BlockÂpex

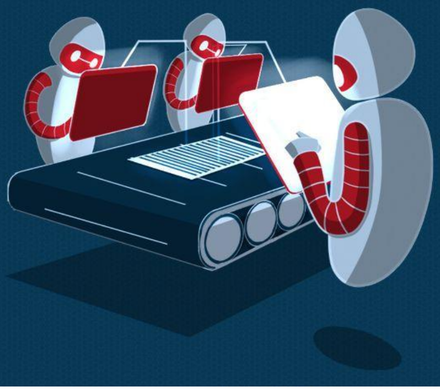
SMART CONTRACT SECURITY ANALYSIS REPORT

```
pragma solidity 0.7.0;
contract Contract {

    function hello() public returns (string) {
        return "Hello World!";
    }

    function findVulnerability() public returns (string) {
        return "Finding Vulnerability";
    }

    function solveVulnerability() public returns (string) {
        return "Solve Vulnerability";
    }
}
```



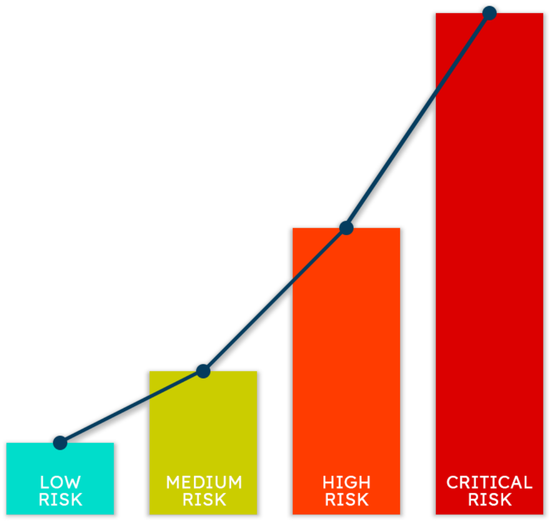
PREFACE

Objectives

The purpose of this document is to highlight the identified bugs/issues in the provided codebase. This audit has been conducted in a closed and secure environment, free from influence or bias of any sort. This document may contain confidential information about IT systems/architecture and intellectual property of the client. It also contains information about potential risks and the processes involved in mitigating/exploiting the risks mentioned below.

The usage of information provided in this report is limited, internally, to the client. However, this report can be disclosed publicly with the intention to aid our growing blockchain community; under the discretion of the client.

Key understandings



CRITICAL RISK xxxx	Critical vulnerabilities are too easy to exploit and can lead to damages/loss in assets or manipulations.
HIGH RISK xxx	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution.
MEDIUM RISK xx	Medium-level vulnerabilities are equally imperative to fix but they tend to have minimal impact on asset loss or data manipulations.
LOW RISK x	Lowest-level vulnerabilities, informational errors, violating code styles/practices usually can't affect smart contract execution; hence they can be ignored.

TABLE OF CONTENTS

INTRODUCTION

BlockApex (Auditor) was contracted by Chainpals (Client) for the purpose of conducting a Smart Contract Audit/Code Review. This document presents the findings of our analysis which took place on 3 June 2022 .

Name
Chainpals Presale
Auditor
Kaif Ahmed | Mohammad Memon
Platform
Ethereum/Solidity/BSC
Type of review
Manual Code Review | Automated Code Review
Methods
Architecture Review, Functional Testing, Computer-Aided Verification, Manual Review
BSC Scan/Contract address
https://bscscan.com/address/0x46ca81a88f3fae582427a92b44d784b307b2058e#code
Documentation
https://chainpals.io/assets/document/ChainpalsLightpaper.pdf
Document log
Initial Audit: 7th June 2022
Final Audit: 17 June 2022

Scope

The git-repository shared was checked for common code violations along with vulnerability-specific probing to detect **major issues/vulnerabilities**. Some specific checks are as follows:

