referrer)

Retrieval Functions

The following functions can be used to retrieve information:

- getBonusAmountBNB(uint256 bnbAmount)
- getBonusAmountUSDT(uint256 usdtAmount)
- getBonusAmountUSDC(uint256 usdcAmount)
- getTokenAmountBNB(uint256 amountBNB)
- getTokenAmountUSDT(uint256 amountUSDT)
- getTokenAmountUSDC(uint256 amountUSDC)
- getLatestPriceBNBPerUSD()
- getPriceInUSD()
- getTotalSaleAmounts()
- getTokenSolds()
- getRemainingTokens()
- getReferralCount(address referrer)
- getReferralAmount(address referrer, Currency currency)

Findings Breakdown



●	Critical	0
●	Medium	0
●	Minor / Informative	20

Severity	Unresolved	Acknowledged	Resolved	Other
● Critical	0	0	0	0
● Medium	0	0	0	0
● Minor / Informative	0	20	0	0

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
⚪	PCALS	Presale Continues After Last Stage	Acknowledged
⚪	CR	Code Repetition	Acknowledged
⚪	CCR	Contract Centralization Risk	Acknowledged
⚪	DDP	Decimal Division Precision	Acknowledged
⚪	IDI	Immutable Declaration Improvement	Acknowledged
⚪	ICS	Ineffective Conditional Statement	Acknowledged
⚪	MVN	Misleading Variables Naming	Acknowledged
⚪	MC	Missing Check	Acknowledged
⚪	ODM	Oracle Decimal Mismatch	Acknowledged
⚪	PLTE	Potential Liquidity Timing Error	Acknowledged
⚪	POSD	Potential Oracle Stale Data	Acknowledged
⚪	PTRP	Potential Transfer Revert Propagation	Acknowledged

●	RC	Redundant Calculations	Acknowledged
●	RCS	Redundant Conditional Statements	Acknowledged
●	RC	Repetitive Calculations	Acknowledged
●	TSI	Tokens Sufficiency Insurance	Acknowledged
●	OCTD	Transfers Contract's Tokens	Acknowledged
●	L02	State Variables could be Declared Constant	Acknowledged
●	L04	Conformance to Solidity Naming Conventions	Acknowledged
●	L13	Divide before Multiply Operation	Acknowledged

PCALS - Presale Continues After Last Stage

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L171,231,288
Status	Acknowledged

Description

The `buyTokens` functions are missing a check if the last stage is completed. This is because if the targeted amount of USD is reached the total supply of the last stage may still have some tokens left, meaning that users may still be able to call the `buyTokens` functions until the total supply of the last stage is empty. This will also emit `StageCompleted` event more times than intended.

```
function buyTokensWithUSDC(uint256 usdcAmount, address
referrer) external nonReentrant {}

function getBonusAmountBNB(uint256 bnbAmount) public view
returns (uint256) {}

function getBonusAmountUSDC(uint256 usdcAmount) public view
returns (uint256) {}
```

Recommendation

It is recommended that each of these functions check if the last stage is completed to protect the contract from unintentional purchase of tokens and event emissions.

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L231,288,362,378,400,405
Status	Acknowledged

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.

```
//buyTokens
function buyTokensWithUSDT(uint256 usdtAmount, address
referrer) external nonReentrant {}
function buyTokensWithUSDC(uint256 usdcAmount, address
referrer) external nonReentrant {}

//getBonus
function getBonusAmountUSDT(uint256 usdtAmount) public view
returns (uint256) {}
function getBonusAmountUSDC(uint256 usdcAmount) public view
returns (uint256) {}

//getTokenAmount
function getTokenAmountUSDT(uint256 amountUSDT) public view
returns (uint256) {}
function getTokenAmountUSDC(uint256 amountUSDC) public view
returns (uint256) {}
```

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.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L416,452
Status	Acknowledged

Description

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

For example the tokens for the presale are not transferred at the start of the presale inside the contract and also the admin can retrieve them whenever they want to.

Another example is that if `amountSpentInUsd` is set to very high number the `getBonusAmountBNB` , `getBonusAmountUSDT` and `getBonusAmountUSDC` will always return 0.

```
function setAmountSpentInUsd(uint256 _newamount) external
onlyAdmin {}

function getUnsoldTokens() external onlyAdmin {}
```

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.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L171,231,288
Status	Acknowledged

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 remainingAmount = usdcAmount - (usdcAmount *
current.referralPercent) / 100;
usdcToken.safeTransferFrom(msg.sender, ICO_WALLET,
remainingAmount);
//...
uint256 referrerBonus = (usdcAmount * current.referralPercent)
/ 100;
usdcToken.safeTransferFrom(msg.sender, referrer,
referrerBonus);
```

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.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L156
Status	Acknowledged

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.

