Cyberscope

Audit Report

BasaltCoin

November 2023

Network	BSC
Address	0x7a28ba519e9dcdc4f5fbc4f09911ebd15dea8d6c
Audited by	© cyberscope

Table of Contents

Review

Explorer	https://bscscan.com/address/0x7a28ba519e9dcdc4f5fbc4f09911ebd15dea8d6c

Audit Updates

Initial Audit	10 Nov 2023

Source Files

Filename	SHA256
contracts/BasaltTokenSale.sol	055a9c3de801de8860b40bcc066573c5ea4cc82983d4fc211710ebe1c2773901
@openzeppelin/contracts/utils/Context.sol	1458c260d010a08e4c20a4a517882259a23a4baa0b5bd9add9fb6d6a1549814a
@openzeppelin/contracts/utils/Address.sol	8160a4242e8a7d487d940814e5279d934e81f0436689132a4e73394bab084a6d
@openzeppelin/contracts/token/ERC20/IERC20.sol	94f23e4af51a18c2269b355b8c7cf4db8003d075c9c541019eb8dcf4122864d5
@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol	0c8a43f12ac2081c6194d54da96f02ebc457760d6514f6b940689719fcef8c0a
@openzeppelin/contracts/token/ERC20/extensions/draft-IERC20Permit.sol	3e7aa0e0f69eec8f097ad664d525e7b3f0a3fda8dcdd97de5433ddb131db86ef
@openzeppelin/contracts/access/Ownable.sol	9353af89436556f7ba8abb3f37a6677249aa4df6024fbfaa94f79ab2f44f3231

Overview

buyBasaltTokens Functionality

The `buyBasaltTokens` function serves the purpose of facilitating the purchase of Basalt tokens. This function allows a user to buy a specified amount of Basalt tokens using a chosen payment token, identified by `_paymentTokenID`. It ensures that the purchase amount meets the minimum required and does not exceed the available balance of Basalt tokens in the contract.

The function also integrates a referral system. If the buyer is making their first purchase, they must specify a non-zero address as their referrer, who then receives a referral bonus. The function calculates the payable amount in the selected payment token based on the Basalt token amount and the current price. A part of this payment is allocated as a referral fee, which is deducted and sent to the referrer's address. The rest of the payment is transferred to the contract owner.

Additionally, the function updates the user's total purchased amount and the contract's total locked amount with the new Basalt token purchase. It also sets the unlock start time for the buyer, ensuring that subsequent purchases can only be made only if the unlock start time (equal to 30 days since the first buy was made) have not passed. This mechanism is intended to prevent the continuous purchases of the token and maintain its economic stability.

withdrawUnlockedBasaltTokens Functionality

The `withdrawUnlockedBasaltTokens` function manages the withdrawal of Basalt tokens that users have previously purchased. This function enables users to withdraw a portion of their tokens, subject to certain conditions and limitations.

The withdrawal process starts by calculating the user's locked amount of Basalt tokens, which is the difference between their total purchased amount and the amount already withdrawn. The function ensures that the user has tokens available for withdrawal and that the unlock period for these tokens has started, based on the current block timestamp.

The `getAllowedAmount` function calculates the allowed amount a user can withdraw at any given time. It takes into account the time elapsed since the user's first Basalt token purchase and the current block timestamp. The allowed amount is determined based by the time period (months) that have passed since the first purchase of the tokens. As time (months) passes, more tokens become available for withdrawal.

Once the allowed amount for withdrawal is determined and confirmed to be greater than zero, the function updates the user's total withdrawn amount and reduces the total locked amount in the contract accordingly. Finally, the function executes the payment, transferring the allowed amount of Basalt tokens from the contract to the user's address. This mechanism ensures a controlled and staged withdrawal of tokens, intended to maintain the token's economic stability and prevent sudden market fluctuations.