Code review		**Functional review**
Reentrancy	Unchecked external call	Business Logics Review
Ownership Takeover	ERC20 API violation	Functionality Checks
Timestamp Dependence	Unchecked math	Access Control & Authorization
Gas Limit and Loops	Unsafe type inference	Escrow manipulation
DoS with (Unexpected) Throw	Implicit visibility level	Token Supply manipulation
DoS with Block Gas Limit	Deployment Consistency	Asset's integrity
Transaction-Ordering Dependence	Repository Consistency	User Balances manipulation
Style guide violation	Data Consistency	Kill-Switch Mechanism
Costly Loop		Operation Trails & Event Generation

Project Overview

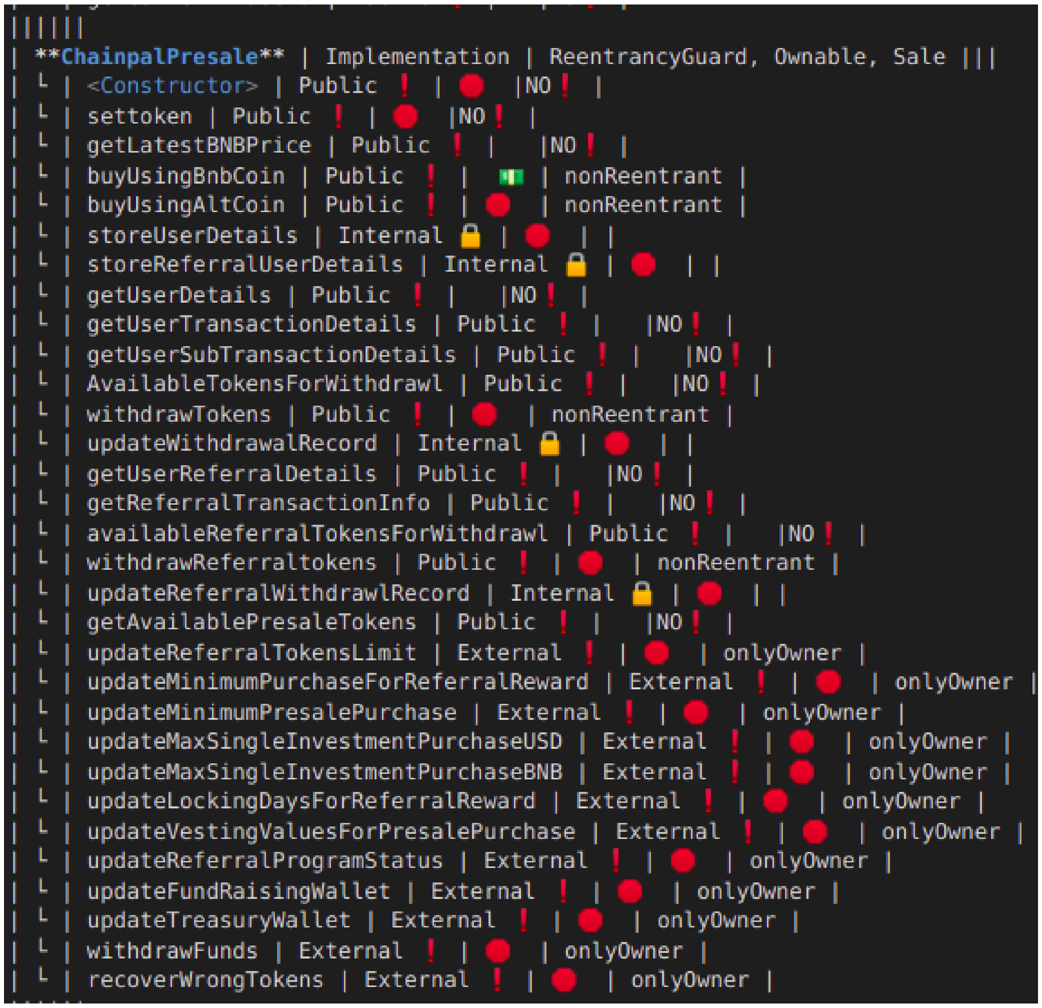
Chainpals Token is a BEP20 token contract. Our contract at hand is the Chainpals Presale contract that uses the BEP20 token.

