



QuillAudits



Audit Report
June, 2021

 ZORT

Contents

Introduction	01
Audit Goals	02
Issue Categories	03
Manual Audit	04
Automated Testing	09
Summary	15
Disclaimer	16

Introduction

This audit report highlights the overall security of the ZortCoin Token Contract. With this report, QuillAudits have tried to ensure the reliability of the smart contract by completing the assessment of their system's architecture and smart contract codebase.

Auditing Approach and Methodologies applied

In this audit, we consider the following crucial features of the code.

- Whether the implementation of token standards.
- Whether the code is secure.
- Whether the code meets the best coding practices.
- Whether the code meets the SWC Registry issue.

The audit has been performed according to the following procedure:

Manual Audit

- Inspecting the code line by line and revert the initial algorithms of the protocol and then compare them with the specification
- Manually analyzing the code for security vulnerabilities.
- Gas Consumption and optimisation
- Assessing the overall project structure, complexity & quality.
- Checking SWC Registry issues in the code.
- Unit testing by writing custom unit testing for each function.
- Checking whether all the libraries used in the code of the latest version.
- Analysis of security on-chain data.
- Analysis of the failure preparations to check how the smart contract performs in case of bugs and vulnerability.

Automated analysis

- Scanning the project's code base with Mythril, Slither, Echidna, Manticore, others.
- Manually verifying (reject or confirm) all the issues found by tools.
- Performing Unit testing.
- Manual Security Testing (SWC-Registry, Overflow)
- Running the tests and checking their coverage.

Audit Details

Project Name: Zort Coin Token

Token symbol: ZORT

Code-Link: <https://etherscan.io/address/0x825cd4201f8a2bbb1a69668eac4e5fa71283273d#code>

Languages: Solidity

Platforms and Tools: HardHat, Remix, VScode, solhint and other tools mentioned in the automated analysis section.

Audit Goals

The focus of this audit was to verify whether the smart contract is secure, resilient, and working according to ERC20 specs. The audit activity can be grouped into three categories.

Security

Identifying security related issues within each contract and the system of contract.

Sound Architecture

Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.

Code Correctness and Quality

A full review of the contract source code. The primary areas of focus include:

- Correctness.
- Section of code with high complexity.
- Readability.
- Quantity and quality of test coverage.

Issue Categories

Every issue in this report was assigned a severity level from the following:

High severity issues

Issues on this level are critical to the smart contract’s performance/ functionality and should be fixed before moving to a live environment.

Medium severity issues

Issues on this level could potentially bring problems and should eventually be fixed.

Low severity issues

Issues on this level are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Number of issues per severity

	High	Medium	Low	Informational
Open	0	0	0	0
Closed	0	0	3	3

Manual Audit

SWC Registry test

We have tested SWC registry issues as well. Out of all tests, three issues were found, which are of low priority. We have put the details about it below in the low priority section.

Serial No.	Description	Comments
SWC-132	Unexpected Ether balance	Pass: Avoided strict equality checks for the Ether balance in a contract
SWC-131	Presence of unused variables	Pass: No unused variables
SWC-128	DoS With Block Gas Limit	Pass
SWC-122	Lack of Proper Signature Verification	Pass
SWC-120	Weak Sources of Randomness from Chain Attributes	Pass
SWC-119	Shadowing State Variables	Pass: No ambiguity found.
SWC-118	Incorrect Constructor Name	Pass. No incorrect constructor name used
SWC-116	Timestamp Dependence	Pass
SWC-115	Authorization through tx.origin	Pass: No tx.origin found
SWC-114	Transaction Order Dependence	Pass