Owner Functions

The owner of the BasaltTokenSale presale contract has the authority to transfer the `basaltToken` to the contract.

addPaymentToken Function

This function allows the contract owner to add a new payment token to the system. It ensures that the token being added is not already present in the list of accepted payment tokens. Once verified, the new token is marked as an accepted payment token and added to the list, enabling users to utilize this token for transactions such as purchasing Basalt tokens.

removePaymentToken Function

This function is designed for the contract owner to remove an existing payment token from the system. It verifies the token ID and removes the specified token from the list of accepted payment tokens. This action ensures that the token can no longer be used for transactions within the system.

changePaymentToken Function

This function empowers the contract owner to adjust the price of a specific payment token. By providing the token ID and the new price, the owner can update the token's price, affecting how it is used in the purchasing of Basalt tokens.

inCaseTokensGetStuck Function

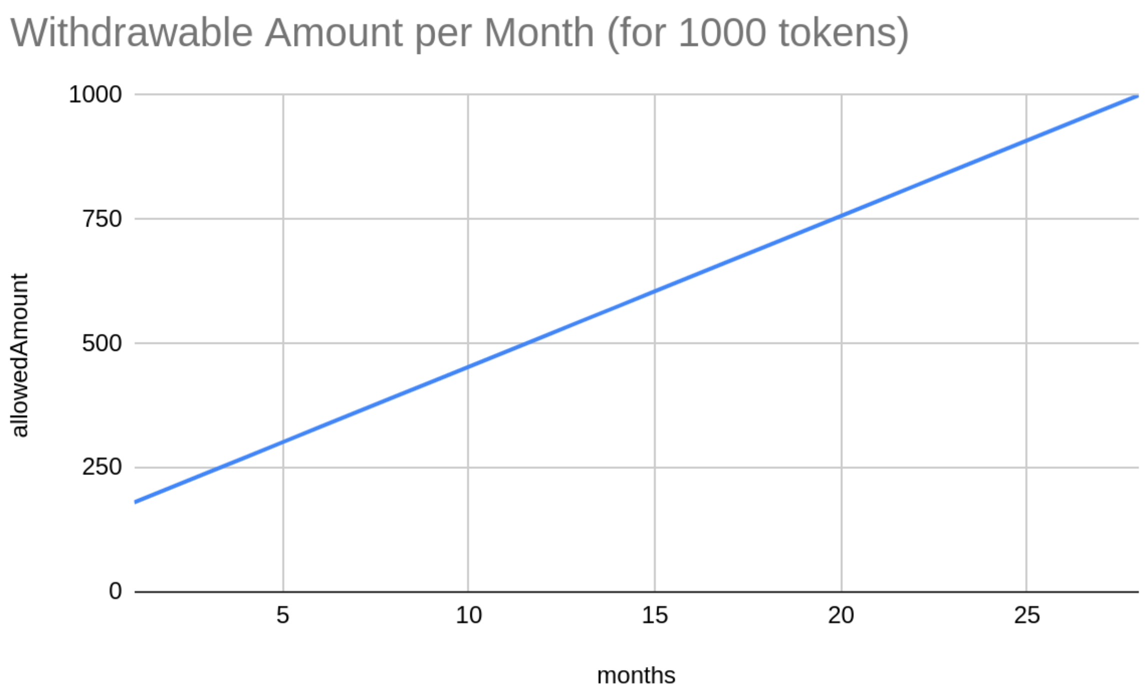
This function gives the authority to the contract owner to handle situations where tokens get stuck in the contract. It allows the owner to transfer a specified amount of any token, including Basalt tokens, to a designated address, ensuring that tokens can be retrieved in exceptional circumstances.

unlockAcceleration Function

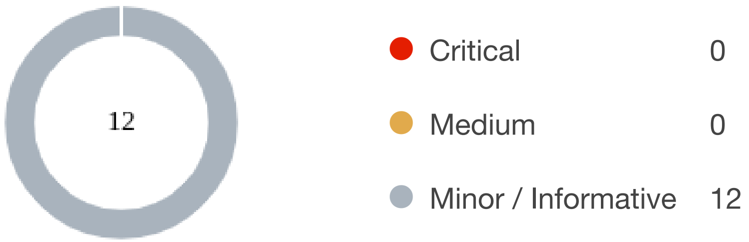
This function provides the contract owner the ability to unlock Basalt tokens for a specific user. The owner can specify an unlock percentage, and the function calculates the amount of tokens to be unlocked based on this percentage. This function can be used to release a portion of a user's locked tokens ahead of the scheduled unlock time.