System Architecture

The main contract is called **ChainpalsPresale.sol**. This contract contains the functionality for the presale of the BEP20-based Chainpals Token. The presale is supposed to go forward in three stages, each with fixed purchasable amounts and at a fixed cost. The cost starts off at 0.25 USD in the first phase, moves to 0.35 USD in the second phase and then to 0.45 in the last phase. Also, the contract holds the 35% of 20M initial supply of the Chainpals Tokens, which is fixed.

Methodology & Scope

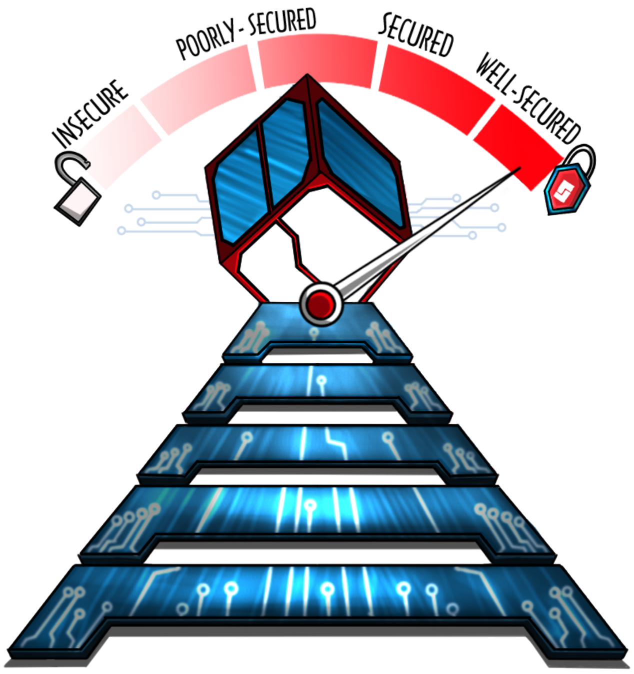
The code came to us in the form of a zip, containing a truffle directory, the contract and the tests. Initially, we ran the contract and tested the functionality of all the functions manually. After that, we moved to Foundry to try all kinds of scenarios. After all the logical and functional testing, we moved to code optimizations and solidity design patterns to ensure consistency and readability.



```
||||||
| **ChainpalPresale** | Implementation | ReentrancyGuard, Ownable, Sale |||
| └ | <Constructor> | Public ❗️ | 🛑  |NO❗️ |
| └ | settoken | Public ❗️ | 🛑  |NO❗️ |
| └ | getLatestBNBPrice | Public ❗️ |   |NO❗️ |
| └ | buyUsingBnbCoin | Public ❗️ |  💵 | nonReentrant |
| └ | buyUsingAltCoin | Public ❗️ | 🛑  | nonReentrant |
| └ | storeUserDetails | Internal 🔒 | 🛑  | |
| └ | storeReferralUserDetails | Internal 🔒 | 🛑  | |
| └ | getUserDetails | Public ❗️ |   |NO❗️ |
| └ | getUserTransactionDetails | Public ❗️ |   |NO❗️ |
| └ | getUserSubTransactionDetails | Public ❗️ |   |NO❗️ |
| └ | AvailableTokensForWithdrawl | Public ❗️ |   |NO❗️ |
| └ | withdrawTokens | Public ❗️ | 🛑  | nonReentrant |
| └ | updateWithdrawalRecord | Internal 🔒 | 🛑  | |
| └ | getUserReferralDetails | Public ❗️ |   |NO❗️ |
| └ | getReferralTransactionInfo | Public ❗️ |   |NO❗️ |
| └ | availableReferralTokensForWithdrawl | Public ❗️ |   |NO❗️ |
| └ | withdrawReferraltokens | Public ❗️ | 🛑  | nonReentrant |
| └ | updateReferralWithdrawlRecord | Internal 🔒 | 🛑  | |
| └ | getAvailablePresaleTokens | Public ❗️ |   |NO❗️ |
| └ | updateReferralTokensLimit | External ❗️ | 🛑  | onlyOwner |
| └ | updateMinimumPurchaseForReferralReward | External ❗️ | 🛑  | onlyOwner |
| └ | updateMinimumPresalePurchase | External ❗️ | 🛑  | onlyOwner |
| └ | updateMaxSingleInvestmentPurchaseUSD | External ❗️ | 🛑  | onlyOwner |
| └ | updateMaxSingleInvestmentPurchaseBNB | External ❗️ | 🛑  | onlyOwner |
| └ | updateLockingDaysForReferralReward | External ❗️ | 🛑  | onlyOwner |
| └ | updateVestingValuesForPresalePurchase | External ❗️ | 🛑  | onlyOwner |
| └ | updateReferralProgramStatus | External ❗️ | 🛑  | onlyOwner |
| └ | updateFundRaisingWallet | External ❗️ | 🛑  | onlyOwner |
| └ | updateTreasuryWallet | External ❗️ | 🛑  | onlyOwner |
| └ | withdrawFunds | External ❗️ | 🛑  | onlyOwner |
| └ | recoverWrongTokens | External ❗️ | 🛑  | onlyOwner |
```