Serial No.	Description	Comments
<u>SWC-113</u>	DoS with Failed Call	Pass: No failed call
<u>SWC-112</u>	Delegatecall to Untrusted Callee	Pass
<u>SWC-111</u>	Use of Deprecated Solidity Functions	Pass : No deprecated function used
<u>SWC-108</u>	State Variable Default Visibility	Pass: Explicitly defined visibility for all state variables
<u>SWC-107</u>	Reentrancy	Found
<u>SWC-106</u>	Unprotected SELF-DESTRUCT Instruction	Pass: Not found any such vulnerability
<u>SWC-104</u>	Unchecked Call Return Value	Pass: Not found any such vulnerability
<u>SWC-103</u>	Floating Pragma	Found
<u>SWC-102</u>	Outdated Compiler Version	Found: Latest version is Version 0.8.4 In code 0.5.17 is used
<u>SWC-101</u>	Integer Overflow and Underflow	Pass

High level severity issues

No issues found

Medium level severity issues

No issues found

Low level severity issues

1. Description → SWC 102: Outdated Compiler Version [Line no. 5, 7]

```
1  /**
2   *Submitted for verification at Etherscan.io on 2021-05-24
3   */
4
5  pragma solidity ^0.5.17;
6
7  pragma solidity >=0.4.22 <0.6.0;
8
```

Using an outdated compiler version can be problematic, especially if there are publicly disclosed bugs and issues that affect the current compiler version.

Also, there are two different compiler versions used. It's better to use only a single compiler version.

Remediation

It is recommended to use a recent version of the Solidity compiler, which is Version 0.8.4.

Also, use only a single compiler version instead of two.

Status: Closed

2. Description: Using the approve function of the token standard [Line 108-113]

```
106  * @param _value the max amount they can spend
107  */
108  function approve(address spender, uint256 value) public
109  returns (bool success) {
110      allowance[msg.sender][spender] = _value;
111      emit Approval(msg.sender, spender, _value);
112      return true;
113  }
114
```

The **approve** function of ERC-20 is vulnerable. Using a front-running attack, one can spend approved tokens before the change of allowance value.

To prevent attack vectors described above, clients should make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. However, the contract itself shouldn't enforce it to allow backward compatibility with contracts deployed before.

Detailed reading around it can be found at [EIP 20](#)

Status: Acknowledged by the auditee.

3. Description: Prefer external to public visibility level [Line 79, 93, 124-125, 141, 157]

```
122  * @param _extraData some extra information to send to the approved contract
123  */
124  function approveAndCall(address _spender, uint256 _value, bytes memory _extraData)
125  public
126  returns (bool success) {
127      tokenRecipient spender = tokenRecipient(_spender);
128      if (approve(_spender, _value)) {
129          spender.receiveApproval(msg.sender, _value, address(this), _extraData);
130          return true;
131      }
132  }
133
```

A function with a **public** visibility modifier that is not called internally. Changing the visibility level to **external** increases code readability. Moreover, in many cases, functions with **external** visibility modifiers spend less gas compared to functions with **public** visibility modifiers.

The function definition of **renounceOwnership()**, **transferOwnership()**, **increaseAllowance()** and **decreaseAllowance()** is marked as "public". However, it is never directly called by another function in the same contract or any of its descendants. Consider marking it as "external" instead.

Recommendations: Use the **external** visibility modifier for functions never called from the contract via internal call. [Reading Link](#).

Note: Exact same issue was found while using automated testing by Mythx and Slither.

Status: Acknowledged by the auditee.

Informational

1. Requirement Violation (SWC 123), Improper Following of Specification by Caller [Line 129,168]

Description: The Solidity `require()` construct is meant to validate external inputs of a function. In most cases, such external inputs are provided by callers, but they may also be returned by callees. In the former case, we refer to them as precondition violations. Violations of a requirement can indicate one of two possible issues:

- A bug exists in the contract that provided the external input.
- The condition used to express the requirement is too strong.

Recommendations: If the required logical condition is too strong, it should be weakened to allow all valid external inputs.

Otherwise, the bug must be in the contract that provided the external input and one should consider fixing its code by making sure no invalid inputs are provided.

Relevant sources to read: [Link](#)

Status: Acknowledged by the auditee.