Withdrawable Amount per Month

The "Withdrawable Amount per Month" chart outlines how a user with a total of 1000 total purchased amount can withdraw over 28 months. The monthly withdrawal limit is cumulative, not additional. This means if a user withdraws some amount in one month, the next month's limit will be the total allowed (of the month) minus what has already been withdrawn. This process continues until the 28th month, with the cumulative limit reaching 1000 tokens.



Findings Breakdown



	Severity	Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	0	0	0	0
●	Minor / Informative	4	8	0	0

Diagnostics

🔴 Critical 🟠 Medium ● Minor / Informative

Severity	Code	Description	Status
●	CCR	Contract Centralization Risk	Unresolved
●	RFB	Referral Fee Bypass	Unresolved
●	RC	Repetitive Calculations	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	DPI	Decimals Precision Inconsistency	Acknowledged
●	EIS	Excessively Integer Size	Acknowledged
●	L04	Conformance to Solidity Naming Conventions	Acknowledged
●	L11	Unnecessary Boolean equality	Acknowledged
●	L13	Divide before Multiply Operation	Acknowledged
●	L14	Uninitialized Variables in Local Scope	Acknowledged
●	L16	Validate Variable Setters	Acknowledged
●	L19	Stable Compiler Version	Acknowledged

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	contracts/BasaltTokenSale.sol#L107
Status	Unresolved

Description

The `unlockAcceleration` function grants the contract owner the ability to selectively unlock tokens for specific users. 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.

```
    function unlockAcceleration(
        address _to,
        uint256 _unlockBP
    ) external onlyOwner {
        require(_unlockBP <= BP, "incorrect _unlockBP");
        UserInfo storage user = userInfo[_to];
        require(user.referrer != address(0), "user {_to} not found");
        uint256 unlockingAmount = ((user.totalPurchasedAmount -
            user.totalWithdrawnAmount) * _unlockBP) / BP;

        if (unlockingAmount > 0) {
            user.totalWithdrawnAmount += unlockingAmount;
            totalLockedAmount -= unlockingAmount;
            _pay(basaltToken, address(this), _to, unlockingAmount);
        }
    }
```

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.

RFB - Referral Fee Bypass

Criticality	Minor / Informative
Location	contracts/BasaltTokenSale.sol#L149
Status	Unresolved

Description

The contract facilitates the purchase of Basalt tokens through the `buyBasaltTokens` function. This function allows users to specify a `_referrer` address, which is intended to identify another user who referred them to the token purchase. However the lack of a check to prevent users from entering their own address as the `_referrer` . This enables users to bypass the intended referral mechanism, allowing them to avoid paying the referrer fee that would normally pay to the `_referrer` address. As a result the the referral fee system can be bypassed

```
   function buyBasaltTokens(
        uint256 _paymentTokenID,
        uint256 _basaltTokenAmount,
        address _referrer
    ) external correctID(_paymentTokenID) {
        ...
        if (user.referrer == address(0)) {
            require(
                _referrer != address(0),
                "_referrer cannot be a zero address"
            );
            referrals[_referrer].push(msg.sender);
            user.referrer = _referrer;
            user.unlockStartTime = block.timestamp + PURCHASE_PERIOD;
        } else {
           ...
        }

        user.totalPurchasedAmount += _basaltTokenAmount;
        totalLockedAmount += _basaltTokenAmount;

        uint256 payableAmount = (_basaltTokenAmount * paymentInfo.price) /
            10 ** 18;

        uint256 referrerFee = (payableAmount * REFERRER_FEE_BP) / BP;
        payableAmount -= referrerFee;
        _pay(paymentInfo.tokenAddress, msg.sender, user.referrer,
referrerFee);
        _pay(paymentInfo.tokenAddress, msg.sender, owner(), payableAmount);
    }
```

`

Recommendation

It is recommended to revise the `buyBasaltTokens` function to include a validation check that prevents the `msg.sender` from using their own address as the `_referrer` . This can be achieved by adding a condition in the function to compare the `_referrer` address with the `msg.sender` address and revert the transaction if they are identical. Implementing this check will ensure that the referral system functions as intended, preventing users from unfairly benefiting from the referral rewards without a legitimate referrer.

