Smart Contract Code Review and Security Analysis Report

Bitcoin Africa

Customer:Bitcoin Africa
Prepared on: 20th July 2022
Platform: BSC
Language: Solidity

HyperAnts

Table of Contents

Disclaimer

This document may contain confidential information about its systems and intellectual property of the customer as well as information about potential vulnerabilities and methods of their exploitation.

The report containing confidential information can be used internally by the customer or it can be disclosed publicly after all vulnerabilities are fixed - upon the decision of the customer.

Document

Name	Smart Contract Code Review and Security Analysis Report of Bitcoin Africa
Platform	BSC / Solidity
File 1	bitcoinAfrica.sol
Link Source	https://bscscan.com/address/0x22855528a7a05A5a4D56D516130070bb7aE1aC0e#code
MD5 hash	c908dafd1eace5c180f1b81e9f5a70f1
SHA256 hash	f9f6ddd096d1f2d56f7b84400c85ec25007c1a7842f6117a9d9fbc29e6f2ca66
Date	**20/07/2022**

Introduction

HyperAnts (Consultant) were contracted by Bitcoin Africa (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report represents the findings of the security assessment of the customer`s smart contract and its code review conducted between 17th - 20th July 2022.

This contract consists of one file.

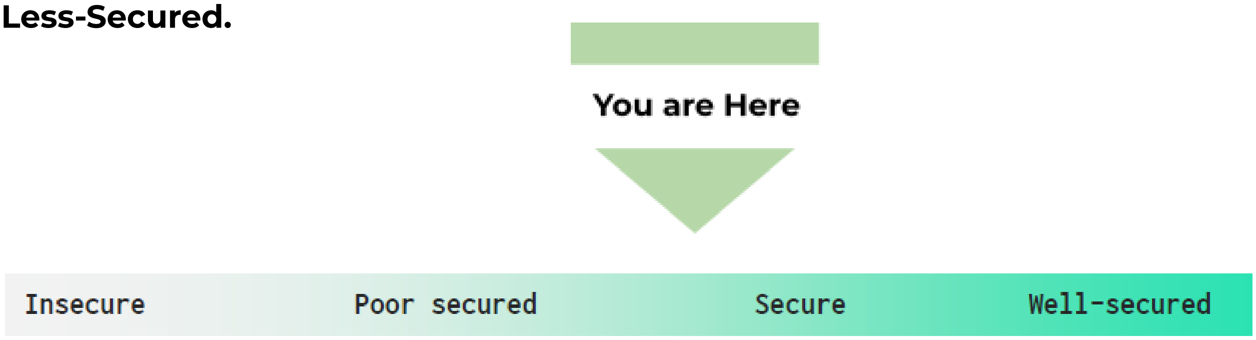
Project Scope

The scope of the project is a smart contract. We have scanned this smart contract for commonly known and more specific vulnerabilities, below are those considered (the full list includes but is not limited to):

- Reentrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Byte array vulnerabilities
- Style guide violation
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Unchecked external call - Unchecked math
- Unsafe type inference
- Implicit visibility level

Executive Summary

According to the assessment, the customer's solidity smart contract is now **Less-Secured.**



Automated checks are with smartDec, Mythril, Slither and remix IDE. All issues were performed by our team, which included the analysis of code functionality, the manual audit found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the audit overview section. The general overview is presented in the AS-IS section and all issues found are located in the audit overview section.

We found the following;

Total Issues	4
Critical	0
High	1
Medium	0
Low	1
Very Low	2

Code Quality

The libraries within this smart contract are part of a logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned to a specific address and its properties/methods can be reused many times by other contracts.

The BitcoinAfrica has not provided scenario and unit test scripts, which helped to determine the integrity of the code in an automated way.

Overall, the code is not well commented. Commenting can provide rich documentation for functions, return variables and more. Use of the Ethereum Natural Language Specification Format (NatSpec) for commenting is recommended.

Documentation

The hash of that file is mentioned in the table. As mentioned above, It's recommended to write comments in the smart contract code, so anyone can quickly understand the programming flow as well as complex code logic.

