QuillAudits

Audit Report
April, 2021

TURUS

Contents

Introduction

During the period of **April 12th, 2021 to April 14th**, 2021 – QuillHash Team performed security audit for **Turus Network** smart contracts. The code for audit was taken from the following link:

https://github.com/TurusNetwork/Solidity/tree/contract

Commit Hash - **98f7eea7b4e0554f2c72ea040573f502cefc21f6**

Overview of Turus Network

Turus (TRS) as a token is built on the Binance Smart chain which will be used on a decentralized platform where holders vote to shape the future of the platform.
TRS will also be used as a governance token for a Decentralized Autonomous Organization (DAO). This an entity in a digital system supported by intelligent contracts; comprising digital tools and protocols used to handle specific transactions or other elements a smart contract. Holders of the Turus Network governance token (TRS) through staking can make decisions, vote for ideas and receive rewards if those projects (ideas) yield profit.

Features:

- **Instant Fee Split**
  For every transaction carried out, holders of TRS earn a fraction from the total fees generated - thereby increasing their stake.
- **Burn Fee (reducing supply)**
  At certain blocks, 2% of each transfer is distributed between all holders of TRS.
- **Staking**
  Deposit your crypto assets, earn the most optimised interest rates and have access to voting privileges on the Turus DAO platform.
  Create multiple staking pools.
- **Non-Fungible Token (NFT)**
  TRS holders will get exclusive NFT's by auction sales or by farming TRS and getting rewards that can be used to redeem cards.

Tokenomics

TRS is a **BEP-20 token** developed on Binance Smart Chain.
Total Supply 50M
Initial Circulating Supply 35M
Price: 1 BNB = 10,000 TRS
Minimum Contribution: 0.2 BNB
Maximum Contribution: 10 BNB

Sales will be in 4 phases and details will be uploaded soon:

- Seed Sale
- Private Sale
- Public Sale A
- Public Sale B

** Unsold public sale tokens will be burnt.
** Public sale will be on a FCFS bases on Bounce Finance.
** Liquidity pair will be locked on Unicrypt.

Details:http://turus.network

Scope of Audit

The scope of this audit was to analyse **Turus Network (Turu.sol)** smart contract codebase for quality, security, and correctness.

OUT-OF-SCOPE: External contracts, External Oracles, other smart contracts in the repository or imported by Turus Network contract, economic attacks.

Check Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level
- Address hardcoded

- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Using delete for arrays
- Integer overflow/underflow
- Locked money
- Private modifier
- Revert/require functions
- Using var
- Visibility
- Using blockhash
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contracts care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.
SmartCheck.

Static Analysis

Static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step a series of automated tools are used to test security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerability or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of automated analysis were manually verified.

Gas Consumption

In this step we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Ganache, Solhint, Mythril, Slither, SmartCheck.

Issue Categories

Every issue in this report has been assigned with a severity level. There are four levels of severity and each of them has been explained below.

High severity issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality and we recommend these issues to be fixed before moving to a live environment.

Medium level severity issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems and they should still be fixed.

Low level severity issues

Low level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity four issues which indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

	High	Medium	Low	Informational
Reported	0	0	6	5
Open	0	0	6	5
Closed	0	0	0	0

Issues Found – Code Review / Manual Testing

High severity issues

None.

Medium severity issues

None.

Low severity issues

1. **Coding Style Issues**

   Coding style issues influence code readability and in some cases may lead to bugs in future. Smart Contracts have naming convention, indentation and code layout issues. It's recommended to use **Solidity Style Guide** to fix all the issues. Consider following the Solidity guidelines on formatting the code and commenting for all the files. It can improve the overall code quality and readability.

```
Turu.sol
   60:2  error  Line length must be no more than 120 but current length is 127  max-line-length
   63:2  error  Line length must be no more than 120 but current length is 122  max-line-length
  121:2  error  Line length must be no more than 120 but current length is 128  max-line-length
  236:2  error  Line length must be no more than 120 but current length is 126  max-line-length

✖ 4 problems (4 errors, 0 warnings)
```

