



**RD
AUDITORS**

Hugo Inu, Smart Contract, Code Review and Security Analysis Report



Customer: Hugo Inu
Prepared on: 2nd June 2023
Platform: Binance
Language: Solidity

rdauditors.com

Table of Contents

Disclaimer	2
Documentation	3
Introduction	4
Project Scope	5
Executive Summary	6
Code Quality	6
Documentation	8
Use of Dependencies	9
AS-IS Overview	10
Code Flow Diagram - Hugo.sol	13
Code Flow Diagram - Slither Results Log	14
Severity Definitions	20
Audit Findings	21
Conclusion	22
Note For Contract Users	22
Our Methodology	24
Disclaimers	26

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.

Documentation

Name	Smart Contract Code Review and Security Analysis Report of Hugo Inu
Platform	Binance/ Solidity
File 1	Hugo.sol
MD5 hash	ad08ad2cd0bcc98eb36bdaab6ded4f10
SHA256 hash	968e8de30434fe89067dd2c28327f52282340d9b1ce4eda25e68de033847e5b9
Date	02/06/2023

Introduction

RD Auditors (Consultant) were contracted by Hugo Inu (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 1st- 2nd June 2023.

This contract consists of one file.

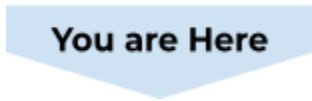
Project Scope

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 **Well-Secured**.








You are Here

 **Insecure** **Poorly Secured** **Secure** **Well-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	0
 Critical	0
 High	0
 Medium	0
 Low	0
 Very Low	0

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

Documentation

We were given a Hugo Inu smart contract code in the form of a BSCScan web link. The hash of that code is mentioned above 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

Hugo.sol

File And Function Level Report

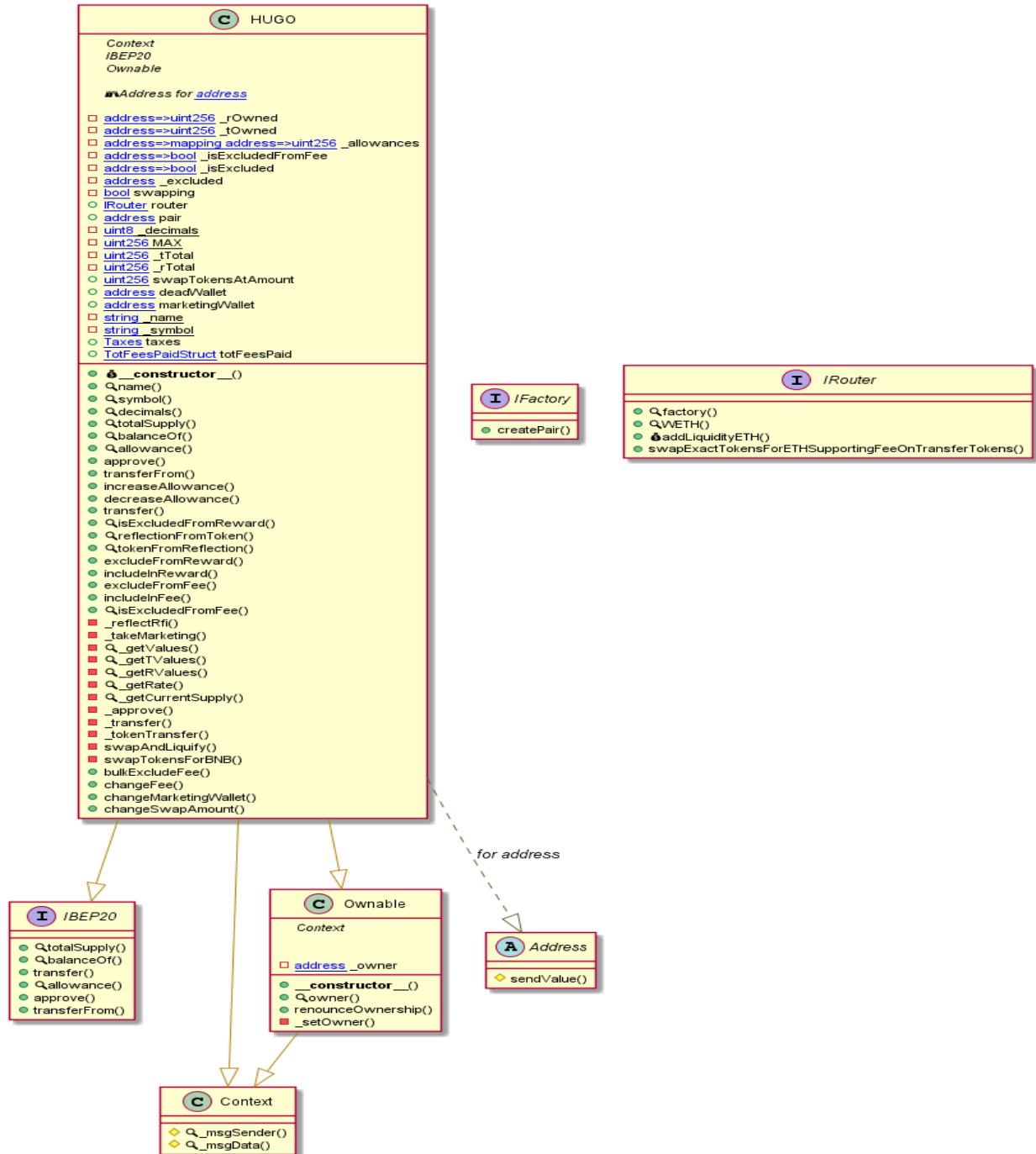
File : Hugo.sol
Contract: HUGO
Inherit: Context, IBEP20, Ownable
Observation: Passed
Test Report: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	name	public	Passed	All Passed	No Issue	Passed
2	symbol	public	Passed	All Passed	No Issue	Passed
3	decimals	public	Passed	All Passed	No Issue	Passed
4	totalSupply	public	Passed	All Passed	No Issue	Passed
5	balanceOf	public	Passed	All Passed	No Issue	Passed
6	allowance	public	Passed	All Passed	No Issue	Passed
7	approve	public	Passed	All Passed	No Issue	Passed
8	transferFrom	public	Passed	All Passed	No Issue	Passed
9	increaseAllowance	public	Passed	All Passed	No Issue	Passed
10	decreaseAllowance	public	Passed	All Passed	No Issue	Passed
11	transfer	public	Passed	All Passed	No Issue	Passed