```
address admin;
```

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.

ICS - Ineffective Conditional Statement

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L203
Status	Acknowledged

Description

The `buyTokens` functions take as a parameter a `referrer` address. Then it is checked if the referrer is the `msg.sender` and if it is, the referrer is chnaged to the `MARKETING_WALLET` address. This is used to not allow the `msg.sender` to get `referrerBonus` however the user can add is input another address that they own making this if statement ineffective.

```
if (referrer == address(0) || referrer == msg.sender) {
    referrer = MARKETING_WALLET;
}
```

Recommendation

The team is advised to consider adding another method of verifying the referrer like if they already participated in the sale.

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L36
Status	Acknowledged

Description

Variables can have misleading names if their names do not accurately reflect the value they contain or the purpose they serve. The contract uses some variable names that are too generic or do not clearly convey the information stored in the variable. Misleading variable names can lead to confusion, making the code more difficult to read and understand.

In the `Stage struct` there is the `tokenSolds` property that keeps track of the total of tokens sold in the Stage. However along with the actual tokens sold, the bonus tokens are also added in the total. The bonus tokens are practically not sold tokens.

```
struct Stage {
    //...
    uint256 tokenSolds;
    //...
}
```

Recommendation

It's always a good practice for the contract to contain variable names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

MC - Missing Check

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L452
Status	Acknowledged

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.

In the function `getUnsoldTokens` it is not checked if the stages of the presale are finished in order for the `AIRDROP_WALLET` to retrieve the remaining tokens.

```
function getUnsoldTokens() external onlyAdmin {
    uint256 remainingTokens = token.balanceOf(address(this));
    if (remainingTokens == 0) revert NoUnsoldTokensLeft();
    token.safeTransfer(AIRDROP_WALLET, remainingTokens);

    emit UnsoldTokensTransferred(AIRDROP_WALLET,
remainingTokens);
}
```

Recommendation

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

ODM - Oracle Decimal Mismatch

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L412
Status	Acknowledged

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 getLatestPriceBNBPerUSD() public view returns
(uint256) {
     (, int256 price, , , ) = priceFeedBNB.latestRoundData();
    price = (price * (10 ** 10));
    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.

PLTE - Potential Liquidity Timing Error

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L171,231,288
Status	Acknowledged

Description

Users acquire tokens immediately after purchase, allowing them to add liquidity to the pair contract before the presale concludes. This can lead to a liquidity rate that differs from the intended value.

```
token.safeTransfer(msg.sender, totalTokens);
```

Recommendation

It is recommended that the contract holds the tokens of the presale until it ends and users can claim them afterwards. This will allow for the addition of liquidity in the intended value.

POSD - Potential Oracle Stale Data

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L410
Status	Acknowledged

Description

The contract relies on retrieving price data from an oracle. However, it lacks proper checks to ensure the data is not stale. The absence of these checks can result in outdated price data being trusted, potentially leading to significant financial inaccuracies.

```
function getLatestPriceBNBPerUSD() public view returns
(uint256) {
    (, int256 price, , , ) = priceFeedBNB.latestRoundData();
    price = (price * (10 ** 10));
    return uint256(price);
}
```

Recommendation

To mitigate the risk of using stale data, it is recommended to implement checks on the round and period values returned by the oracle's data retrieval function. The value indicating the most recent round or version of the data should confirm that the data is current. Additionally, the time at which the data was last updated should be checked against the current interval to ensure the data is fresh. For example, consider defining a threshold value, where if the difference between the current period and the data's last update period exceeds this threshold, the data should be considered stale and discarded, raising an appropriate error.

For contracts deployed on Layer-2 solutions, an additional check should be added to verify the sequencer's uptime. This involves integrating a boolean check to confirm the sequencer is operational before utilizing oracle data. This ensures that during sequencer downtimes, any transactions relying on oracle data are reverted, preventing the use of outdated and potentially harmful data.

By incorporating these checks, the smart contract can ensure the reliability and accuracy of the price data it uses, safeguarding against potential financial discrepancies and enhancing overall security.

PTRP - Potential Transfer Revert Propagation

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L194
Status	Acknowledged

Description

The contract sends funds to `ICO_WALLET` as part of the transfer flow. This address is a proxy contract. This may result in reverting from incoming payment. As a result, the error will propagate to the token's contract and revert the transfer.