2. **Order of layout**

   The order of functions as well as the rest of the code layout does not follow solidity style guide.
   Layout contract elements in the following order:
   a. Pragma statements
   b. Import statements
   c. Interfaces
   d. Libraries
   e. Contracts
   Inside each contract, library or interface, use the following order:
   a. Type declarations
   b. State variables
   c. Events

d. Functions

Please read following documentation links to understand the correct order:
https://solidity.readthedocs.io/en/v0.8.3/style-guide.html#order-of-layout

3. **Naming convention issues – Variables, Structs, Functions**

   It is recommended to follow all solidity naming conventions to maintain readability of code.
   Details:
   https://docs.soliditylang.org/en/v0.8.3/style-guide.html#naming-conventions

4. **Be explicit about which `uint` the code is using to avoid overflow and underflow issues**

   `uint` is an alias for `uint256`, but using the full form is preferable. Be consistent and use one of the forms. Explicitly declaring data types is recommended to avoid unexpected behaviours and/or errors.
   Code Lines: 22, 26, 29, 50, 72, 91, 116, 142, 154, 159, 193, 235, 264, 300, 312, 315, 316, 322, 331, 347, 356, 378, 382, 383, 389, 390, 396,

5. **Use local variable instead of state variable like length in a loop**

   For every iteration of for loop - state variables like length of non-memory array will consume extra gas. To reduce the gas consumption, it’s advisable to use local variable.
   Code Lines: 91.
   Links: Gas Limits and loops

6. **Function setFeesPercentage should emit an event**

   It’s a good practice to log important events like setting fees.

Informational severity issues

1. Overpowered owner

The contract is tightly coupled to the owner, making some functions callable only by the owner's address. This poses a serious risk: if the private key of the owner gets compromised, then an attacker can gain control over the contract.

2. Approve function of ERC-20 is vulnerable

A security issue called "Multiple Withdrawal Attack" - originates from two methods in the ERC20 standard for approving and transferring tokens. The use of these functions in an adverse environment (e.g., front-running) could result in more tokens being spent than what was intended. This issue is still open on the GitHub and several solutions have been made to mitigate it.
For more details on how to resolve the multiple withdrawal attack check the following document for details:
https://drive.google.com/file/d/1skR4BpZ0VBSQICIC_eqRBgGnACf2li5X/view?usp=sharing

3. Use of block.timestamp should be avoided

Do not use block.timestamp, now or blockhash as a source of randomness. Malicious miners can alter the timestamp of their blocks, especially if they can gain advantages by doing so. Contracts often need access to the current timestamp to trigger time-dependent events. As Ethereum is decentralized, nodes can synchronize time only to some degree. It is risky to use block.timestamp, as block.timestamp can be manipulated by
miners. Therefore block.timestamp is recommended to be supplemented with some other strategy in the case of high-value/risk applications.
Remediations:

- https://consensys.github.io/smart-contract-best-practices/recommendations/#timestamp-dependence
- https://consensys.github.io/smart-contract-best-practices/recommendations/#avoid-using-blocknumber-as-a-timestamp

4. Gas optimization tips

As the gas costs are increasing it is highly recommended to implement gas optimization techniques to reduce gas consumption.
https://blog.polymath.network/solidity-tips-and-tricks-to-save-gas-and-reduce-bytecode-size-c44580b218e6

5. Use of assembly is error-prone and should be avoided.

Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This bypasses several important safety features and checks of Solidity. You should only use it for tasks that need it, and only if you are confident with using it.0x vulnerability was caused by assembly code put there to optimize space. For security critical applications assembly is not a good idea. Assembly code is harder to read/audit, and it’s also just harder to get right.
Examples how inline assembly can be exploited:

- https://samczsun.com/the-0x-vulnerability-explained/
- https://medium.com/consensys-diligence/return-data-length-validation-a-bug-we-missed-4b7bbea8e9ab
- https://blog.0xproject.com/post-mortem-0x-v2-0-exchange-vulnerability-763015399578

Automated Testing

Slither

Slither is an open-source Solidity static analysis framework. This tool provides rich information about Ethereum smart contracts and has the critical properties. It 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.

```
# Check TRS

## Check functions
[🅱] totalSupply() is present
        [🅱] totalSupply() -> () (correct return value)
        [🅱] totalSupply() is view
[🅱] balanceOf(address) is present
        [🅱] balanceOf(address) -> () (correct return value)
        [🅱] balanceOf(address) is view
[🅱] transfer(address,uint256) is present
        [🅱] transfer(address,uint256) -> () (correct return value)
        [🅱] Transfer(address,address,uint256) is emitted
[🅱] transferFrom(address,address,uint256) is present
        [🅱] transferFrom(address,address,uint256) -> () (correct return value)
        [🅱] Transfer(address,address,uint256) is emitted
[🅱] approve(address,uint256) is present
        [🅱] approve(address,uint256) -> () (correct return value)
        [🅱] Approval(address,address,uint256) is emitted
[🅱] allowance(address,address) is present
        [🅱] allowance(address,address) -> () (correct return value)
        [🅱] allowance(address,address) is view
[🅱] name() is present
        [🅱] name() -> () (correct return value)
        [🅱] name() is view
[🅱] symbol() is present
        [🅱] symbol() -> () (correct return value)
        [🅱] symbol() is view
[🅱] decimals() is present
        [🅱] decimals() -> () (correct return value)
        [🅱] decimals() is view

## Check events
[🅱] Transfer(address,address,uint256) is present
        [🅱] parameter 0 is indexed
        [🅱] parameter 1 is indexed
[🅱] Approval(address,address,uint256) is present
        [🅱] parameter 0 is indexed
        [🅱] parameter 1 is indexed


        [🅱] TRS has increaseAllowance(address,uint256)
```

Slither didn't raise any critical issue with smart contracts. The smart contracts were well tested and all the minor issues that were raised have been documented in the report. Also, all other vulnerabilities of importance have already been covered in the manual audit section of the report.

Smartcheck

SmartCheck is a tool for automated static analysis of Solidity source code for security vulnerabilities and best practices. SmartCheck translates Solidity source code into an XML-based intermediate representation and checks it against XPath patterns. SmartCheck shows significant improvements over existing alternatives in terms of false discovery rate (FDR) and false negative rate (FNR).

```
ruleId: SOLIDITY_VISIBILITY
patternId: 910067
severity: 1
line: 89
column: 4
content: constructor(address[]memoryinitialFeeBlackList){balances[msg.sender]=totalSuppl
kList[i]);}emitTransfer(address(0),msg.sender,totalSupply);}

SOLIDITY_VISIBILITY :1
SOLIDITY_OVERPOWERED_ROLE :1
SOLIDITY_EXTRA_GAS_IN_LOOPS :1
SOLIDITY_GAS_LIMIT_IN_LOOPS :2
SOLIDITY_USING_INLINE_ASSEMBLY :1
SOLIDITY_SHOULD_RETURN_STRUCT :1
SOLIDITY_SHOULD_NOT_BE_VIEW :1
```

SmartCheck did not detect any high severity issue. All the considerable issues raised by SmartCheck are already covered in the code review section of this report.