12	isExcludedFromReward	public	Passed	All Passed	No Issue	Passed
13	reflectionFromToken	public	Passed	All Passed	No Issue	Passed
14	tokenFromReflection	public	Passed	All Passed	No Issue	Passed
15	excludeFromReward	public	Passed	All Passed	No Issue	Passed
16	includeInReward	external	Passed	All Passed	No Issue	Passed
17	excludeFromFee	public	Passed	All Passed	No Issue	Passed
18	includeInFee	public	Passed	All Passed	No Issue	Passed
19	isExcludedFromFee	public	Passed	All Passed	No Issue	Passed
20	_reflectRfi	private	Passed	All Passed	No Issue	Passed
21	_takeMarketing	private	Passed	All Passed	No Issue	Passed
22	_getValues	private	Passed	All Passed	No Issue	Passed
23	_getTValues	private	Passed	All Passed	No Issue	Passed
24	_getRValues	private	Passed	All Passed	No Issue	Passed
25	_getRate	private	Passed	All Passed	No Issue	Passed
26	_getCurrentSupply	private	Passed	All Passed	No Issue	Passed
27	_approve	private	Passed	All Passed	No Issue	Passed
28	_transfer	private	Passed	All Passed	No Issue	Passed
29	_tokenTransfer	private	Passed	All Passed	No Issue	Passed
30	swapAndLiquify	private	Passed	All Passed	No Issue	Passed
31	swapTokensForBNB	private	Passed	All Passed	No Issue	Passed

32	bulkExcludeFee	external	Passed	All Passed	No Issue	Passed
33	changeFee	public	Passed	All Passed	No Issue	Passed
34	changeMarketingWallet	public	Passed	All Passed	No Issue	Passed
35	changeSwapAmount	public	Passed	All Passed	No Issue	Passed