```
Address.sendValue(payable(ICO_WALLET), remainingAmount);
```

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.

RC - Redundant Calculations

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L352,368,384
Status	Acknowledged

Description

The struct `Stage` has a property called `bonusPercent` that is used as a multiplier to calculate the `calculatedBonus`. However in all the different stages of the presale the calculated bonus is equal to 1. Therefore these calculations are redundant.

```
uint256 bonusPercent = hundredDollarIncrements *
current.bonusPercent;
```

Recommendation

The team is advised to remove the redundant calculations as they increase gas costs and reduce code readability.

RCS - Redundant Conditional Statements

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L207
Status	Acknowledged

Description

The contract contains redundant conditional statements that can be simplified or be removed to improve code efficiency and performance. Conditional statements that merely return the result of an expression are unnecessary and lead to larger code size, increased memory usage, and slower execution times. By directly returning the result of the expression, the code can be made more concise and efficient, reducing gas costs and improving runtime performance. Such redundancies are common when simple comparisons or checks are performed within conditional statements, leading to redundant operations.

In the functions `buyTokensWithBNB` , `buyTokensWithUSDT` and `buyTokensWithUSDC` there is an `if` statement to ensure referrer is not the `address(0)` or the `msg.sender` however in the line above there is another if statement that check if either of these are true and changes the referrer to the `MARKETING_WALLET` . Therefore the second if statement is redundant

```
if (referrer == address(0) || referrer == msg.sender) {
    referrer = MARKETING_WALLET;
}

if (referrer != address(0) && referrer != msg.sender) {
//..
}
```

Recommendation

It is recommended to refactor conditional statements that return results by eliminating unnecessary code structures and directly returning the outcome of the expression. This practice minimizes the number of operations required, reduces the code footprint, and optimizes memory and gas usage. Simplifying such statements makes the code more readable and improves its overall performance.

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L171,231,288
Status	Acknowledged

Description

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

Specifically, in function `buyTokensWithBNB` calculations are performed multiple times instead of used once and then passed as parameters. In this specific case, an external function in `getLatestPriceBNBPerUSD` performed three times.

Similarly in `buyTokensWithUSDT` and `buyTokensWithUSDC` calculations are performed multiple times.

```
function buyTokensWithBNB(address referrer) external payable
nonReentrant {
     //...
     uint256 amount = getTokenAmountBNB(bnbAmount);
     uint256 bonusTokens = getBonusAmountBNB(bnbAmount);
     uint256 getPriceUSD = getLatestPriceBNBPerUSD();
}
function buyTokensWithUSDT(uint256 usdtAmount, address
referrer) external nonReentrant {
     //...
     uint256 amount = getTokenAmountUSDT(usdtAmount);
     uint256 bonusTokens = getBonusAmountUSDT(usdtAmount);
}
function buyTokensWithUSDC(uint256 usdcAmount, address
referrer) external nonReentrant {
     //...
     uint256 amount = getTokenAmountUSDC(usdcAmount);
     uint256 bonusTokens = getBonusAmountUSDC(usdcAmount);
}
```

Recommendation

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

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

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L452
Status	Acknowledged

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.

Furthermore, the admin can retrieve all the tokens to the `AIRDROP_WALLET` by using the `getUnsoldTokens` function before the presale ends.

```
function getUnsoldTokens() external onlyAdmin {
    uint256 remainingTokens = token.balanceOf(address(this));
    if (remainingTokens == 0) revert NoUnsoldTokensLeft();
    token.safeTransfer(AIRDROP_WALLET, remainingTokens);

    emit UnsoldTokensTransferred(AIRDROP_WALLET,
remainingTokens);
}
```

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 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.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L452
Status	Acknowledged

Description

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

```
function getUnsoldTokens() external onlyAdmin {
    uint256 remainingTokens = token.balanceOf(address(this));
    if (remainingTokens == 0) revert NoUnsoldTokensLeft();
    token.safeTransfer(AIRDROP_WALLET, remainingTokens);

    emit UnsoldTokensTransferred(AIRDROP_WALLET,
remainingTokens);
}
```

Recommendation

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

Temporary Solutions:

These measurements do not decrease the severity of the finding

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

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L68,69,70,72
Status	Acknowledged

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
IERC20 public token =
IERC20(0x43ED084aaDC897FF94F41d0DAc02a9571dC8339F)
IERC20 public usdtToken =
IERC20(0x55d398326f99059fF775485246999027B3197955)
IERC20 public usdcToken =
IERC20(0x8AC76a51cc950d9822D68b83fE1Ad97B32Cd580d)
AggregatorV3Interface public priceFeedBNB =
AggregatorV3Interface(0x0567F2323251f0Aab15c8dFb1967E4e8A7D42ae
E)
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L416
Status	Acknowledged

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
uint256 _newamount
```

