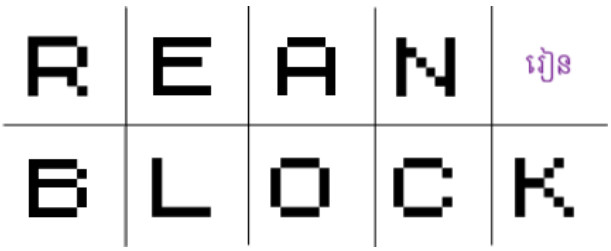

Shisha Coin TRC20 Audit Report

Version 1.0

reanblock.com

1st July 2024

Shisha Coin TRC20 Audit Report

reanblock.com

1st July 2024

Prepared by: Reanblock Lead Security Researcher: - Darren Jensen

Table of Contents

See table

Introduction

This document outlines the findings for smart contract code review for Shisha Coin TRC20 created on the TRON blockchain at this address.

Overview

Project Name	Shisha Coin (TRC20)
Repository	N/A (verified code provided onchain)
Commit SHA	N/A (verified code provided via TRON Block Explorer)
Documentation	N/A
Methods	Manual review & CLI review (Slither, Aderyn, Solhint)

Summary

The Shisha Coin (TRC20) is a standard TRC20 token for deployment on TRON blockchain. It uses standard TRC20 functions and has the following default settings:

Name	`Shisha Coin`
Symbol	`SHISHA`
Total Supply	`69,420,000 SHISHA`

Notes

- The total supply is **minted to the designated holder** which was supplied as the `holder` address during deployment.
- There are no `mint` or `burn` functions included in the contract.

Test approach and methodology

Auditor performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).

- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions Slither & Aderyn.
- PoC and testing (Foundry).

Risk Methodology

Every vulnerability and issue observed is ranked based on two sets of Metrics and a Severity Coefficient. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two Metric sets are: **Exploitability** and **Impact**. Exploitability captures the ease and technical means by which vulnerabilities can be exploited and Impact describes the consequences of a successful exploit.

The Severity Coefficients is designed to further refine the accuracy of the ranking with two factors: Reversibility and Scope. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

Exploitability

Attack Origin (AO)

Captures whether the attack requires compromising a specific account.

Attack Cost (AC)

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX)

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situations, available third-party liquidity and regulatory challenges.

Metrics

Exploitability Metric (m_e)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A)	1
	Specific (AO:S)	0.2
Attack Cost (AC)	Low (AC:L)	1
	Medium (AC:M)	0.67
	High (AC:H)	0.33
Attack Complexity (AX)	Low (AX:L)	1
	Medium (AX:M)	0.67
	High (AX:H)	0.33

Exploitability E is calculated using the following formula:

E = $\prod m_e$

Impact

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_i)	Metric Value	Numerical Value
Confidentiality (C)	None (C:N)	0
	Low (C:L)	0.25
	Medium (C:M)	0.5
	High (C:H)	0.75
	Critical (C:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical (A:C)	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75

Impact Metric (m_i)	Metric Value	Numerical Value
	Critical (Y:H)	1

Impact I is calculated using the following formula:

I = max(m_i) + $\frac{\sum m_i - max(m_i)}{4}$

Severity Coefficient

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts re- sources in other contracts.

Metrics:

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient C is obtained by the following product:

C=rs

The Vulnerability Severity Score S is obtained by:

S=min(10, EIC * 10)

The score is rounded up to 1 decimal place.

Severity	Score Value Range
Critical	9 -10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

Actions required by severity level

- **Critical** - client must fix the issue.
- **High** - client must fix the issue.
- **Medium** - client should fix the issue.
- **Low** - client could fix the issue.
- **Informational** - client could consider design/UX related decision
- **Recommendation** - client could have an internal team discussion on whether the recommendations provide any UX or security enhancement and if it is technically and economically feasible to implement the recommendations

Actions required by severity level

- **High** - client must fix the issue.
- **Medium** - client should fix the issue.
- **Low** - client could fix the issue.
- **Informational** - client could consider design/UX related decision
- **Recommendation** - client could have an internal team discussion on whether the recommendations provide any UX or security enhancement and if it is technically and economically feasible to implement the recommendations
- **Gas Findings** - client could consider implementing suggestions for better UX

Audit Details

Scope

In this report the auditor has focused on all contracts found in the deployment address (TYJxozCVMUiHg3TbQp9PKeuXdTsX9ugJz9) via the TRON Block Explorer as follows:

```
|- SafeMath.sol
|- ShishaToken.sol
|- TRC20.sol
```

Roles

There were no access roles, ownership or admin control found in the contracts.