AUDIT REPORT

Executive Summary

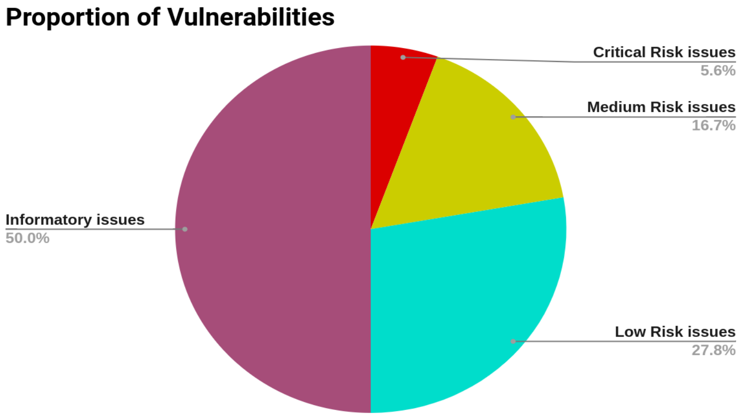


The analysis indicates that all of the functionalities in the contracts audited are **working properly**.

Our team performed a technique called "Filtered Audit", where the contract was separately audited by two individuals. After their thorough and rigorous process of manual testing, an automated review was carried out using Surya. All the flags raised were manually reviewed and re-tested.

Our team found:

# of issues	Severity of the risk
1	Critical Risk issue(s)
0	High Risk issue(s)
3	Medium Risk issue(s)
5	Low Risk issue(s)
9	Informatory issue(s)



Findings

#	Findings	Risk	Status
1.	Potential economic crash	Critical	Fixed
2.	Unsupervised contract balance return	Medium	Fixed
3.	Potential out-of-gas scenario	Medium	acknowledged
4.	Check does not fulfill purpose of limit purchase	Medium	acknowledged
5.	Zero address check missing in *recoverWrongTokens()*	Low	Fixed
6.	Missing zero address checks in constructor	Low	Fixed
7.	Missing zero value checks in constructor	Low	Fixed
8.	*updateWithdrawalRecord():* Condition satisfaction does not terminate loop	Low	Fixed
9.	*updateReferralWithdrawRecord():* Condition satisfaction does not terminate loop	Low	Fixed
10.	*withdrawReferralTokens()* is gas-costly	Informatory	acknowledged
11.	Hardcode the USDT and BUSD token addresses	Informatory	Fixed
12.	Use struct to simplify readability	Informatory	Fixed
13.	*buyUsingBnbCoin()* and *buyUsingAltCoin()* should be set to external.	Informatory	Fixed
14.	Spelling mistakes in two functions	Informatory	Fixed
15.	Unnecessary conversion	Informatory	Fixed
16.	Put all external functions on the top, below the constructor.	Informatory	Fixed