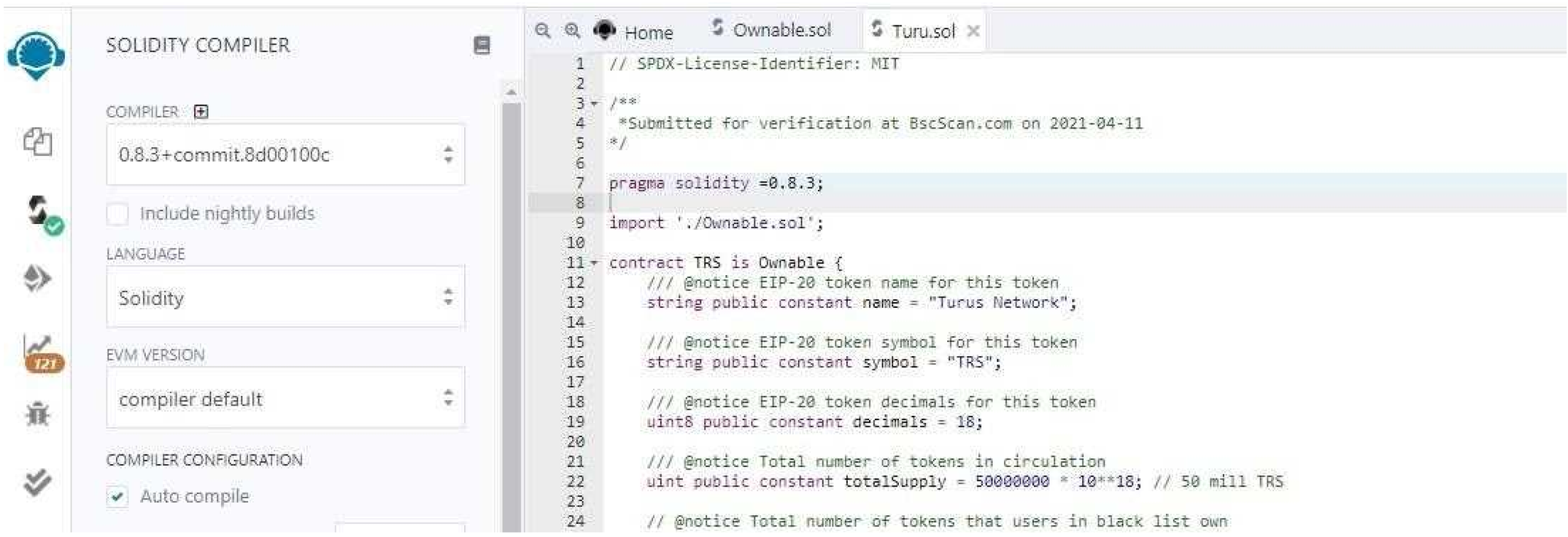
Mythril

Mythril is a security analysis tool for EVM bytecode. It detects security vulnerabilities in smart contracts built for Ethereum. It uses symbolic execution, SMT solving and taint analysis to detect a variety of security vulnerabilities.
Mythril did not detect any high severity issue. All the considerable issues raised by Mythril are already covered in the code review section of this report.

Remix IDE

Remix is a powerful, open source tool that helps you write Solidity contracts straight from the browser. Written in JavaScript, Remix supports both usage in the browser and locally.

The smart contract compiled without any error on Remix IDE.



The contract was successfully deployed Binance Smart Chain Testnet. And the gas consumption was found to be normal.

https://testnet.bscscan.com/tx/0xc43e1b7dc644c2de0e4444152o5d328c2c4275637b2bfcf6766871e2e

Closing Summary

Overall, the smart contract code is extremely well documented, follows a high-quality software development standard, contains many utilities and automation scripts to support continuous deployment / testing / integration, and does NOT contain any obvious exploitation vectors that QuillHash was able to leverage within the timeframe of testing allotted.
Overall, the smart contracts adhered to **BEP20** guidelines. No critical or major vulnerabilities were found in the audit. Several issues of **low severity were found** and reported during the audit.
The outcome of this security audit is satisfactory; due to time and resource constraints, only testing and verification of essential properties were performed to achieve objectives and deliverables set in the scope.
QuillHash recommends performing further testing to validate extended safety and correctness in context to the whole set of contracts.

Disclaimer

QuillHash audit is not a security warranty, investment advice, or an endorsement of the **Turus Network platform**. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the **Turus Network Team** put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

TURUS

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

hello@quillhash.com