BlockApex

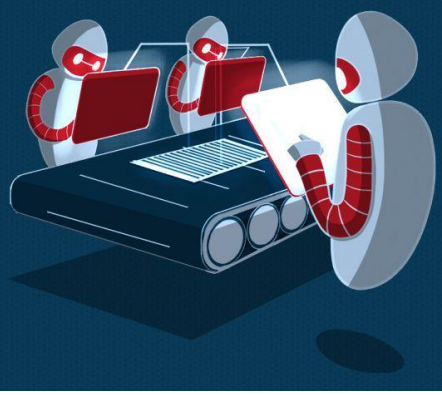
SMART CONTRACT SECURITY ANALYSIS REPORT

```
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";
    }
}
```



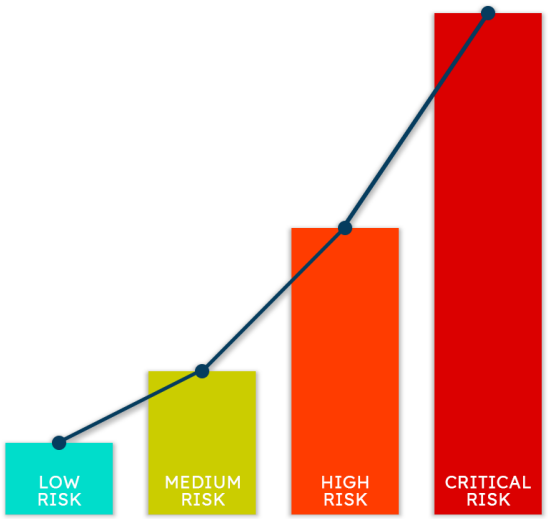
PREFACE

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 ××××	Critical vulnerabilities are too easy to exploit and can lead to damages/loss in assets or manipulations.
HIGH RISK ×××	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution.
MEDIUM RISK ××	Medium-level vulnerabilities are equally imperative to fix but they tend to have minimal impact on asset loss or data manipulations.
LOW RISK ×	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 30 may 2022.

Name
Chainpals Token (BEP20)
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/0x20aaa94BC42e361bc351c9D5F023FC1a288C8aC2#code
Documentation
https://chainpals.io/assets/document/ChainpalsLightpaper.pdf
Document log
Initial Audit: 1st june 2022
Final Audit: 17th 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

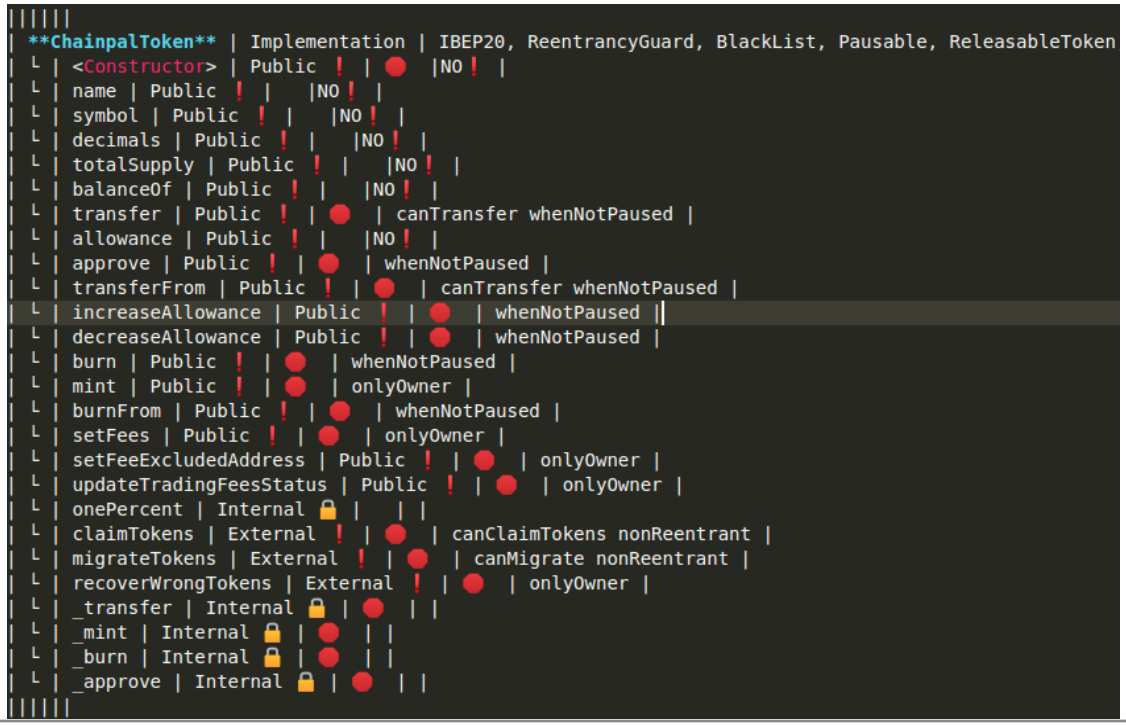
Chainpals Token is a BEP20 token contract. It works slightly differently from the traditional BEP20 contract.

