QuillAudits

Audit Report
April, 2021

Contents

Introduction

During the period of **April 5th, 2021 to April 9th, 2021** – QuillHash Team performed security audit for **TRIFORCE TFC** Pool smart contract. The code for audit was taken from the following link:

https://github.com/triforce-proj/triforce/blob/main/TFC_POOL.sol

Commit Hash - **674fef947b85b9b22d0337e6771da1b2336506a9**

Updated Contracts:
Commit ID: ff1122debef3fac8504c917ead2af25750165773

Overview of TRIFORCE

TRIFORCE is a decentralized protocol that seeks to encompass multi-functional mechanisms using its deflationary token $TFC.
The token includes mechanisms such as automatic liquidity generation, frictionless rewards to token holders, automated rebalancer function, and token burn.
This is driven by TRIFORCE — TRIad of Fee levied On each tRansaCtion.

$TFC Transaction Fee

On each transaction (sell, buy, and transfer) of $TFC, the following fees are levied.
- 3% Liquidity Fee (to automatically generate liquidity).
- 3% Tax Fee (to reward holders).
- 3% Burn Fee (to decrease the supply and create higher market price).

Example: (For a transaction to buy 3333 tokens).
- 99.99 tokens as a fee to automatically generate liquidity.
- 99.99 tokens as a fee to reward holders.
- 99.99 tokens as a fee to burn.

So, the buyer receives 3333- (99.99 + 99.99 + 99.99) = 3033.33 tokens.

TRIFORCE Mechanism

-Automatically Generates Liquidity
-Frictionless rewards to holders
-Token burn on each transaction
-Automated BuyBacks

Tokenomics

#	DESCRIPTION	SUPPLY	TOKEN ALLOCATION
1	Presale	50%	5.00 M
2	Liquidity	23.8%	2.38 M
3	Staking Rewards	26.2%	2.62 M

Details:https://www.tfcprotocol.com/#/

Scope of Audit

The scope of this audit was to analyse TRIFORCE TFC Pool (TFC_POOL.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 TFC_POOL 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
- 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
- Unsafe type inference
- Implicit visibility level
- Address hardcoded
- 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
- Style guide violation
- Unchecked external call
- Unchecked math

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	2	23	5
Open	0	0	11	0
Closed	0	2	12	0

Issues Found – Code Review / Manual Testing

High severity issues

None.

Medium severity issues

1. 1.Reentrancy

   One of the major dangers of calling external contracts is that they can take over the control flow, and make changes to your data that the calling function wasn't expecting. It's recommended that the calls to external functions/events should happen after any changes to state variables in your contract so your contract is not vulnerable to a reentrancy exploit. When control is transferred to recipient, care must be taken to not create reentrancy vulnerabilities. OpenZeppelin has its own mutex implementation you can use called ReentrancyGuard. This library provides a modifier you can apply to any function called nonReentrant that guards the function with a mutex.
   Consider using **ReentrancyGuard** or the **checks-effects-interactions pattern.**

   Following link explains more about Reentrancy and ReentrancyGuard
   https://docs.openzeppelin.com/contracts/2.x/api/utils#ReentrancyGuard
   https://blog.openzeppelin.com/reentrancy-after-istanbul/

   Code Lines:

   - constructor() [#824-844]
   - transferFrom(address,address,uint256) [#896-905]
   - _transfer(address,address,uint256) [#1014-1073]
   - swapAndLiquify(uint256) [#1135-1165]
   - _rebalance(uint256) [#1253-1282]
   - deposit(uint256,uint256) [#1561-1579]
   - withdraw(uint256,uint256) [#1582-1609]
   - emergencyWithdraw(uint256) [#1612-1622]

Auditors Remarks: Fixed.

2. **Avoid using tx.origin for transfer**

Details: Abuse of Tx-origin Exploitation Code

Code Lines: 1164

Auditors Remarks: Fixed.

Low level severity issues

1. **Compiler version should be fixed**

Solidity source files indicate the versions of the compiler they can be compiled with. It's recommended to lock the compiler version in code, as future compiler versions may handle certain language constructions in a way the developer did not foresee. **It is recommended to use any fixed compiler version from 0.6.6 to 0.6.12**.
Code Lines: #3

Auditors Remarks: Fixed.

2. **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.

```
TFC_POOL.sol
    22:2  error  Line length must be no more than 120 but current length is 132  max-line-length
   334:2  error  Line length must be no more than 120 but current length is 160  max-line-length
   359:2  error  Line length must be no more than 120 but current length is 156  max-line-length
   732:2  error  Line length must be no more than 120 but current length is 127  max-line-length
   777:2  error  Line length must be no more than 120 but current length is 138  max-line-length
   779:2  error  Line length must be no more than 120 but current length is 151  max-line-length
   783:2  error  Line length must be no more than 120 but current length is 174  max-line-length
   784:2  error  Line length must be no more than 120 but current length is 152  max-line-length
   799:2  error  Line length must be no more than 120 but current length is 157  max-line-length
   800:2  error  Line length must be no more than 120 but current length is 127  max-line-length
   903:2  error  Line length must be no more than 120 but current length is 121  max-line-length
  1245:2  error  Line length must be no more than 120 but current length is 125  max-line-length
  1262:2  error  Line length must be no more than 120 but current length is 137  max-line-length
  1393:2  error  Line length must be no more than 120 but current length is 121  max-line-length
  1399:2  error  Line length must be no more than 120 but current length is 132  max-line-length
  1407:2  error  Line length must be no more than 120 but current length is 130  max-line-length
  1414:2  error  Line length must be no more than 120 but current length is 128  max-line-length

✖ 17 problems (17 errors, 0 warnings)
```

Auditors Remarks: Fixed.