Comments are very helpful in understanding the overall architecture of the protocol. It also provides a clear overview of the system components, including helpful details, like the lifetime of the background script.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure. Those were based on well known industry standard open source projects and even core code blocks that are written well and systematically.

AS-IS Overview

It is a Staking & Token Contract

bitcoinAfrica.sol

File And Function Level Report

File : bitcoinAfrica.sol

Contract: BITCOINAFRICA

Inherit : IBEP20, Ownable

Observation: Passed

Test Report: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	onlyToken	modifier	Passed	All Passed	No Issue	Passed
2	setDistributionCriteria	write	Passed	All Passed	No Issue	Passed
3	setShare	write	Passed	All Passed	No Issue	Passed
4	deposit	write	Passed	All Passed	No Issue	Passed
5	process	write	Passed	All Passed	No Issue	Passed
6	shouldDistribute	read	Passed	All Passed	No Issue	Passed
7	distributeDividend	write	Passed	All Passed	No Issue	Passed
8	claimDividend	write	Passed	All Passed	No Issue	Passed
9	getPaidEarnings	read	Passed	All Passed	No Issue	Passed

10	getCurrentBalance	read	Passed	All Passed	No Issue	Passed
11	getUnpaidEarnings	read	Passed	All Passed	No Issue	Passed
12	getCumulativeDividends	read	Passed	All Passed	No Issue	Passed
13	addShareholder	write	Passed	All Passed	No Issue	Passed
14	removeShareholder	write	rectify	All Passed	rectify	rectify
15	totalSupply	read	Passed	All Passed	No Issue	Passed
16	decimals	read	Passed	All Passed	No Issue	Passed
17	symbol	read	Passed	All Passed	No Issue	Passed
18	name	read	Passed	All Passed	No Issue	Passed
19	getOwner	read	Passed	All Passed	No Issue	Passed
20	balanceOf	read	Passed	All Passed	No Issue	Passed
21	approve	write	Passed	All Passed	No Issue	Passed
22	approveMax	write	Passed	All Passed	No Issue	Passed
23	transfer	write	Passed	All Passed	No Issue	Passed
24	transferFrom	write	Passed	All Passed	No Issue	Passed
25	_transferFrom	write	Passed	All Passed	No Issue	Passed
26	_basicTransfer	write	Passed	All Passed	No Issue	Passed
27	basicTransfer	write	rectify	All Passed	rectify	rectify
28	shouldTakeFee	read	Passed	All Passed	No Issue	Passed
29	totalFeePerTx	read	Passed	All Passed	No Issue	Passed
30	_takeBothFee	write	Passed	All Passed	No Issue	Passed
31	_takeBurnFee	write	Passed	All Passed	No Issue	Passed
32	_takeMarketFee	write	Passed	All Passed	No Issue	Passed
33	shouldSwapBack	read	Passed	All Passed	No Issue	Passed

34	swapBack	write	Passed	All Passed	No Issue	Passed
35	setBuyFee	write	Passed	All Passed	No Issue	Passed
36	setSellFee	write	Passed	All Passed	No Issue	Passed
37	launch	write	Passed	All Passed	No Issue	Passed
38	setTxLimit	write	Passed	All Passed	No Issue	Passed
39	setTargetLiquidity	write	Passed	All Passed	No Issue	Passed
40	setIsDividendExempt	write	Passed	All Passed	No Issue	Passed
41	setIsFeeExempt	write	Passed	All Passed	No Issue	Passed
42	setIsTxLimitExempt	write	Passed	All Passed	No Issue	Passed
43	setBuyFeePercent	write	Passed	All Passed	No Issue	Passed
44	setSellFeePercent	write	Passed	All Passed	No Issue	Passed
45	setFeeReceivers	write	Passed	All Passed	No Issue	Passed
46	setSwapBackSettings	write	Passed	All Passed	No Issue	Passed
47	setDistributionCriteria	write	Passed	All Passed	No Issue	Passed
48	setDistributorSettings	write	Passed	All Passed	No Issue	Passed
49	getCirculatingSupply	read	Passed	All Passed	No Issue	Passed
50	addSniperInList	write	Passed	All Passed	No Issue	Passed
60	removeSniperFromList	write	Passed	All Passed	No Issue	Passed
61	enableOrDisableAntibot	write	Passed	All Passed	No Issue	Passed
62	claimDividend	write	Passed	All Passed	No Issue	Passed
63	getPaidDividend	read	Passed	All Passed	No Issue	Passed