2. Missing License

Developers need to use a license according to your project. The suggestion to use here would be an SPDX license. You can find a list of licenses here: <https://spdx.org/licenses/>

The SPDX License List is an integral part of the SPDX Specification. The SPDX License List itself is a list of commonly found licenses and exceptions used in free and open or collaborative software, data, hardware, or documentation. The SPDX License List includes a standardized short identifier, the full name, the license text, and a canonical permanent URL for each license and exception.

Status: Acknowledged by the auditee.

3. Open Zeppelin standards

Follow [Open Zeppelin standard](#) for token contract making. There were few functions like Pausable and few others which are usually put in token contracts in general as best practice.

Status: Acknowledged by the auditee.

Automated Testing

We have used multiple automated testing frameworks. This makes code more secure common attacks. The results are below.

Slither

Slither is a Solidity static analysis framework that runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses. Slither enables developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom analyses. After running Slither, we got the results below.

```
INFO:Detectors:
Different versions of Solidity is used in :
  - Version used: ['>=0.4.22<0.6.0', '^0.5.17']
  - ^0.5.17 (zort.sol#5)
  - >=0.4.22<0.6.0 (zort.sol#7)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
INFO:Detectors:
Pragma version>=0.4.22<0.6.0 (zort.sol#7) allows old versions
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Contract tokenRecipient (zort.sol#9-11) is not in CapWords
Parameter TokenERC20.transfer(address,uint256)._to (zort.sol#79) is not in mixedCase
Parameter TokenERC20.transfer(address,uint256)._value (zort.sol#79) is not in mixedCase
Parameter TokenERC20.transferFrom(address,address,uint256)._from (zort.sol#93) is not in mixedCase
Parameter TokenERC20.transferFrom(address,address,uint256)._to (zort.sol#93) is not in mixedCase
Parameter TokenERC20.transferFrom(address,address,uint256)._value (zort.sol#93) is not in mixedCase