System Architecture

The main contract is called **ChainpalsToken**. The minting and transfer of the complete supply is done during deployment. This means new tokens can only be minted if the old ones are burnt. There is mention of sale and presale, meaning these features will be introduced in the future. The contract holds 35% of the total supply, while the rest is transferred to known actors.

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.



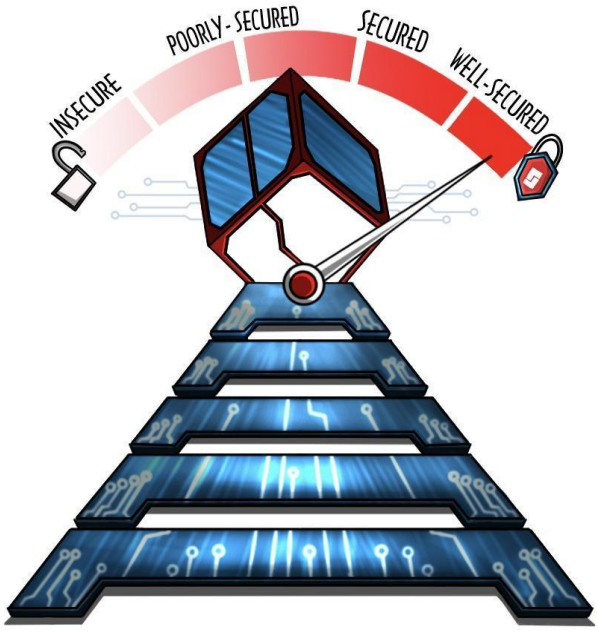
```
||||||
| **ChainpalToken** | Implementation | IBEP20, ReentrancyGuard, BlackList, Pausable, ReleasableToken
| └ | <Constructor> | Public ❗ | 🔴 |NO❗ |
| └ | name | Public ❗ | |NO❗ |
| └ | symbol | Public ❗ | |NO❗ |
| └ | decimals | Public ❗ | |NO❗ |
| └ | totalSupply | Public ❗ | |NO❗ |
| └ | balanceOf | Public ❗ | |NO❗ |
| └ | transfer | Public ❗ | 🔴 | canTransfer whenNotPaused |
| └ | allowance | Public ❗ | |NO❗ |
| └ | approve | Public ❗ | 🔴 | whenNotPaused |
| └ | transferFrom | Public ❗ | 🔴 | canTransfer whenNotPaused |
| └ | increaseAllowance | Public ❗ | 🔴 | whenNotPaused |
| └ | decreaseAllowance | Public ❗ | 🔴 | whenNotPaused |
| └ | burn | Public ❗ | 🔴 | whenNotPaused |
| └ | mint | Public ❗ | 🔴 | onlyOwner |
| └ | burnFrom | Public ❗ | 🔴 | whenNotPaused |
| └ | setFees | Public ❗ | 🔴 | onlyOwner |
| └ | setFeeExcludedAddress | Public ❗ | 🔴 | onlyOwner |
| └ | updateTradingFeesStatus | Public ❗ | 🔴 | onlyOwner |
| └ | onePercent | Internal 🔒 | | |
| └ | claimTokens | External ❗ | 🔴 | canClaimTokens nonReentrant |
| └ | migrateTokens | External ❗ | 🔴 | canMigrate nonReentrant |
| └ | recoverWrongTokens | External ❗ | 🔴 | onlyOwner |
| └ | _transfer | Internal 🔒 | 🔴 | |
| └ | _mint | Internal 🔒 | 🔴 | |
| └ | _burn | Internal 🔒 | 🔴 | |
| └ | _approve | Internal 🔒 | 🔴 | |
||||||
```