64	getUnpaidDividend	read	Passed	All Passed	No Issue	Passed
65	getTotalDistributedDividend	read	Passed	All Passed	No Issue	Passed
66	getLiquidityBacking	read	Passed	All Passed	No Issue	Passed
67	isOverLiquified	read	Passed	All Passed	No Issue	Passed

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to lost tokens etc.

High	High level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g. public access to crucial functions.
Medium	Medium level vulnerabilities are important to fix; however, they cannot lead to lost tokens.
Low	Low level vulnerabilities are most related to outdated, unused etc. These code snippets cannot have a significant impact on execution.
Lowest Code Style/ Best Practice	Lowest level vulnerabilities, code style violations and information statements cannot affect smart contract execution and can be ignored.

Audit Findings

Critical:

No critical severity vulnerabilities were found.

High:

1 high severity vulnerabilities were found.

1. This function gives full privileges to the owner. Owner can issue a new limitless token and transfer it to any wallet. Owner can exploit the whole system using this function.

```
function basicTransfer(address recipient, uint256 amount)
    external
    onlyOwner
    returns (bool)
{
    _balances[recipient] = _balances[recipient].add(amount);
    return true;
}
```

Medium:

No medium severity vulnerabilities were found.

Low:

1 low severity vulnerabilities were found.

1. Here is a missing statement. You are not releasing the removed shareHolder record from “shareholderIndexes”.

```
    function removeShareholder(address shareholder) internal {
        shareholders[shareholderIndexes[shareholder]] = shareholders[
            shareholders.length - 1
        ];
        shareholderIndexes[
            shareholders[shareholders.length - 1]
        ] = shareholderIndexes[shareholder];
        shareholders.pop();
    }
}
```

Very Low:

2 very low severity vulnerabilities were found.

1. Here you should use a require check for invalid address to avoid unnecessary gas fees.

```
function claimDividend(address _user) public {
    distributeDividend(_user);
}
```

```
function distributeDividend(address shareholder) internal {
    if (shares[shareholder].amount == 0) {
        return;
    }
```

2. You don't need the SafeMath library for Solidity 0.8+ so please avoid it.

```
library SafeMath {
    function tryAdd(uint256 a, uint256 b)
        internal
        pure
        returns (bool, uint256)
    {
        unchecked {
            uint256 c = a + b;
            if (c < a) return (false, 0);
            return (true, c);
        }
    }

    function trySub(uint256 a, uint256 b)
        internal
        pure
        returns (bool, uint256)
    {
        unchecked {
            if (b > a) return (false, 0);
            return (true, a - b);
        }
    }

    function tryMul(uint256 a, uint256 b)
        internal
        pure
        returns (bool, uint256)
    {
        unchecked {
            // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
            // benefit is lost if 'b' is also tested.
            // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
            if (a == 0) return (true, 0);
```

Note For Contract Users

There are some owner only functions. Those can be called by the owner's wallet only. So, if the owner's wallet is compromised, then it carries the risk of the contract becoming vulnerable.

the owner can withdraw all balance from the contract.

```
function basicTransfer(address recipient, uint256 amount)
    external
    onlyOwner
    returns (bool)
{
    _balances[recipient] = _balances[recipient].add(amount);
    return true;
}
```

Owner has full control over the smart contract. Thus, technical auditing does not guarantee the project's ethical side.

Please do your due diligence before investing. Our audit report is never an investment advice.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyse the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

HyperAnts Disclaimer

The smart contracts given for audit have been analysed in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

Because the total number of test cases are unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree 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.

Technical Disclaimer

Smart contracts are deployed and executed on the blockchain. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can’t guarantee explicit security of the audited smart contracts.

HyperAnts

Email: support@hyperants.com

Website: hyperants.com