Parameter TokenERC20.approve(address,uint256)._spender (zort.sol#108) is not in mixedCase
Parameter TokenERC20.approve(address,uint256)._value (zort.sol#108) is not in mixedCase
Parameter TokenERC20.approveAndCall(address,uint256,bytes)._spender (zort.sol#124) is not in mixedCase
Parameter TokenERC20.approveAndCall(address,uint256,bytes)._value (zort.sol#124) is not in mixedCase
Parameter TokenERC20.approveAndCall(address,uint256,bytes)._extraData (zort.sol#124) is not in mixedCase
Parameter TokenERC20.burn(uint256)._value (zort.sol#141) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformity-to-solidity-naming-conventions
INFO:Detectors:
ZortCoin.constructor() (zort.sol#170-172) uses literals with too many digits:
  - TokenERC20(10000000000,Zort Coin,ZORT) (zort.sol#170)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
INFO:Detectors:
TokenERC20.decimals (zort.sol#17) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant
INFO:Detectors:
transfer(address,uint256) should be declared external:
  - TokenERC20.transfer(address,uint256) (zort.sol#79-82)
transferFrom(address,address,uint256) should be declared external:
  - TokenERC20.transferFrom(address,address,uint256) (zort.sol#93-98)
approveAndCall(address,uint256,bytes) should be declared external:
  - TokenERC20.approveAndCall(address,uint256,bytes) (zort.sol#124-132)
burn(uint256) should be declared external:
  - TokenERC20.burn(uint256) (zort.sol#141-147)
burnFrom(address,uint256) should be declared external:
  - TokenERC20.burnFrom(address,uint256) (zort.sol#157-165)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:zort.sol analyzed (3 contracts with 46 detectors), 23 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

There were some issues found by Slither. Those are listed below.

Description 1

Functions like transfer, transferFrom, approveAndCall, burn, burnFrom are declared as internal, whereas they should be marked as external. This issue has already been discussed in manual analysis. [Reading link](#).

Description 2

Uses literals with too many digits:

ZortCoin.constructor() (zort.sol#170-172) uses literals with too many digits:

- TokenERC20(10000000000,Zort Coin,ZORT) (zort.sol#170)

Detailed [reading link](#).

