



**RD
AUDITORS**

ZFM Coin, Contract, Code Review and Security Analysis Report

Customer: ZFM Coin
Prepared on: 12th October 2022
Platform: Binance
Language: Solidity

rdauditors.com

Table of Contents

Disclaimer	2
Document	3
Introduction	4
Project Scope	5
Executive Summary	6
Code Quality	6
Documentation	8
Use of Dependencies	8
AS-IS Overview	9
Code Flow Diagram - ZFM Coin	13
Code Flow Diagram - Slither Results Log	14
Severity Definitions	19
Audit Findings	20
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.

Document

Name	Smart Contract Code Review and Security Analysis Report of ZFM Coin
Platform	Binance/ Solidity
File	ZFM Coin.sol
MD5 hash	d41d8cd98f00b204e9800998ecf8427e
SHA256 hash	4b4d2b757b3af5b24776c617e128edd36ec9b62638ff5c60dfaa2d6dd57ec2ef
Date	12/10/2022

Introduction

RD Auditors (Consultant) were contracted by ZFM Coin (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 10th - 12th October 2022.

This contract consists of one file.

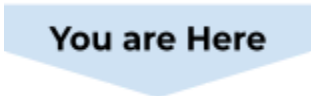
Project Scope

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 **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	3
 Critical	0
 High	0
 Medium	0
 Low	0
 Very Low	3

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 ZFM Coin team 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 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

We were given the ZFM Coin code as a link:

<https://bscscan.com/address/0xce6b8B2787C657f1b98b7A66B5B63178863fd719#code>

The hash of that file is mentioned in the table. As mentioned above, it's well commented 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

ZFMCOIN.sol

File And Function Level Report

File: ZFM Coin
Interface: IBEP20
Observation: Passed
Test Report: Passed

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

Contract: Context

Observation: Passed

Test Report: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	_msgSender	internal	Passed	All Passed	No Issue	Passed
2	_msgData	internal	Passed	All Passed	No Issue	Passed

Contract: Ownable

Inherit: Context

Observation: Passed

Test Report: Passed

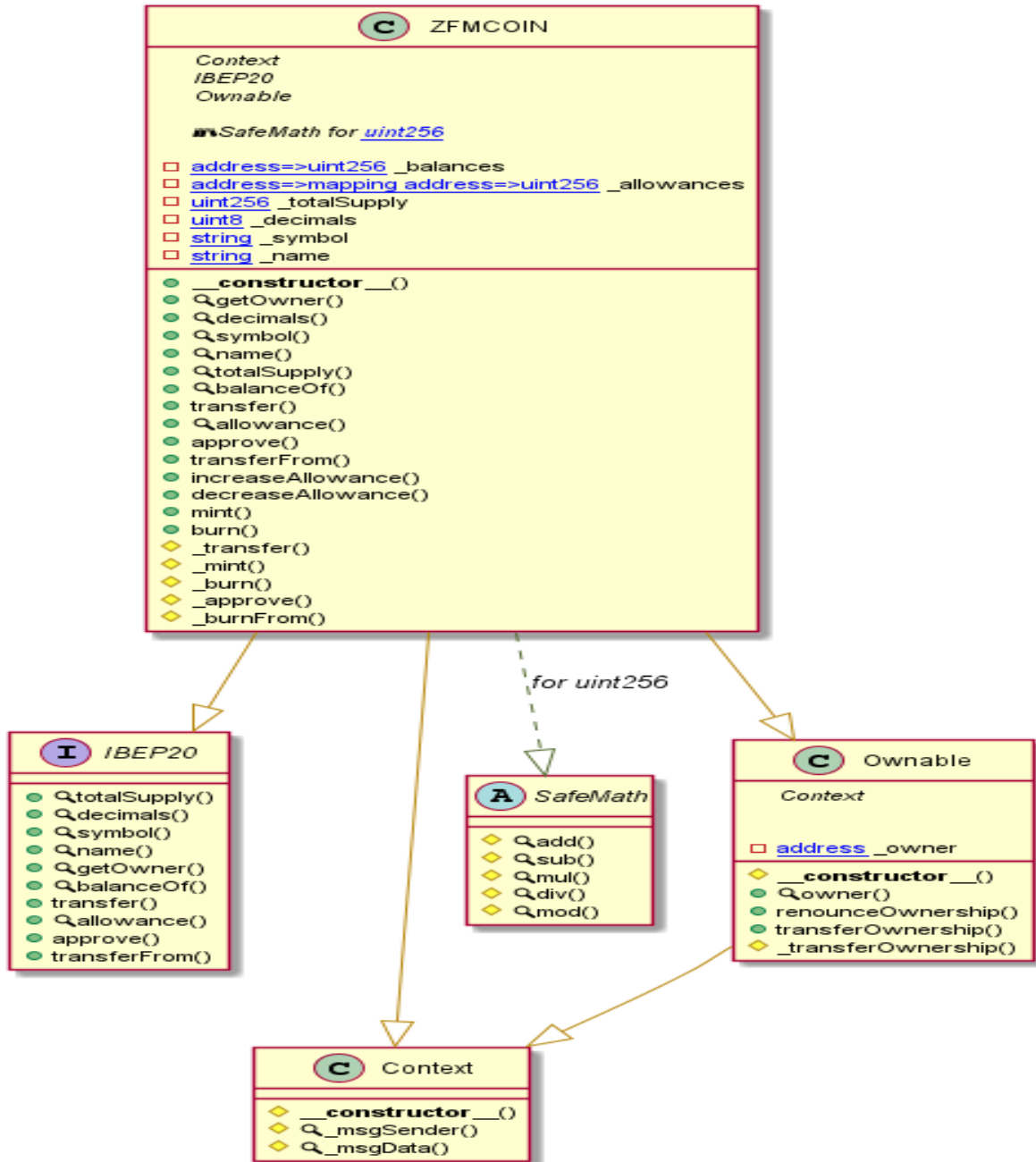
Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	Owner	read	Passed	All Passed	No Issue	Passed
2	renounceOwnership	onlyowner	Passed	All Passed	No Issue	Passed
3	transferOwnership	onlyowner	Passed	All Passed	No Issue	Passed
4	_transferOwnership	onlyowner	Passed	All Passed	No Issue	Passed