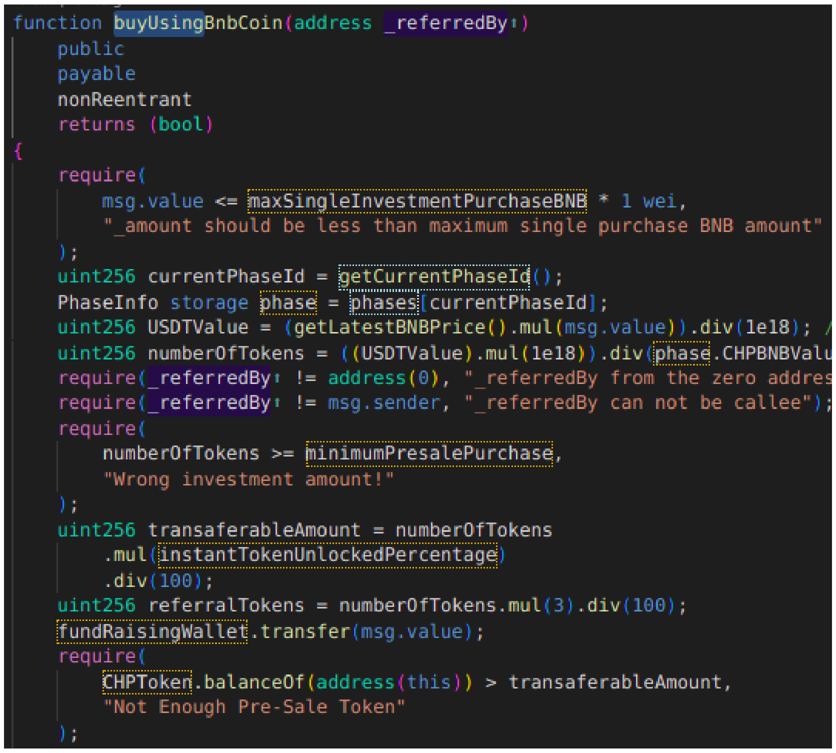
17.	TreasuryWallet address check	Informatory	Fixed
18.	Make all necessary variables constant	Informatory	acknowledged

Critical risk issues

1. Potential economic crash

Description: The *buyUsingBnbCoin()* and *buyUsingAltCoin()* both only check for the total remaining amount of tokens in the contract. This includes the yet-to-be unlocked amount purchased by other users.

- Impact:
    1. If all the tokens of the contract have been sold in presale, only x% amount will be withdrawn (where x is the instant unlock percentage). The contract checks the entire balance present, not taking into consideration whether any of that is the locked amount belonging to someone. The contract therefore allows others to potentially buy other people's locked tokens. Ultimately, when the locked tokens are unlocked and a user comes to claim them, the contract would not have them.
    2. The *AvailableTokensForWithdrawal()* and *availableReferralTokensForWithdrawal()* functions will return the valid amount to withdraw, but their associated withdrawal function also checks the balance of the CHP Token, which will not pass.



```
function buyUsingBnbCoin(address _referredBy)
    public
    payable
    nonReentrant
    returns (bool)
{
    require(
        msg.value <= maxSingleInvestmentPurchaseBNB * 1 wei,
        "_amount should be less than maximum single purchase BNB amount"
    );
    uint256 currentPhaseId = getCurrentPhaseId();
    PhaseInfo storage phase = phases[currentPhaseId];
    uint256 USDTValue = (getLatestBNBPrice().mul(msg.value)).div(1e18); /
    uint256 numberOfTokens = ((USDTValue).mul(1e18)).div(phase.CHPBNBValu
    require(_referredBy != address(0), "_referredBy from the zero addres
    require(_referredBy != msg.sender, "_referredBy can not be callee");
    require(
        numberOfTokens >= minimumPresalePurchase,
        "Wrong investment amount!"
    );
    uint256 transaferableAmount = numberOfTokens
        .mul(instantTokenUnlockedPercentage)
        .div(100);
    uint256 referralTokens = numberOfTokens.mul(3).div(100);
    fundRaisingWallet.transfer(msg.value);
    require(
        CHPToken.balanceOf(address(this)) > transaferableAmount,
        "Not Enough Pre-Sale Token"
    );
```

Status: Acknowledged.