AUDIT REPORT

Executive Summary

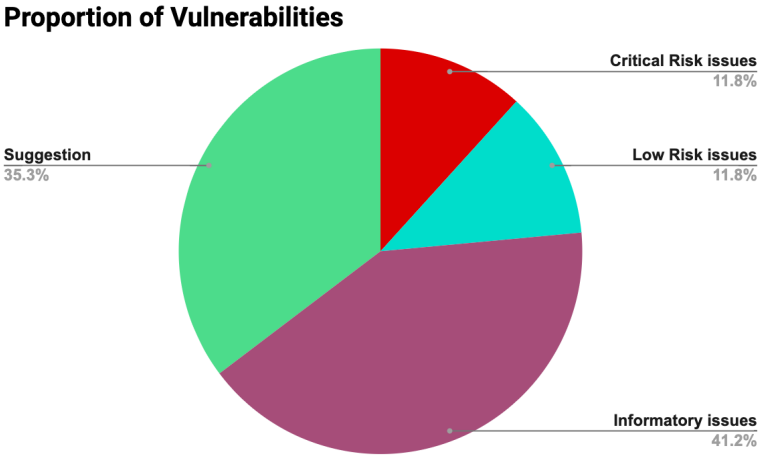
The analysis indicates that all of the functionalities in the contracts audited are **working properly**.

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
0	Critical Risk issue(s)
0	High Risk issue(s)
0	Medium Risk issue(s)
2	Low Risk issue(s)
7	Informatory issue(s)
6	Suggestion(s)



Findings

#	Findings	Risk	Status
1.	Potential Centralization risk	Low	acknowledged
2.	No upper limit on fee percentage	Low	Fixed
3.	Use struct to simplify readability	Informatory	acknowledged
4.	Expensive uint comparison in require statements.	Informatory	Fixed
5.	Use memory keyword instead of calldata on parameters inside external function	Informatory	acknowledged
6.	V1 contract will hold both V1 and V2 tokens at the time of V2 token Migration.	Informatory	acknowledged
7.	Follow Solidity style guide	Informatory	acknowledged
8.	Loop running till the end	Informatory	Fixed
9.	Zero address checks in constructor	Informatory	Fixed
10.	*setFeeExcludedAddress* needs to contract only address	Suggestion	Fixed
11.	There is no view function to view ongoing stage	Suggestion	acknowledged
12.	Known parameters should be hardcoded	Suggestion	acknowledged
13.	Potential destruction of blacklisted address's funds	Suggestion	acknowledged
14.	Make sure that the error messages clearly explain the reason for the error	Suggestion	Fixed
15.	*transfer()* should not deduct 1% fee from whitelisted addresses	Suggestion	Fixed

Low-risk issues

1. **Potential centralization risk.**

   Description: The *recoverWrongTokens()* function allows the owner to pass an address, parse it into BEP20 and withdraw the tokens. This creates centralization risk that the owner is able to withdraw 35% share of contract tokens.

```
[⠢] Solc 0.6.12 finished in 3.01s
Compiler run successful

Running 1 test for test/Contract.t.sol:ContractTest
[PASS] test_migrateTokens() (gas: 222393)
Logs:
  **** Recovering wrong tokens ****
  Owner's address initially: , 0
  Sent V1 as wrong address, this amount was withdrawn: , 34650000000000000000
  Owner balance after withdraw, 34650000000000000000
```

   Status: Acknowledged.

2. **No upper limit on fee percentage.**

   Description: The *setFees()* function allows the owner to set the fee percentage and the fee reward address. There is no check to limit the fee percentage. Setting a very high fee percentage can basically drain funds in big amounts.

```
ftrace | funcSig
function setFees(uint256 _feeRewardPct, address _feeRewardAddress)
    public
    onlyOwner
{
    require(
        _feeRewardAddress != address(0),
        "Fee reward address must not be zero address"
    );

    FeeRewardPct = _feeRewardPct;
    FeeRewardAddress = _feeRewardAddress;
}
```

   Remedy: Make a require function that makes sure that the value sent as parameter is legitimate (ideally, under 10%).