White Paper - Specifications Document

There was no White Paper or technical specifications document provided.

LoC (Lines of Code)

The `cloc` utility was used to determine the lines of code under review. The utility excludes empty lines and comments to leave a count of auditable lines of code in each contract. Since all contracts in scope are under the src folder the cloc command was run like so:

```
cloc --md  src
```

The results were as follows:

Language	files	blank	comment	code
Solidity	3	28	3	111
———	———	———	———	———
SUM:	3	28	3	111

SHA-256 File Fingerprints

To generate the SHA-256 fingerprint for all the smart contracts in a directory run:

```
shasum -a 256 src/*.sol
```

Which outputs the following:

```
0173f4f5f74cb67f2d54bde25f8afce1fd97c3d793210c94ef91873659d04b7e  SafeMath.sol
f418d09f937535ede891a8718851347920b7a6ba2e606385117989ab3d5d10b5  ShishaToken.sol
e707249d6b9066498cdb5d3f7d0f60b4dc0e3c9a2c8a412cb329ce80566d3ef0  TRC20.sol
```

Test Run

The project has foundry tests and all tests are passing:

```
Running 5 tests for test/SafeMath.t.sol:SafeMathTest
[PASS] test_Add() (gas: 9782)
[PASS] test_Div() (gas: 9821)
[PASS] test_Mod() (gas: 11353)
[PASS] test_Mul() (gas: 10013)
[PASS] test_Sub() (gas: 9887)
Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 3.64ms

Running 13 tests for test/ShishaToken.t.sol:ShishaTokenTest
[PASS] testApprove() (gas: 34598)
[PASS] testDescreaseAllowance() (gas: 40395)
[PASS] testFailApproveFromZeroAddress() (gas: 8551)
[PASS] testFailApproveToZeroAddress() (gas: 5712)
[PASS] testFailTransferFromInsufficientApprove() (gas: 36161)
[PASS] testFailTransferFromInsufficientBalance() (gas: 36208)
[PASS] testFailTransferFromZeroAddress() (gas: 8683)
[PASS] testFailTransferInsufficientBalance() (gas: 10948)
[PASS] testFailTransferToZeroAddress() (gas: 8714)
[PASS] testIncreaseAllowance() (gas: 36596)
[PASS] testTransfer() (gas: 68082)
[PASS] testTransferFrom() (gas: 97703)
[PASS] test_tokenDeployed() (gas: 28704)
Test result: ok. 13 passed; 0 failed; 0 skipped; finished in 4.06ms

Ran 2 test suites: 18 tests passed, 0 failed, 0 skipped (18 total tests)
```

Test Coverage

The project has nearly 100% test coverage.

Linting

Linting is a valuable tool for finding potential issues in smart contract code. It can find stylistic errors, violations of programming conventions, and unsafe constructs in your code. There are many great linters available, such as Solhint. Linting can help find potential problems, even security problems such as re-entrancy vulnerabilities before they become costly mistakes.

After installation, Solhint can be run via the terminal as follows.

```
solhint 'src/*.sol'
```

This command will run `solhint` for the contracts in the root of the project directory. The auditor has executed this command and there were no issues found.

Slither Security Analysis

Auditor uses Slither (version 0.9.6) static analyzer tool. The slither cli was run against all contracts in scope of the project.

For each of the above contracts the correct version of `solc` using `solc-select` and then run the `slither` command to analyze specific contract under test:

```
solc-select install 0.8.19

solc-select use 0.8.19

slither --checklist src 2>&1 | tee 25-05-2024-slither-report.md
```

The findings were checked and, if appropriate, included in the Executive Summary of this report. The final Slither report is also available as a separate file upon request. A summary is provided in the appendix here.

Aderyn Security Analysis

Auditor uses Aderyn (version 0.0.10) static analyzer tool. The `aderyn` cli was run against all contracts in scope of the project.

Run the `aderyn` command to analyze specific contracts under test as follows:

```
aderyn .
```

The findings were checked and, if appropriate, included in the Executive Summary of this report. The final Aderyn report is also available as a separate file upon request. A summary is provided in the appendix here.

Executive Summary

Assessment Summary and Findings Overview

Critical	High	Medium	Low	Informational	Gas
0	0	0	1	3	3

Findings

Medium

Low

[L-1] Solidity pragma should be specific not wide

Description:

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

Impact:

May lead to inconsistencies if the compiler version is not specific. For example if the contracts are compiled using `0.8.20` then the `PUSH0` opcode will be included in the bytecode which is not currently supported in TRON. This will cause the deployment to fail.