Medium risk issues

2. Unsupervised contract balance return

Description: The *getAvailablePresaleTokens()* function allows a user to get the total available presale tokens. This function returns the total balance of the contract, which includes the yet-to-be unlocked tokens belonging to the other users. This unchecked value can cause problems in the long run when the tokens unlock and the token owners attempt to withdraw them.

```
function getAvailablePresaleTokens() public view returns (uint256) {
        return CHPToken.balanceOf(address(this));
   }
```

Remedy: Each user already has a Struct to keep track of their purchases. This function should return the difference between all held tokens and the total tokens, which will be the true value of available presale tokens.

Status:: Fixed as per BlockApex Recommendation.

3. Potential out-of-gas scenario

Description: Inside the *WithdrawTokens()* function, we have calls going to *availableTokensForWithdrawal()* and *updateWithdrawalRecord()*. Both of these functions contain nested loops, which may result in a potential out-of-gas problem.

```
function withdrawTokens(uint256 _amount)
        public
        nonReentrant
        returns (bool)
    {
        uint256 availableTokensToClaim = AvailableTokensForWithdrawl(
            msg.sender
        );
        require(_amount <= availableTokensToClaim, "Wrong withdrawal amount");
        require(
            CHPToken.balanceOf(address(this)) > _amount,
            "Not Enough Pre-Sale Token"
        );

        if (updateWithdrawalRecord(address(msg.sender), _amount)) {
            CHPToken.safeTransfer(address(_msgSender()), _amount);
        }
        emit TokenWithdrawal(_msgSender(), _amount);
        return true;
    }
```

Remedy: Create a mapping which will track time from user purchase to 1st redeem, 1st redeem to last redeem and run the loop according to the time return from mapping.

Developer Response: Functions referred in these two 2 points are functional operations and we will require to have loops in there to check if the tokens are unlocked for the user to claim/redeem.

Status: Acknowledged.

4. Check does not fulfill purpose of limit purchase

Description: Inside the buy tokens functions, namely *buyUsingBnbCoin()* and *buyUsingAltCoin()*, the contract has a check to ensure that a user does not spend more than 10,000 stable coins or 35 BNB. This is a one time limit to purchase in presale, but it does not limit the user from buying in a loop or multiple times at once.

Remedy: Limit the user from buying in a loop, or buying multiple times.

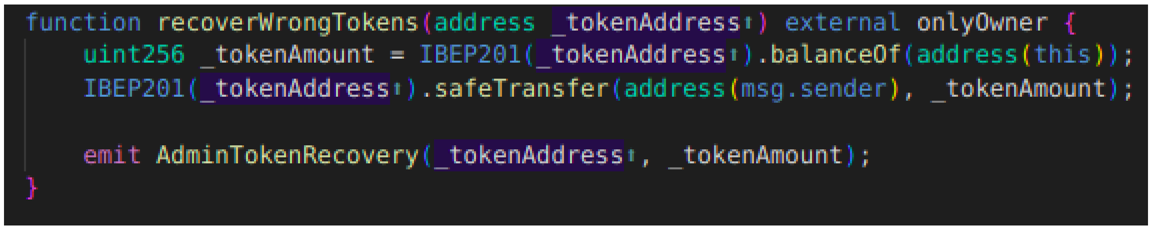
Developer Response: We're limiting the user to purchase only 10,000 USD or 35BNB worth of tokens from presale is handled through the frontend interface of the system.

Status: Acknowledged.

Low risk issues

5. Zero address check missing in recoverWrongTokens()

Description: The function does not check for zero address.



```
function recoverWrongTokens(address _tokenAddress) external onlyOwner {
    uint256 _tokenAmount = IBEP201(_tokenAddress).balanceOf(address(this));
    IBEP201(_tokenAddress).safeTransfer(address(msg.sender), _tokenAmount);

    emit AdminTokenRecovery(_tokenAddress, _tokenAmount);
}
```

Remedy: Place a zero address check inside the function to save a caller from redundant calls.

Status:: Fixed as per BlockApex Recommendation.

6. Missing zero address checks in constructor

Description: The constructor does not check if any of the input addresses are equal to the zero addresses.