Code Flow Diagram - Hugo.sol



Code Flow Diagram - Slither Results Log

Hugo.sol

```
HUGO.allowance(address,address).owner (HUGO.sol#214) shadows:
- Ownable.owner() (HUGO.sol#50-52) (function)
HUGO._approve(address,address,uint256).owner (HUGO.sol#417) shadows:
- Ownable.owner() (HUGO.sol#50-52) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing

HUGO.changeSwapAmount(uint256) (HUGO.sol#528-530) should emit an event for:
- swapTokensAtAmount = _swapAmount (HUGO.sol#529)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-events-arithmetic

HUGO.constructor(address)._pair (HUGO.sol#175) lacks a zero-check on :
- pair = _pair (HUGO.sol#178)
HUGO.changeMarketingWallet(address)._marketingWallet (HUGO.sol#524) lacks a zero-check on :
- marketingWallet = _marketingWallet (HUGO.sol#525)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation

Reentrancy in HUGO.transfer(address,address,uint256) (HUGO.sol#427-454):
  External calls:
  - swapAndLiquify() (HUGO.sol#448)
    - (success) = recipient.call{value: amount}() (HUGO.sol#108)
    - address(marketingWallet).sendValue(deltaBalance) (HUGO.sol#490)
    - router.swapExactTokensForETHSupportingFeeOnTransferTokens(tokenAmount,0,path,address(this),block.timestamp) (HUGO.sol#504-510)
  External calls sending eth:
  - swapAndLiquify() (HUGO.sol#448)
    - (success) = recipient.call{value: amount}() (HUGO.sol#108)
  State variables written after the call(s):
  - _tokenTransfer(from,to,amount,takeFee) (HUGO.sol#453)
    - totFeesPaid.marketing += tMarketing (HUGO.sol#326)
    - totFeesPaid.rfi += tRfi (HUGO.sol#321)
Reentrancy in HUGO.transferFrom(address,address,uint256) (HUGO.sol#223-235):
  External calls:
  - _transfer(sender,recipient,amount) (HUGO.sol#228)
    - (success) = recipient.call{value: amount}() (HUGO.sol#108)
    - address(marketingWallet).sendValue(deltaBalance) (HUGO.sol#490)
    - router.swapExactTokensForETHSupportingFeeOnTransferTokens(tokenAmount,0,path,address(this),block.timestamp) (HUGO.sol#504-510)
  External calls sending eth:
  - swapAndLiquify() (HUGO.sol#448)
    - (success) = recipient.call{value: amount}() (HUGO.sol#108)
  Event emitted after the call(s):
  - Transfer(sender,recipient,s.tTransferAmount) (HUGO.sol#480)
  - _tokenTransfer(from,to,amount,takeFee) (HUGO.sol#453)
Reentrancy in HUGO.transferFrom(address,address,uint256) (HUGO.sol#223-235):
  External calls:
  - _transfer(sender,recipient,amount) (HUGO.sol#228)
    - (success) = recipient.call{value: amount}() (HUGO.sol#108)
    - address(marketingWallet).sendValue(deltaBalance) (HUGO.sol#490)
    - router.swapExactTokensForETHSupportingFeeOnTransferTokens(tokenAmount,0,path,address(this),block.timestamp) (HUGO.sol#504-510)
  External calls sending eth:
  - _transfer(sender,recipient,amount) (HUGO.sol#228)
    - (success) = recipient.call{value: amount}() (HUGO.sol#108)
  Event emitted after the call(s):
  - Approval(owner,spender,amount) (HUGO.sol#424)
  - _approve(sender,_msgSender(),currentAllowance - amount) (HUGO.sol#232)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3

HUGO.includeInReward(address) (HUGO.sol#294-305) has costly operations inside a loop:
- _excluded.pop() (HUGO.sol#301)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#costly-operations-inside-a-loop
```

```
Context._msgData() (HUGO.sol#35-38) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code

HUGO._rTotal (HUGO.sol#133) is set pre-construction with a non-constant function or state variable:
- (MAX - (MAX % _tTotal))
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#function-initializing-state

Pragma version^0.8.17 (HUGO.sol#6) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.1
6
solc-0.8.17 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Low level call in Address.sendValue(address,uint256) (HUGO.sol#105-110):
- (success) = recipient.call{value: amount}{} (HUGO.sol#108)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls

Function IRouter.WETH() (HUGO.sol#77) is not in mixedCase
Struct HUGO.valuesFromGetValues (HUGO.sol#157-165) is not in CapWords
Parameter HUGO.changeFee(HUGO.Taxes)._taxes (HUGO.sol#519) is not in mixedCase
Parameter HUGO.changeMarketingWallet(address)._marketWallet (HUGO.sol#524) is not in mixedCase
Parameter HUGO.changeSwapAmount(uint256)._swapAmount (HUGO.sol#528) is not in mixedCase
Constant HUGO._decimals (HUGO.sol#129) is not in UPPER_CASE_WITH_UNDERSCORES
Constant HUGO._name (HUGO.sol#140) is not in UPPER_CASE_WITH_UNDERSCORES
Constant HUGO._symbol (HUGO.sol#141) is not in UPPER_CASE_WITH_UNDERSCORES
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions

Redundant expression "this (HUGO.sol#36)" inContext (HUGO.sol#30-39)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements

HUGO.slitherConstructorVariables() (HUGO.sol#113-534) uses literals with too many digits:
- _tTotal = 420000000000000000 * 10 ** _decimals (HUGO.sol#132)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits

HUGO._tTotal (HUGO.sol#132) should be constant
HUGO.deadWallet (HUGO.sol#137) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant

HUGO.pair (HUGO.sol#127) should be immutable
HUGO.router (HUGO.sol#126) should be immutable
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-immutable
HUGO.sol analyzed (7 contracts with 84 detectors), 30 result(s) found
```