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/stable/style-guide.html#naming-conventions.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/EBMPresale.sol#L351,352,367,368,383,384
Status	Acknowledged

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 hundredDollarIncrements = bnbUsdValue /
amountSpentInUsd
uint256 bonusPercent = hundredDollarIncrements *
current.bonusPercent
```

Recommendation

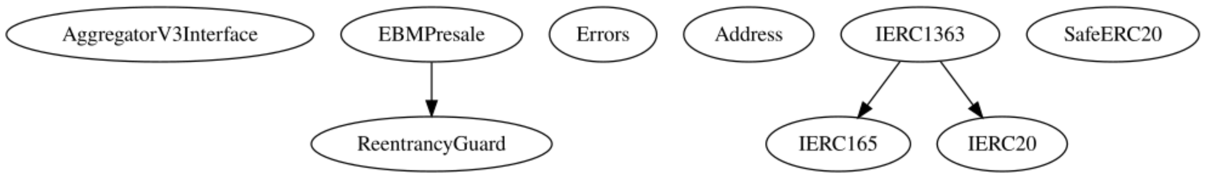
To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

Functions Analysis

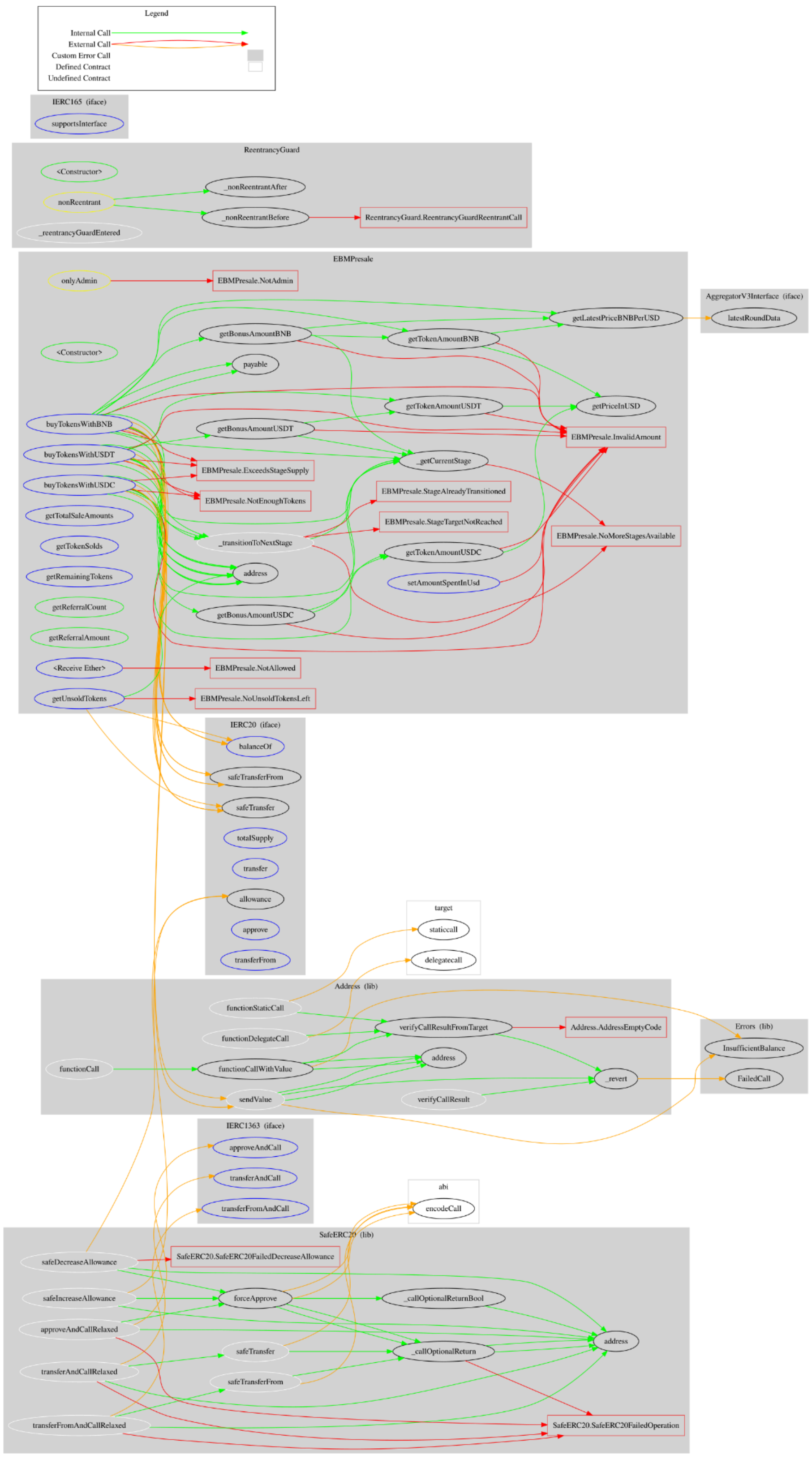
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
EBMPresale	Implementation	ReentrancyGuard		
		Public	✓	-
	_transitionToNextStage	Internal	✓	
	buyTokensWithBNB	External	Payable	nonReentrant
	buyTokensWithUSDT	External	✓	nonReentrant
	buyTokensWithUSDC	External	✓	nonReentrant
	getBonusAmountBNB	Public		-
	getBonusAmountUSDT	Public		-
	getBonusAmountUSDC	Public		-
	getTokenAmountBNB	Public		-
	getTokenAmountUSDT	Public		-
	getTokenAmountUSDC	Public		-
	getLatestPriceBNBPerUSD	Public		-
	setAmountSpentInUsd	External	✓	onlyAdmin
	getPriceInUSD	Public		-
	getTotalSaleAmounts	External		-
	getTokenSolds	External		-
	getRemainingTokens	External		-
	_getCurrentStage	Internal		

	getReferralCount	Public		-
	getReferralAmount	Public		-
	getUnsoldTokens	External	✓	onlyAdmin
		External	Payable	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
IERC165 (iface)
supportsInterface
ReentrancyGuard
<Constructor>
_nonReentrantAfter
nonReentrant
_nonReentrantBefore
ReentrancyGuard.ReentrancyGuardReentrantCall
_reentrancyGuardEntered
EBMPresale
onlyAdmin
EBMPresale.NotAdmin
AggregatorV3Interface (iface)
getLatestPriceBNBPerUSD
latestRoundData