Code Location:

```
1 pragma solidity ^0.8.0;
```

Recommended Mitigation:

Use a specific version of solidity in the `pragma` statement.

Informational

[I-1] TRC20 is missing mint and burn functions

Description:

If the protocol intends to increase / decrease supply over the lifetime of the token deployment then it will need `mint` and `burn` functionality. Note since the white paper or technical specifications have not been provided this is just an informational issue.

Impact:

Without `mint` and `burn` the token supply cannot be controlled.

Recommended Mitigation:

Include `mint` and `burn` functions if required.

[I-2] Token contracts are not upgradable

Description:

The Shisha Coin is not upgradable. If the protocol intends to make functional changes to the token at a later date then it will need to be an upgradable contract. Note since the white paper or technical specifications have not been provided this is just an informational issue.

Impact:

The token cannot be upgraded.

Recommended Mitigation:

Make use of OpenZeppelin proxy upgradable patterns to enable upgradability of the protocol if required.

[I-3] Event is missing `indexed` fields

Description:

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields).

Impact:

Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

Code Location:

```
1 event Transfer(address indexed from, address indexed to, uint256 value)
      ;
```

Recommended Mitigation:

Consider indexing all possible event fields.

Gas

[G-1] Use of `SafeMath` library is not required for solidity `0_8_x`

Description:

The use of `SafeMath` library is not required for solidity 0.8.x since the compiler includes overflow and underflow checks natively.

Impact:

There can be gas savings if the `SafeMath` library is removed, however the `TRC20` contract code will need to be refactored.

Code Location:

```
contract TRC20 {
    using SafeMath for uint256;
    ...
```

Recommended Mitigation:

It is possible to remove the `SafeMath` library, however, refactoring (and testing) will be required.

[G-2] Dead code in `SafeMath` library

Description:

There are several functions defined in the `SafeMath` library that are not called in the protocol.

Impact:

The `mul`, `div` and `mod` functions are not used and can be removed.

Recommended Mitigation:

These functions are never called and can be removed. Alternatively as outlined in issue G-1 the entire library can be removed.

[G-3] TotalSupply and Decimals can be made immutable

Description:

`TRC20._totalSupply` & `TRC20._decimals` should be immutable to save gas.

Code Location:

```
uint8 private _decimals;
uint256 private _totalSupply;
```

Recommended Mitigation:

Make `TRC20._totalSupply` & `TRC20._decimals` as immutable storage variables to save gas.

Conclusion

Overall the protocol is a basic implementation of a `TRC20` token contract targeting the TRON blockchain. The code is well written, there are tests and the contracts are stright forward. The auditor has pointed out some gas saving changes that can be made to the contracts. Full details shown in the main body of this report.

Appendix

Slither Report Summary

Back to Slither Security Analysis

- **dead-code** (11 results) (Informational)
- **solc-version** (9 results) (Informational)
- **similar-names** (2 results) (Informational)
- **immutable-states** (2 results) (Optimization)

Aderyn Report Summary

Back to Aderyn Security Analysis

- Low Issues
  - L-1: Solidity pragma should be specific, not wide
  - L-2: PUSH0 is not supported by all chains
- NC Issues
  - NC-1: Functions not used internally could be marked external
  - NC-2: Constants should be defined and used instead of literals
  - NC-3: Event is missing `indexed` fields

Disclaimer

As of the date of publication, the information provided in this report reflects the presently held understanding of the auditor's knowledge of security patterns as they relate to the client's contract(s), assuming that blockchain technologies, in particular, will continue to undergo frequent and ongoing

development and therefore introduce unknown technical risks and flaws. The scope of the audit presented here is limited to the issues identified in the preliminary section and discussed in more detail in subsequent sections. The audit report does not address or provide opinions on any security aspects of the Solidity compiler, the tools used in the development of the contracts or the blockchain technologies themselves, or any issues not specifically addressed in this audit report.

The audit report makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, the legal framework for the business model, or any other statements about the suitability of the contracts for a particular purpose, or their bug-free status.

To the full extent permissible by applicable law, the auditors disclaim all warranties, express or implied. The information in this report is provided “as is” without warranty, representation, or guarantee of any kind, including the accuracy of the information provided. The auditors hereby disclaim, and each client or user of this audit report hereby waives, releases and holds all auditors harmless from, any and all liability, damage, expense, or harm (actual, threatened, or claimed) from such use.