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	contracts/BasaltTokenSale.sol#L172,206
Status	Unresolved

Description

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

```
   function withdrawUnlockedBasaltTokens() external {
        UserInfo storage user = userInfo[msg.sender];
        uint256 lockedAmount = user.totalPurchasedAmount -
            user.totalWithdrawnAmount;
    ...
    }

    function getAllowedAmount(
        address _user
    ) public view returns (uint256 allowedAmount) {
        UserInfo memory user = userInfo[_user];
        uint256 lockedAmount = user.totalPurchasedAmount -
            user.totalWithdrawnAmount;
    ...
    }
```

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.

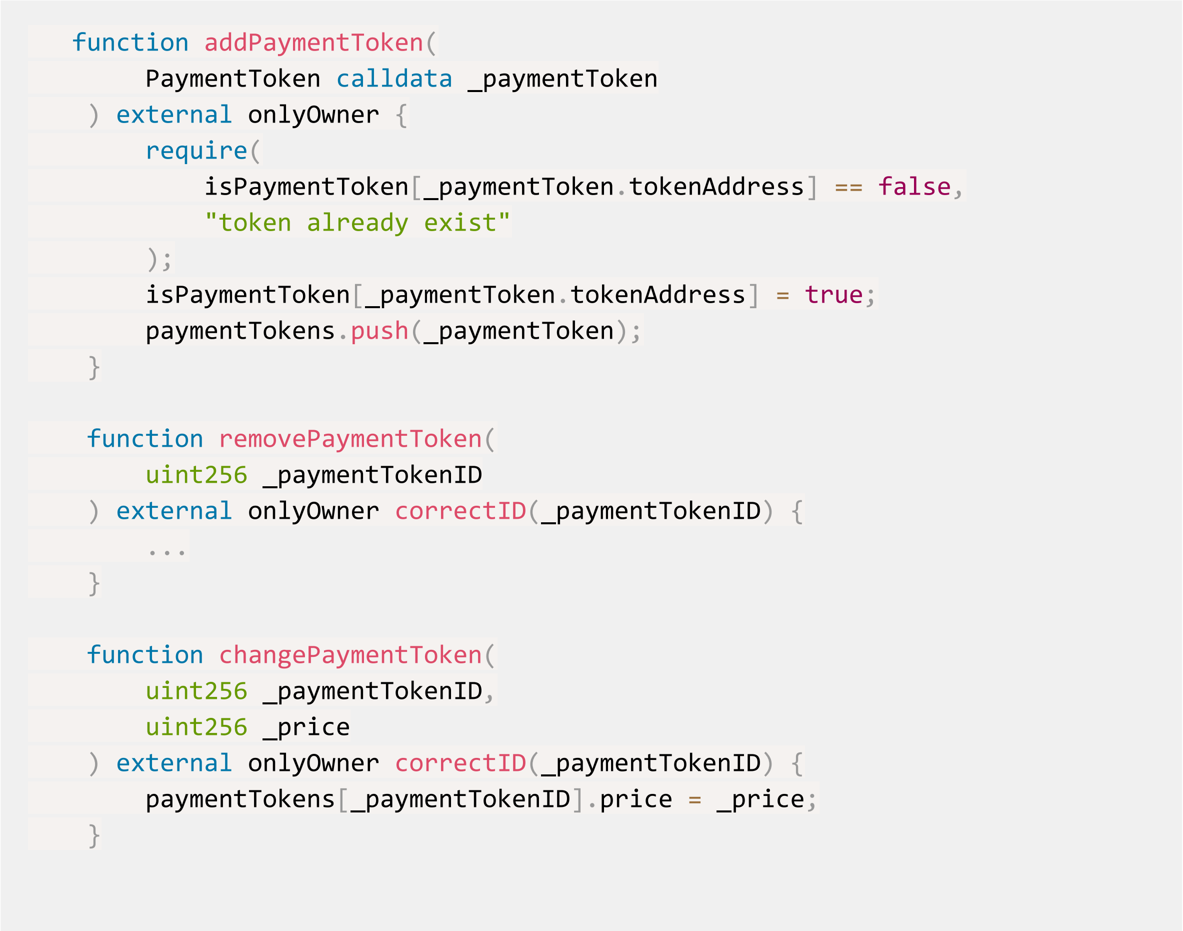
Refactoring the code to assign the calculation result to a variable has several benefits. It improves code readability by making the purpose and intent of the calculation explicit. It also reduces code redundancy, making the method more concise, easier to maintain, and gas effective. Additionally, by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