Solidity Static Analysis

Hugo.sol

Security

Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.

[more](#)

Pos: 509:12:

Low level calls:

Use of "call": should be avoided whenever possible. It can lead to unexpected behavior if return value is not handled properly. Please use Direct Calls via specifying the called contract's interface.

[more](#)

Pos: 108:27:

Gas & Economy

Gas costs:

Gas requirement of function HUGO.includeInReward is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)

Pos: 294:4:

For loop over dynamic array:

Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully. Due to the block gas limit, transactions can only consume a certain amount of gas. The number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. Additionally, using unbounded loops incurs in a lot of avoidable gas costs. Carefully test how many items at maximum you can pass to such functions to make it successful.

[more](#)

Pos: 296:8:

For loop over dynamic array:

Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully. Due to the block gas limit, transactions can only consume a certain amount of gas. The number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. Additionally, using unbounded loops incurs in a lot of avoidable gas costs. Carefully test how many items at maximum you can pass to such functions to make it successful.

[more](#)

Pos: 406:8:

For loop over dynamic array:

Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully. Due to the block gas limit, transactions can only consume a certain amount of gas. The number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. Additionally, using unbounded loops incurs in a lot of avoidable gas costs. Carefully test how many items at maximum you can pass to such functions to make it successful.

[more](#)

Pos: 514:8:

Miscellaneous

Similar variable names:

HUGO.swapTokensForBNB(uint256) : Variables have very similar names "pair" and "path". Note: Modifiers are currently not considered by this static analysis.

Pos: 507:12:

No return:

IRouter.addLiquidityETH(address,uint256,uint256,uint256,address,uint256):

Defines a return type but never explicitly returns a value.

Pos: 79:4:

Guard conditions:

Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.

[more](#)

Pos: 520:8:

Data truncated:

Division of integer values yields an integer value again. That means e.g. $10 / 100 = 0$ instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.

Pos: 412:22:

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:

No high severity vulnerabilities were found.

Medium:

No medium severity vulnerabilities were found.

Low:

No low severity vulnerabilities were found.

Very Low:

No very low severity vulnerabilities were found.

Conclusion

We were given a contract code in the form of a link <https://bscscan.com/address/0x28e3ff085f67163532a843fbf714178770a0210c#code>

and have used all possible tests based on the given object. So it is ready to go for production. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything.

The security state of the reviewed contract is "**well-secured**".

Note For Contract Users

Technical auditing does not guarantee the project's ethical side.

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

RD Auditors 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.



Email: info@rdauditors.com

Website: www.rdauditors.com