   Status:: Fixed as per BlockApex Recommendation.

Informatory issues and Optimization

3. **Use struct to simplify readability.**

   Description: The constructor accepts 16 parameters before deployment.

   Remedy: Make a struct for all the parameters and send the struct into the constructor as a single parameter.

```
struct Deploy {
        uint256 _initialSupply;
        uint256 _feeRewardPct;
        address _feeRewardAddress;
        address _PresaleWallet;
        address _PrivateSaleWallet;
        address _SaleWallet;
        address _CommunityFutureWallet;
        address _BurnWallet;
        address _MarketingWallet;
        address _ProductDevelopmentWallet;
        address _FounderWallet;
        string _marketing;
        string _founder;
        string _productDevelopment;
        string _presale;
        string _sale;
}
    ftrace
    constructor(Deploy memory params) public {

        releaseAgent = msg.sender;
        maxSupply = params._initialSupply;
        uint256 initialSupply = maxSupply;
        PresaleWallet = params._PresaleWallet;
        PrivateSaleWallet = params._PrivateSaleWallet;
        SaleWallet = params._SaleWallet;
        CommunityFutureWallet = params._CommunityFutureWallet;
        BurnWallet = params._BurnWallet;
        MarketingWallet = params._MarketingWallet;
        ProductDevelopmentWallet = params._ProductDevelopmentWallet;
        FounderWallet = params._FounderWallet;
        tradingFeesApplicable = true;
```

Status: Acknowledged.

4. **Expensive uint comparison in require statements.**

   Description: In many cases, we see this check:

   (x > 0)

   The x is uint in every case, meaning it will never go below 0. The lowest value it can have is 0 itself. We can change this condition to test the same thing using less gas.

   Remedy: This check can be replaced by:

   (x != 0)

   This condition will check whether x is 0 or not. If it is 0, the check returns false, otherwise it proves that x is greater than 0 because uint values cannot be negative.

   Status: Fixed as per BlockApex Recommendation.

5. **Use memory keyword instead of calldata on parameters inside external function.**

   Description: Reference-type parameters can be assigned calldata keyword instead of memory. Calldata is another space for storing externally received parameters. It is less expensive than using the memory space.

   Remedy: Replace the memory keywords with calldata.

   Developer Response: There are very few instances where memory keyword can be changed to calldata. Also, replacing them will not make a big difference in the gas cost and hence, we suggest to keep it as it is.

   Status: Acknowledged.

6. **V1 contract will hold both V1 and V2 tokens at the time of V2 token Migration.**

   Description: The 35% share locked inside the V1 contract cannot be migrated. When V2 is launched, the 35% will not be able to migrate unless a mechanism is added in the future.

   Only the owner wallet is able to withdraw this 35% share of the contract.

   Developer Response: V1 tokens will get redeemed by defined wallets within their planned unlock phases after tokens are created and if in future there is a requirement then will create V2 tokens which will give users a facility that they can migrate tokens from V1 to V2. To summarize, none tokens will be locked for only the owner wallet to withdraw while migration is on.

   Status:

7. **Solidity style guide:**

   These were some cases of inconsistency with the style guide:

   a. The external functions should be between the public functions and the constructor.
   b. The public view functions should be below the state changing public functions.
   c. In some functions the starting parenthesis is not on the same line.

   Status: Acknowledged.