getBonusAmountBNB
getTokenAmountBNB
<Constructor>
payable
getTokenAmountUSDT
getPriceInUSD
buyTokensWithBNB
getBonusAmountUSDT
EBMPresale.InvalidAmount
buyTokensWithUSDT
_getCurrentStage
EBMPresale.ExceedsStageSupply
buyTokensWithUSDC
EBMPresale.StageAlreadyTransitioned
EBMPresale.NotEnoughTokens
getTotalSaleAmounts
EBMPresale.StageTargetNotReached
EBMPresale.NoMoreStagesAvailable
_transitionToNextStage
getTokenSolds
getTokenAmountUSDC
address
getRemainingTokens
setAmountSpentInUsd
getReferralCount
getBonusAmountUSDC
getReferralAmount
<Receive Ether>
EBMPresale.NotAllowed
getUnsoldTokens
EBMPresale.NoUnsoldTokensLeft
IERC20 (iface)
balanceOf
safeTransferFrom
safeTransfer
totalSupply
transfer
allowance
approve
transferFrom
target
staticcall
delegatecall
Address (lib)
functionStaticCall
functionDelegateCall
verifyCallResultFromTarget
Address.AddressEmptyCode
Errors (lib)
InsufficientBalance
address
functionCall
functionCallWithValue
_revert
FailedCall
sendValue
verifyCallResult
IERC1363 (iface)
approveAndCall
transferAndCall
transferFromAndCall
abi
encodeCall
SafeERC20 (lib)
safeDecreaseAllowance
SafeERC20.SafeERC20FailedDecreaseAllowance
safeIncreaseAllowance
forceApprove
_callOptionalReturnBool
approveAndCallRelaxed
address
safeTransfer
_callOptionalReturn
transferAndCallRelaxed
safeTransferFrom
transferFromAndCallRelaxed
SafeERC20.SafeERC20FailedOperation

Summary

EBM Presale contract implements a token presale mechanism. This audit investigates security issues, business logic concerns and potential improvements. EBM Presale is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler error or critical issues.

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

cyberscope.io