Remedy: Place a zero address check on all the addresses passing to the constructor.

Status:: Fixed as per BlockApex Recommendation.

7. Missing zero value checks in constructor

Description: The constructor does not check for the value of the integer input in the constructor.

Remedy: Place a zero value check on all the values passing to the constructor.

Status:: Fixed as per BlockApex Recommendation.

8. *updateWithdrawalRecord():* Condition satisfaction does not terminate loop

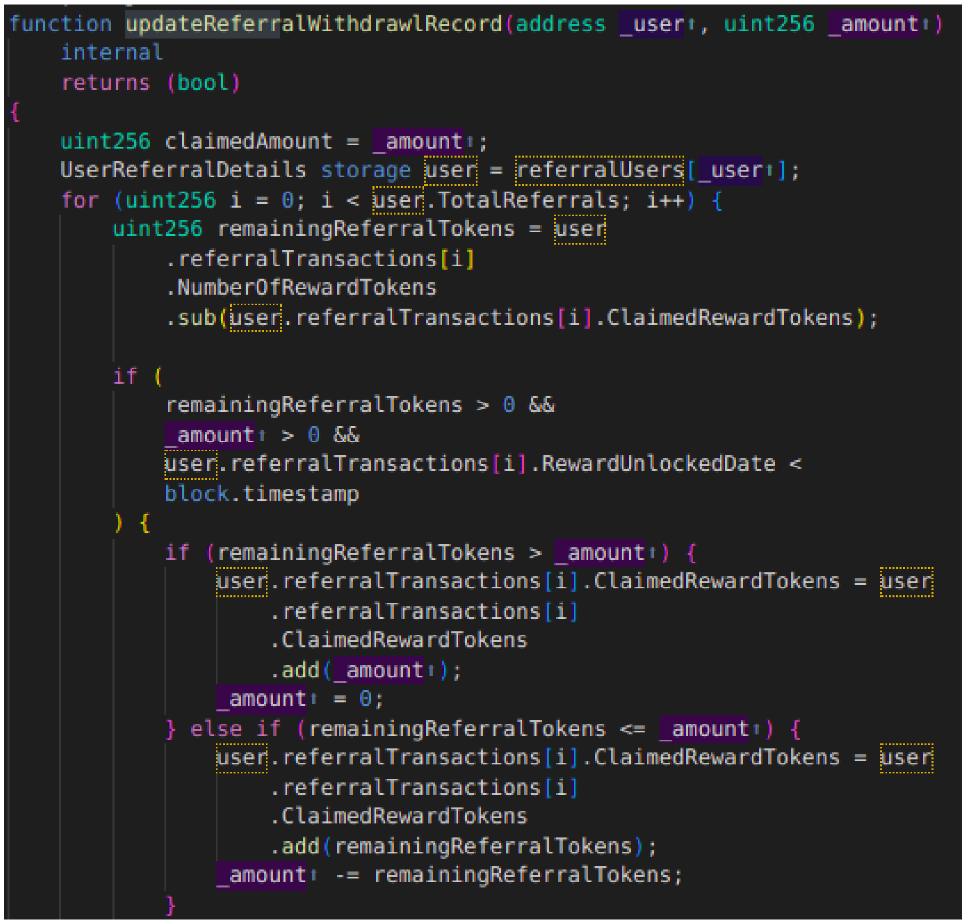
Description: This function runs a loop to check whether the user has enough funds to withdraw. Upon successfully withdrawing all the requested amount, the loop should end. Inside the contract, the loop continues to run till the end. This is a waste of gas.

Remedy: Place a break statement inside the first if statement to terminate the loop.

Status:: Fixed as per BlockApex Recommendation.