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	contracts/BasaltTokenSale.sol#L62,73,84
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 addPaymentToken(
        PaymentToken calldata _paymentToken
    ) external onlyOwner {
        require(
            isPaymentToken[_paymentToken.tokenAddress] == false,
            "token already exist"
        );
        isPaymentToken[_paymentToken.tokenAddress] = true;
        paymentTokens.push(_paymentToken);
    }

    function removePaymentToken(
        uint256 _paymentTokenID
    ) external onlyOwner correctID(_paymentTokenID) {
        ...
    }

    function changePaymentToken(
        uint256 _paymentTokenID,
        uint256 _price
    ) external onlyOwner correctID(_paymentTokenID) {
        paymentTokens[_paymentTokenID].price = _price;
    }
```

Recommendation

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

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	contracts/BasaltTokenSale.sol#L130
Status	Acknowledged

Description

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

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

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.

```
function buyBasaltTokens(
     uint256 _paymentTokenID,
     uint256 _basaltTokenAmount,
     address _referrer
 ) external correctID(_paymentTokenID) {
     require(
         _basaltTokenAmount > MINIMUM_PURCHASED_AMOUNT,
         "_basaltTokenAmount is too small"
     );
 ...
 uint256 payableAmount = (_basaltTokenAmount * paymentInfo.price) /
         10 ** 18;
 }
```

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.

Team Update

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

The Basalt token has 18 decimals, and the price variable is specified in the token we use, which can have any number of decimals. For example, if 1 Basalt costs 1 WBTC with 8 decimals (i.e., price == 10**8), 1e18 10 *8 / 1e18 = 10**8 wbtc.

EIS - Excessively Integer Size

Criticality	Minor / Informative
Location	contracts/BasaltTokenSale.sol#L29
Status	Acknowledged

Description

The contract is using a bigger unsigned integer data type that the maximum size that is required. By using an unsigned integer data type larger than necessary, the smart contract consumes more storage space and requires additional computational resources for calculations and operations involving these variables. This can result in higher transaction costs, longer execution times, and potential scalability bottlenecks.

```
   uint256 public constant PURCHASE_PERIOD = 30 days;
    uint256 public constant STAGE_DURATION = 30 days;
    uint256 public constant NUMBER_OF_STAGES = 28;
    uint256 public constant REFERRER_FEE_BP = 250; //2.5%
    uint256 public constant BP = 10000;
```

Recommendation

To address the inefficiency associated with using an oversized unsigned integer data type, it is recommended to accurately determine the required size based on the range of values the variable needs to represent.

Team Update

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

The Solidity compiler will not reserve space for storage of constant or immutable variables. Regarding other variables, converting the native uint256 to smaller types will be more expensive and unnecessary in this case.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/BasaltTokenSale.sol#L63,74,85,86,92,93,94,108,109,125,126,127,204
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.

```
PaymentToken calldata _paymentToken
uint256 _paymentTokenID
uint256 _price
address _token
uint256 _amount
address _to
uint256 _unlockBP
uint256 _basaltTokenAmount
address _referrer
address _user
```

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.

Team Update

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

Our naming conventions follow the style, and the use of underscores to differentiate between local and global variables is widely used, as seen in the Uniswap V3 code.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	contracts/BasaltTokenSale.sol#L65
Status	Acknowledged

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(
    isPaymentToken[_paymentToken.tokenAddress] == false,
    "token already exist"
)
```

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.

Team Update

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

The code in this section is more reliable and less prone to errors compared to using "!boolean."

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	contracts/BasaltTokenSale.sol#L161,164,210,217
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 payableAmount = (_basaltTokenAmount * paymentInfo.price) /
            10 ** 18
uint256 referrerFee = (payableAmount * REFERRER_FEE_BP) / BP
```

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.

Team Update

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

The referral percentage is derived from the payment amount and should be calculated without rounding up.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	contracts/BasaltTokenSale.sol#L56
Status	Acknowledged

Description

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

```
uint256 i
```

Recommendation

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

Team Update

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

Initializing a variable with 0 will be avoided.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	contracts/BasaltTokenSale.sol#L54
Status	Acknowledged

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.