Contract: ZFMCoin
Inherit: Context, IBEP20, Ownable
Observation: Passed
Test Report: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	getOwner	read	Passed	All Passed	No Issue	Passed
2	decimals	read	Passed	All Passed	No Issue	Passed
3	symbol	read	Passed	All Passed	No Issue	Passed
4	name	read	Passed	All Passed	No Issue	Passed
5	totalSupply	read	Passed	All Passed	No Issue	Passed
6	balanceOf	read	Passed	All Passed	No Issue	Passed
7	transfer	read	Passed	All Passed	No Issue	Passed
8	allowance	read	Passed	All Passed	No Issue	Passed
9	approve	write	Passed	All Passed	No Issue	Passed
10	transferFrom	write	Passed	All Passed	No Issue	Passed
11	increaseAllowance	write	Passed	All Passed	No Issue	Passed
12	decreaseAllowance	write	Passed	All Passed	No Issue	Passed
13	mint	onlyowner	Passed	All Passed	No Issue	Passed
14	burn	write	Passed	All Passed	No Issue	Passed
15	_transfer	internal	Passed	All Passed	No Issue	Passed
16	_mint	internal	Passed	All Passed	No Issue	Passed
17	_burn	internal	Passed	All Passed	No Issue	Passed

18	_approve	internal	Passed	All Passed	No Issue	Passed
19	burnFrom	internal	Passed	All Passed	No Issue	Passed