9. ***updateReferralWithdrawRecord():* Condition satisfaction does not terminate loop**

Description: This function runs a loop to check whether the user has enough funds to withdraw. Upon successfully withdrawing all the requested amount, the loop should end. Inside the contract, the loop continues to run till the end. This is a waste of gas.



```
function updateReferralWithdrawlRecord(address _user, uint256 _amount)
    internal
    returns (bool)
{
    uint256 claimedAmount = _amount;
    UserReferralDetails storage user = referralUsers[_user];
    for (uint256 i = 0; i < user.TotalReferrals; i++) {
        uint256 remainingReferralTokens = user
            .referralTransactions[i]
            .NumberOfRewardTokens
            .sub(user.referralTransactions[i].ClaimedRewardTokens);

        if (
            remainingReferralTokens > 0 &&
            _amount > 0 &&
            user.referralTransactions[i].RewardUnlockedDate <
            block.timestamp
        ) {
            if (remainingReferralTokens > _amount) {
                user.referralTransactions[i].ClaimedRewardTokens = user
                    .referralTransactions[i]
                    .ClaimedRewardTokens
                    .add(_amount);
                _amount = 0;
            } else if (remainingReferralTokens <= _amount) {
                user.referralTransactions[i].ClaimedRewardTokens = user
                    .referralTransactions[i]
                    .ClaimedRewardTokens
                    .add(remainingReferralTokens);
                _amount -= remainingReferralTokens;
            }
```

Remedy: Place a break statement inside the first if-statement to terminate the loop.

Status:: Fixed as per BlockApex Recommendation.

Informatory issues and Optimization

10. *withdrawReferralTokens()* is gas-costly

Description: This function makes a call to *availableReferralTokensForWithdrawal()* and *updateReferralTokensRecord()*. Both of these functions contain loops, which makes this function very expensive in terms of gas. It is possible the gas cost for looping through the mapping could become so significant that the gas limit for the block is reached.

Developer Response: Functions referred in these two 2 points are functional operations and we will require to have loops in there to check if the tokens are unlocked for the user to claim/redeem.

11. Hardcode the USDT and BUSD token addresses

Description: The addresses of the USDT Token and the BUSD Token remain the same throughout the life of the contract. Instead of accepting the address inside the constructor, it should be hardcoded.

Status:: Fixed as per BlockApex Recommendation.

12. Use struct to simplify readability.

Description: The constructor accepts 14 parameters before deployment.

13. *buyUsingBnbCoin()* and *buyUsingAltCoin()* should be set to external.

Description: Both of these functions are supposed to be called from outside the contract. Therefore, it makes no sense to set it to public. Setting it to external allows us to save some gas as well.

Status:: Fixed as per BlockApex Recommendation.

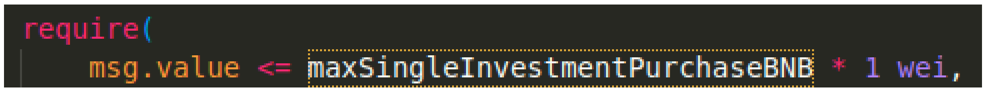
14. Spelling mistakes in two functions

Description: Spelling mistake found *buyUsingBnbCoin()* and *buyUsingAltCoin()*.

Status:: Fixed as per BlockApex Recommendation.

15. Unnecessary conversion

Description: It is unnecessary to multiply value with 1 wei. It is basically multiplying the value with 1, which will not change anything.



```
require(
    msg.value <= maxSingleInvestmentPurchaseBNB * 1 wei,
```

16. Put all external functions on the top, below the constructor.

Description: As stated in the Solidity style guide, the functions should be grouped according to their visibility and ordered:

- constructor
- receive function (if exists)
- fallback function (if exists)
- external
- public
- internal
- Private

Status:: Fixed as per BlockApex Recommendation.

17. TreasuryWallet address check

Description: *updateTreasuryWallet()* should have a check to ensure that the updated address is not equal to the owner address.

Status:: Fixed as per BlockApex Recommendation.

18. Make all necessary variables constant

Description: There are several variables whose value remains the same throughout the life of the contract. All such variables should be marked constant.

Status: Acknowledged.

DISCLAIMER

The smart contracts provided by the client for audit purposes have been thoroughly analyzed in compliance with the global best practices till date w.r.t cybersecurity vulnerabilities and issues in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token, its sale or any other aspect of the project.

Crypto assets/tokens are results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-Party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third-party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have its vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks.

This audit cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.