Description 3: Use of constant

TokenERC20.decimals (zort.sol#17) should be constant

Reference: [link](#)

Description 4: Proper naming convention

Contract tokenRecipient (zort.sol#9-11) is not in CapWords

Parameter TokenERC20.transfer(address,uint256)._to (zort.sol#79) is not in mixedCase

Parameter TokenERC20.transfer(address,uint256)._value (zort.sol#79) is not in mixedCase

Parameter TokenERC20.transferFrom(address,address,uint256)._from (zort.sol#93) is not in mixedCase

Parameter TokenERC20.transferFrom(address,address,uint256)._to (zort.sol#93) is not in mixedCase

Parameter TokenERC20.transferFrom(address,address,uint256)._value (zort.sol#93) is not in mixedCase

Parameter TokenERC20.approve(address,uint256)._spender (zort.sol#108) is not in mixedCase

Parameter TokenERC20.approve(address,uint256)._value (zort.sol#108) is not in mixedCase

Parameter TokenERC20.approveAndCall(address,uint256,bytes)._spender (zort.sol#124) is not in mixedCase

Parameter TokenERC20.approveAndCall(address,uint256,bytes)._value (zort.sol#124) is not in mixedCase

Parameter TokenERC20.approveAndCall(address,uint256,bytes)._extraData (zort.sol#124) is not in mixedCase

Parameter TokenERC20.burn(uint256)._value (zort.sol#141) is not in mixedCase

Parameter TokenERC20.burnFrom(address,uint256)._from (zort.sol#157) is not in mixedCase

Parameter TokenERC20.burnFrom(address,uint256)._value (zort.sol#157) is not in mixedCase

Reference: [Link](#)

Manticore

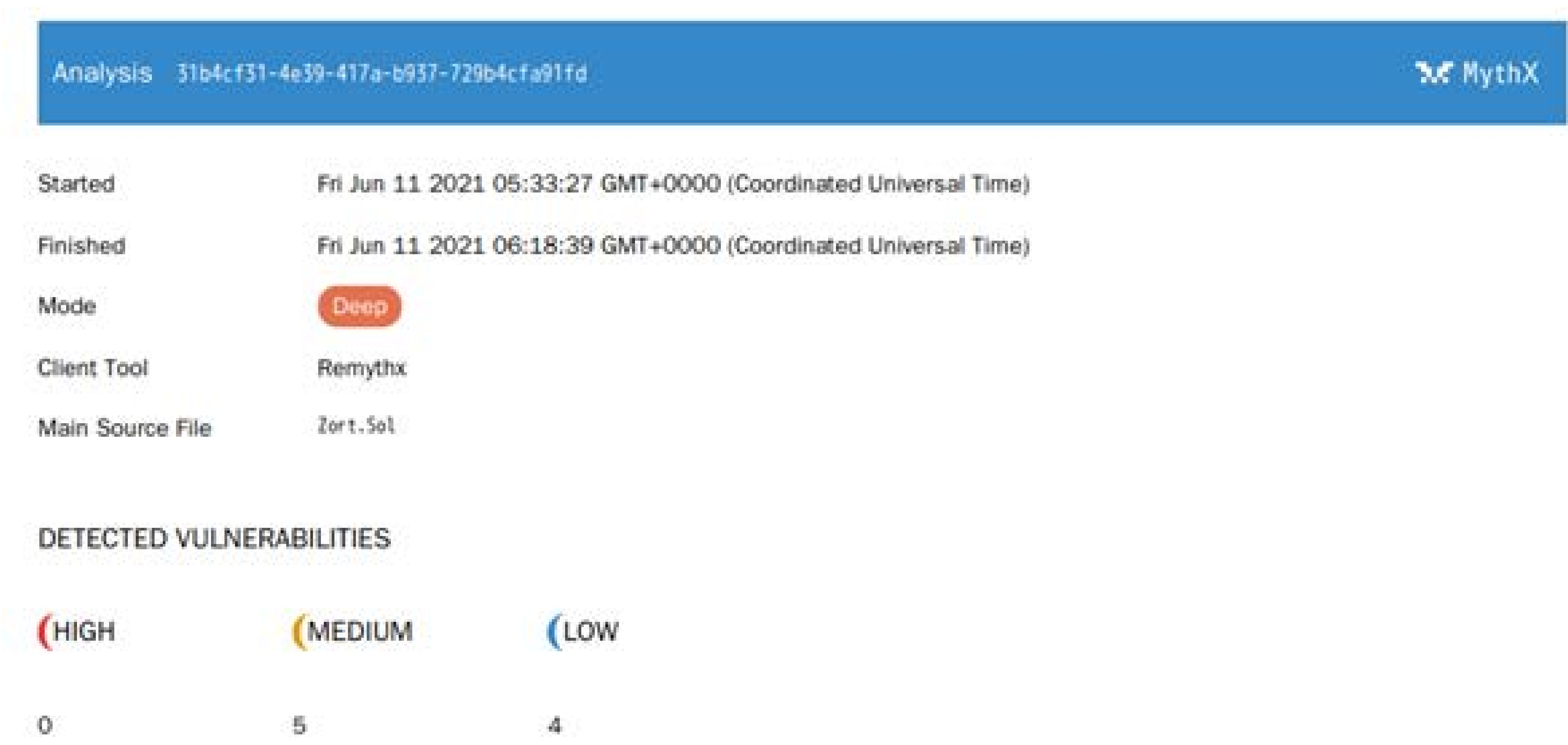
Manticore is a symbolic execution tool for the analysis of smart contracts and binaries. It executes a program with symbolic inputs and explores all the possible states it can reach. It also detects crashes and other failure cases in binaries and smart contracts.

Manticore results throw the same warning which is similar to the Slither warning.

Mythx

MythX is a security analysis tool and API that performs static analysis, dynamic analysis, symbolic execution, and fuzzing on Ethereum smart contracts. MythX checks for and reports on the common security vulnerabilities in open industry-standard SWC Registry.

We have separately put them for analysis. Below are the reports generated for each contract separately.