```
basaltToken = _basaltToken
```

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.

Team Update

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

Checks for a zero address in the constructor will be avoided, since errors with the main token address can prevent the contract from functioning properly, and there's no need to complicate the process with an unnecessary zero address check.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	contracts/BasaltTokenSale.sol#L3
Status	Acknowledged

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.0;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Team Update

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

We are compiling with the latest version, ensuring the code remains compatible across various versions, as exemplified by the Uniswap V4 code.

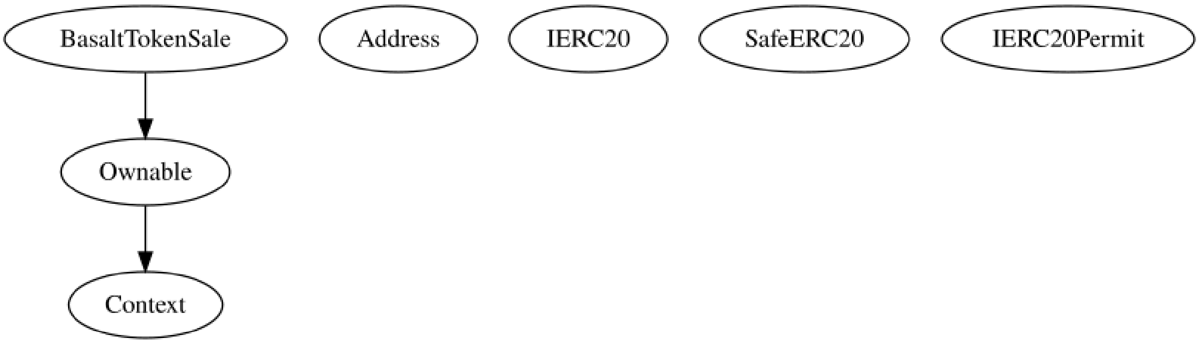
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
BasaltTokenSale	Implementation	Ownable		
		Public	✓	-
	addPaymentToken	External	✓	onlyOwner
	removePaymentToken	External	✓	onlyOwner correctID
	changePaymentToken	External	✓	onlyOwner correctID
	inCaseTokensGetStuck	External	✓	onlyOwner
	unlockAcceleration	External	✓	onlyOwner
	buyBasaltTokens	External	✓	correctID
	withdrawUnlockedBasaltTokens	External	✓	-
	getReferralsInfo	External		-
	getAllowedAmount	Public		-
	_pay	Private	✓	
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		

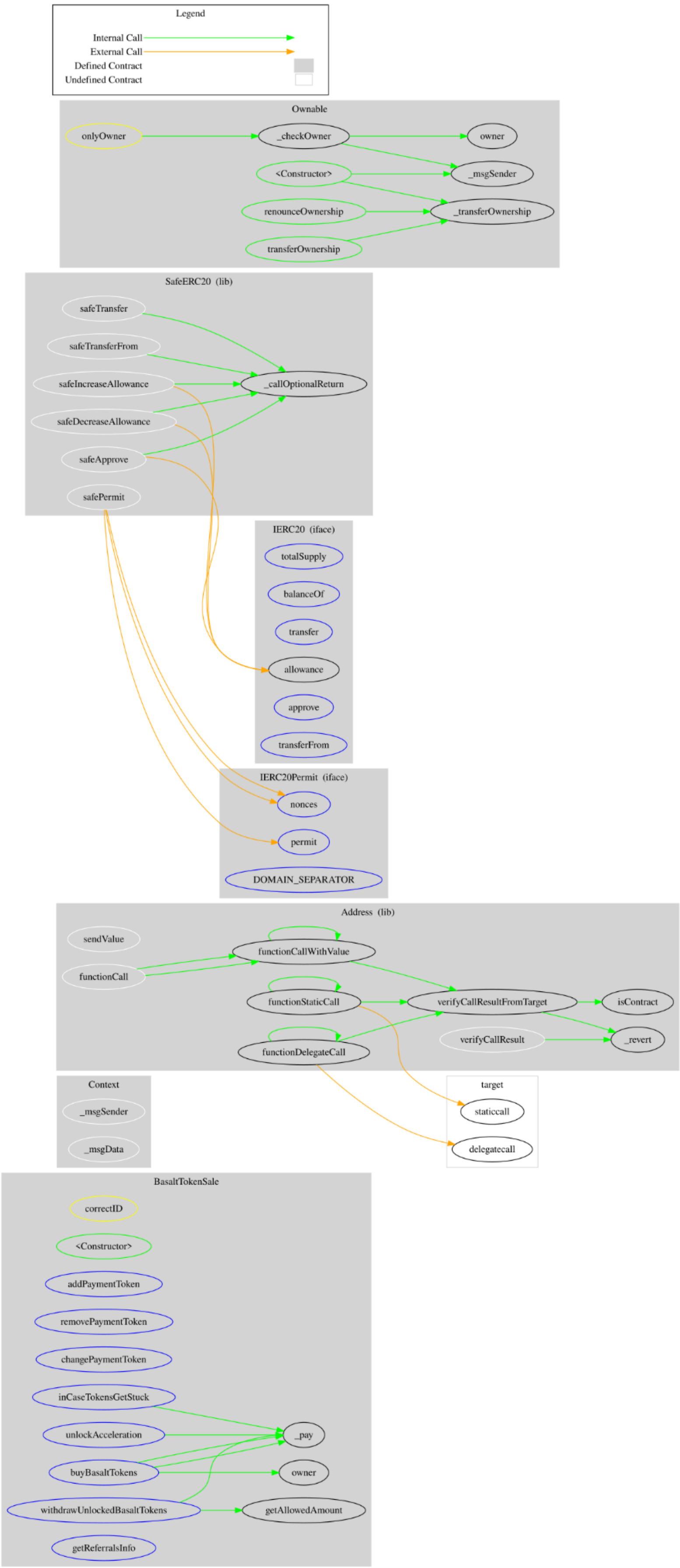
Address	Library			
	isContract	Internal		
	sendValue	Internal	✓	
	functionCall	Internal	✓	
	functionCall	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionStaticCall	Internal		
	functionStaticCall	Internal		
	functionDelegateCall	Internal	✓	
	functionDelegateCall	Internal	✓	
	verifyCallResultFromTarget	Internal		
	verifyCallResult	Internal		
	_revert	Private		
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-