Code Flow Diagram - ZFM Coin



Code Flow Diagram - Slither Results Log

```
INFO:Detectors:
ZFMCOIN.allowance(address,address).owner (ZFMCOIN.sol#421) shadows:
  - Ownable.owner() (ZFMCOIN.sol#299-301) (function)
ZFMCOIN._approve(address,address,uint256).owner (ZFMCOIN.sol#581) shadows:
  - Ownable.owner() (ZFMCOIN.sol#299-301) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
Context._msgData() (ZFMCOIN.sol#115-118) is never used and should be removed
SafeMath.div(uint256,uint256) (ZFMCOIN.sol#214-216) is never used and should be removed
SafeMath.div(uint256,uint256,string) (ZFMCOIN.sol#229-236) is never used and should be removed
SafeMath.mod(uint256,uint256) (ZFMCOIN.sol#249-251) is never used and should be removed
SafeMath.mod(uint256,uint256,string) (ZFMCOIN.sol#264-267) is never used and should be removed
SafeMath.mul(uint256,uint256) (ZFMCOIN.sol#189-201) is never used and should be removed
ZFMCOIN._burnFrom(address,uint256) (ZFMCOIN.sol#595-598) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Detectors:
Redundant expression "this (ZFMCOIN.sol#116)" inContext (ZFMCOIN.sol#106-119)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
INFO:Detectors:
ZFMCOIN.constructor() (ZFMCOIN.sol#353-361) uses literals with too many digits:
  - totalSupply = 1000000000000000000000000000000 (ZFMCOIN.sol#357)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
```

```
INFO:Detectors:
renounceOwnership() should be declared external:
  - Ownable.renounceOwnership() (ZFMCOIN.sol#318-321)
transferOwnership(address) should be declared external:
  - Ownable.transferOwnership(address) (ZFMCOIN.sol#327-329)
increaseAllowance(address,uint256) should be declared external:
  - ZFMCOIN.increaseAllowance(address,uint256) (ZFMCOIN.sol#467-470)
decreaseAllowance(address,uint256) should be declared external:
  - ZFMCOIN.decreaseAllowance(address,uint256) (ZFMCOIN.sol#486-489)
mint(uint256) should be declared external:
  - ZFMCOIN.mint(uint256) (ZFMCOIN.sol#499-502)
burn(uint256) should be declared external:
  - ZFMCOIN.burn(uint256) (ZFMCOIN.sol#504-507)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:ZFMCOIN.sol analyzed (5 contracts with 75 detectors), 17 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Solidity Static Analysis

Gas costs:

Gas requirement of function ZFMCOIN.mint 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: 501:2:

Gas costs:

Gas requirement of function ZFMCOIN.burn 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: 506:2:

ERC

ERC20:

ERC20 contract's "decimals" function should have "uint8" as return type

[more](#)

Pos: 16:2:

ERC20:

ERC20 contract's "decimals" function should have "uint8" as return type

[more](#)

Pos: 375:2:

Constant/View/Pure functions:

SafeMath.div(uint256,uint256) : Is constant but potentially should not be. Note: Modifiers are currently not considered by this static analysis.

[more](#)

Pos: 216:2:

Constant/View/Pure functions:

SafeMath.mod(uint256,uint256) : Is constant but potentially should not be. Note: Modifiers are currently not considered by this static analysis.

[more](#)

Pos: 251:2:

Similar variable names:

ZFMCOIN._burn(address,uint256) : Variables have very similar names "account" and "amount".

Note: Modifiers are currently not considered by this static analysis.

Pos: 563:12:

Similar variable names:

ZFMCOIN._burn(address,uint256) : Variables have very similar names "account" and "amount".

Note: Modifiers are currently not considered by this static analysis.

Pos: 565:14:

Similar variable names:

ZFMCOIN._burnFrom(address,uint256) : Variables have very similar names "account" and "amount". Note: Modifiers are currently not considered by this static analysis.

Pos: 599:48:

Similar variable names:

ZFMCOIN._burnFrom(address,uint256) : Variables have very similar names "account" and "amount". Note: Modifiers are currently not considered by this static analysis.

Pos: 599:75:

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: 584:4:

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: 585:4:

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: 200:12:

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: 234:16:

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:

1) Use latest solidity version

Description - Using the latest solidity will prevent any compiler level bugs.

Suggestion - We suggest using version $\geq 0.8.0$.

2) Make variable constant

```
350     uint256 private _totalSupply;  
351     uint8 private _decimals;  
352     string private _symbol;  
353     string private _name;  
354
```

Description - These above variables will be unchanged, so make it constant. It will save some gas.

Suggestion - Declare those variables as constant, just put a constant keyword.

3) Unlimited minting

```
501     function mint(uint256 amount) public onlyOwner returns (bool) {  
502         _mint(_msgSender(), amount);  
503         return true;  
504     }
```

Description - Owner can mint unlimited tokens if it is part of plan then ignore this point.

Suggestion - There should be an upper cap for minting.

Conclusion

We were given a contract file and have used all possible tests based on the given object. So it is now ready for mainnet deployment. 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.

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

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