Most of the vulnerabilities generated by Mythx are discussed in the Manual sections. Mythx Report contains both Medium and low issues only. The pdf copy of the report can be found at this [link](#).

Anchain:

Anchain sandbox audits the security score of any Solidity-based smart contract, having analyzed the source code of every mainnet EVM smart contract plus the 1M + unique, user-uploaded smart contracts. Code has been analyzed there and got the report below.

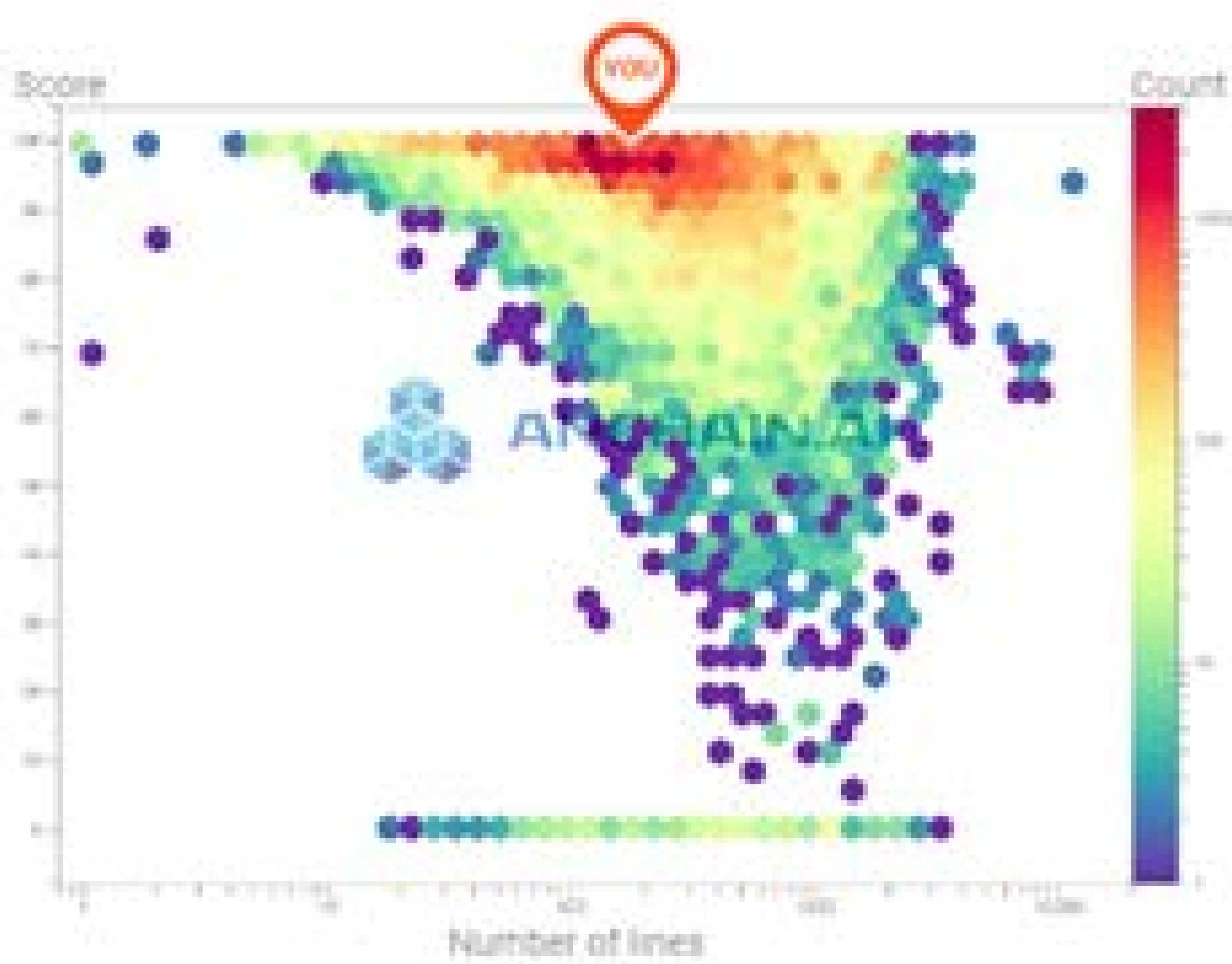
ETH Smart Contract Audit Report

MD5:d31beaa051786a4ec8ea77d6be548499

Runtime:7.7s

Scored higher than
100% of similar code
amongst 50k smart contracts
audited by Anchain.AI.

Score
100
Threat Level
Low
Number of lines
173



Overview

Code Class	EVM Coverage
NASA	2.6%
ZortCoin	2.6%

0 Vulnerabilities Found








 High Risk  Medium Risk  Low Risk

Recommendations








No information is available in this section

Vulnerability Checklist

NASA

-  Integer Underflow
-  Integer Overflow
-  Parity Multisig Bug
-  Callstack Depth Attack
-  Transaction-Ordering Dependency
-  Timestamp Dependency
-  Re-Entrancy

ZortCoin

-  Integer Underflow
-  Integer Overflow
-  Parity Multisig Bug
-  Callstack Depth Attack
-  Transaction-Ordering Dependency
-  Timestamp Dependency
-  Re-Entrancy

There were 0 vulnerabilities found by Anchain.

Disclaimer

The audit does not give any warranties on the security of the code. One audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the code. Besides a security audit, please don't consider this report as investment advice.

Summary

The use of smart contracts is simple, and the code is relatively small. Altogether the code is written and demonstrates effective use of abstraction, separation of concern, and modularity. But there are a few issues/vulnerabilities to be tackled at various security levels; it is recommended to fix them before deploying the contract on the main network. Given the subjective nature of some assessments, it will be up to the ZortCoin team to decide whether any changes should be made.



ZORT



QuillAudits



Canada, India, Singapore and United Kingdom



audits.quillhash.com



audits@quillhash.com