SafeERC20	Library			
	safeTransfer	Internal	✓	
	safeTransferFrom	Internal	✓	
	safeApprove	Internal	✓	
	safeIncreaseAllowance	Internal	✓	
	safeDecreaseAllowance	Internal	✓	
	safePermit	Internal	✓	
	_callOptionalReturn	Private	✓	
IERC20Permit	Interface			
	permit	External	✓	-
	nonces	External		-
	DOMAIN_SEPARATOR	External		-
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	_checkOwner	Internal		
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
	_transferOwnership	Internal	✓	

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Ownable
onlyOwner
_checkOwner
owner
<Constructor>
_msgSender
renounceOwnership
_transferOwnership
transferOwnership
SafeERC20 (lib)
safeTransfer
safeTransferFrom
safeIncreaseAllowance
_callOptionalReturn
safeDecreaseAllowance
safeApprove
safePermit
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
IERC20Permit (iface)
nonces
permit
DOMAIN_SEPARATOR
Address (lib)
sendValue
functionCallWithValue
functionCall
functionStaticCall
verifyCallResultFromTarget
isContract
verifyCallResult
_revert
functionDelegateCall
Context
_msgSender
_msgData
target
staticcall
delegatecall
BasaltTokenSale
correctID
<Constructor>
addPaymentToken
removePaymentToken
changePaymentToken
inCaseTokensGetStuck
unlockAcceleration
_pay
buyBasaltTokens
owner
withdrawUnlockedBasaltTokens
getAllowedAmount
getReferralsInfo

Summary

BasaltCoin contract implements a presale mechanism. This audit investigates security issues, business logic concerns and potential improvements. The BasaltTokenSale contract is a presale mechanism designed for the sale, management, and withdrawal of Basalt tokens. It features functionalities for token purchase, referral rewards, staged withdrawal, and includes special owner privileges for managing payment methods and accelerating token unlock processes.

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