8. **Loop running till the end:**

Description: Inside the *updateWithdrawalRecord()* function, the *claimTokens()* functions sends the name of the sale and the amount to claim. The function checks each slot and gives back the amount by iterating through all the slots. The for loop does not stop even when the requested amount is removed. The extra running will increase the gas cost over time as more slots are added.

```
function updateWithdrawalRecord(string memory _saleName, uint256 _amount)
        internal
        returns (bool)
    {
        SaleInfo storage s = sales[_saleName];
        uint256 amount = _amount;
        for (uint256 i = 0; i < s.TotalSlots; i++) {
            uint256 slotTokens = s.saleSlots[i].TotalTokens.sub(
                s.saleSlots[i].ClaimedTokens
            );
            if (
                slotTokens > 0 &&
                amount > 0 &&
                s.saleSlots[i].WithdrawalDate < block.timestamp
            ) {
                if (slotTokens > amount) {
                    s.saleSlots[i].ClaimedTokens = s
                        .saleSlots[i]
                        .ClaimedTokens
                        .add(amount);
                    amount = 0;
                } else if (slotTokens <= amount) {
                    s.saleSlots[i].ClaimedTokens = s
                        .saleSlots[i]
                        .ClaimedTokens
                        .add(slotTokens);
                    amount -= slotTokens;
                }
            }
        }
        s.TotalClaimed += _amount;
        return true;
    }
```

Remedy: Add a **break** keyword inside the (slotTokens > amount) check so that once the requested amount is found inside the current slot, there is no need for the loop to keep running.

```
function updateWithdrawalRecord(string memory _saleName, uint256 _amount)
        internal
        returns (bool)
    {
        SaleInfo storage s = sales[_saleName];
        uint256 amount = _amount;
        for (uint256 i = 0; i < s.TotalSlots; i++) {
            uint256 slotTokens = s.saleSlots[i].TotalTokens.sub(
                s.saleSlots[i].ClaimedTokens
            );
            if (
                slotTokens > 0 &&
                amount > 0 &&
                s.saleSlots[i].WithdrawalDate < block.timestamp
            ) {
                if (slotTokens > amount) {
                    s.saleSlots[i].ClaimedTokens = s
                        .saleSlots[i]
                        .ClaimedTokens
                        .add(amount);
                    amount = 0;
                     //Break here
                } else if (slotTokens <= amount) {
                    s.saleSlots[i].ClaimedTokens = s
                        .saleSlots[i]
                        .ClaimedTokens
                        .add(slotTokens);
                    amount -= slotTokens;
                }
            }
        }
        s.TotalClaimed += _amount;
        return true;
    }
```

Alternatively, you can change the *claimTokens()* to accept the slot number, so that the loop is removed from the scenario entirely and only the assigned slot is targeted, checked upon and used to give the requested tokens.

Status: Fixed as per BlockApex Recommendation.

9. **Zero address checks in constructor:**

   Description: Constructor is taking a lot of addresses from parameter. There should be a zero/null address check to verify each address.

   Remedy: Place require statements to check each address.

   Status: Fixed as per BlockApex Recommendation.

Suggestions

10. ***setFeeExcludedAddress* needs to contract only address**

    Description: The excluded address should be a contract address, not a wallet address. Since this is a community-centered project, there should be no extra privileges to selected community members.

    Status: Fixed as per BlockApex Recommendation.

11. **There is no view function to view ongoing stage**

    Description: At the moment there is no way to check what stage the platform is in. It would be better to implement a view function that returns the current stage.

    Developer Response: To check status/phase of the token there are two public values released and paused. Using that we can determine that tokens are currently released/unreleased or paused/unpaused.

    Status: Acknowledged

12. **Known parameters should be hardcoded**

    Description: The string values that are sent to the constructor can be hardcoded if they are known from the start.

Status: Acknowledged.

13. Potential destruction of blacklisted address's funds

Description: Blacklisted addresses will have their funds locked for as long as they are blacklisted. For the time that an address is blacklisted, their funds are basically destroyed. That is, no one can access those funds in that period, till the address is removed from the blacklist.

Developer Response: Under our token requirements we will need a functionality using which the owner wallet can set a user/wallet address as blacklisted or remove him from the blacklist. Our functionality doesn't include destroying the tokens under a blacklisted wallet.

Status: Acknowledged.

14. Make sure that the error messages clearly explain the reason for the error

Description: In some cases, the error messages in the revert statements do not clearly deliver the message. They should be changed to clearly explain why a statement was reverted. Certain spelling mistakes were also found, which should be fixed.

Status: Fixed as per BlockApex Recommendation.

15. *transfer()* should not deduct 1% fee from whitelisted addresses

Description: The tokens are already pre-minted. While calling the *claimTokens()* function, it deducts 1% from the caller's requested amount. The whitelisted addresses should be excluded from the fee deduction.

Status: Fixed as per BlockApex Recommendation.

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.