3. 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:

- Pragma statements
- Import statements
- Interfaces
- Libraries
- Contracts

Inside each contract, library or interface, use the following order:

- Type declarations
- State variables
- Events
- Functions

Please read following documentation links to understand the correct order:
- https://solidity.readthedocs.io/en/v0.6.6/style-guide.html#order-of-layout
- https://solidity.readthedocs.io/en/v0.6.6/style-guide.html#order-of-functions

Auditors Remarks: Fixed.

4. 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/v0.6.6/style-guide.html#naming-conventions

Auditors Remarks: Fixed.

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: 552-558, 561-566, 572-577, 581-585, 594-598, 1217, 1354, 1591, 1599, 1660.

Auditors Remarks: Partially Fixed.
Lines: 140, 141, 144, 1618 (Need to be fixed)

6. Implicit Visibility

It's a good practice to explicitly define visibility of state variables, functions, interface functions and fallback functions. The default visibility of state variables – internal; function – public; interface function - external.

Code Lines: 771.

Auditors Remarks: Fixed.

7. Use external function modifier instead of public

The public functions that are never called by contract should be declared external to save gas. List of functions that can be declared external:

- isExcluded(address) [#936-938]
- rebalance() [#1244-1251]
- setExcludedFromFee(address,bool) [#1294-1296]
- setSwapAndLiquifyEnabled(bool) [#1298-1301]
- setTaxFee(uint256) [#1309-1311]
- setBurnFee(uint256) [#1313-1315]
- setLiquidityFee(uint256) [#1317-1319]
- setLpRewardFee(uint256) [#1321-1323]
- setLiquidityRemoveFee(uint256) [#1325-1327]
- setRebalanceCallerFee(uint256) [#1329-1331]
- setSwapCallerFee(uint256) [#1333-1335]
- setMinTokensBeforeSwap(uint256) [#1337-1339]
- setMinTokenBeforeReward(uint256) [#1341-1343]
- setRebalanceInterval(uint256) [#1345-1347]
- setRebalanceEnabled(bool) [#1349-1351]
- transferAnyBEP20Tokens(address,address,uint256) [#1354-1356]
- transferBNB(address,uint256) [#1358-1364]
- add(uint256,IERC20,bool) [#1474-1497]

- set(uint256,uint256,bool) [#1500-1512]
- deposit(uint256,uint256) [#1561-1579]
- withdraw(uint256,uint256) [#1582-1609]
- emergencyWithdraw(uint256) [#1612-1622]
- transferAnyBEP20Tokens(address,address,uint256) [#1660-1662]

Auditors Remarks: Fixed.

8._decimals should be constant [#761]

Auditors Remarks: Not Fixed.

9._feeDecimal should be constant [#776]

Auditors Remarks: Fixed.

10._name should be constant [#759]

Auditors Remarks: Not Fixed.

11._symbol should be constant [#760]

Auditors Remarks: Not Fixed.

12._tokenTotal should be constant [#768]

Auditors Remarks: Not Fixed.

13._taxFee should be constant [#777]

Auditors Remarks: Not Fixed.

14._burnFee should be constant [#778]

Auditors Remarks: Not Fixed.

15._liquidityFee should be constant [#779]

Auditors Remarks: Not Fixed.

16._liquidityRemoveFee should be constant [#782]

Auditors Remarks: Not Fixed.

17._rebalanceCallerFee should be constant [#783]

Auditors Remarks: Not Fixed.

18._swapCallerFee should be constant [#784]

Auditors Remarks: Not Fixed.

19._maxTxAmount should be constant [#786]

Auditors Remarks: Not Fixed.

20.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: 991, 1120, 1532.
Links: Gas Limits and loops

Auditors Remarks: Fixed.

21.View functions should not modify the state

Using inline assembly that contains certain opcodes is considered as modifying the state. Functions that modify the state should not be declared as pure functions. Do not declare functions that change the state as view.
Code Lines: 304.

Auditors Remarks: Fixed.

22.Consider using struct instead of multiple return values.

Replace multiple return values with a struct. It helps improve readability.
Code Lines: 558, 566, 1444.

Auditors Remarks: Fixed.

23.Use require for error handling while using transfer

The BEP-20 transfer function returns a Boolean value if the value was transferred successfully or not. It’s recommended to use require for error handling when transfer function is called.
Code Lines: 1200-1215, 1217-1229, 1354-1356, 1626-1630, 1660-1662

Auditors Remarks: Fixed.

Informational severity issues

1. Favor Pull over push external calls

External calls can fail accidentally or deliberately. To minimize the damage caused by such failures, it is often better to isolate each external call into its own transaction that can be initiated by the recipient of the call. This is especially relevant for payments, where it is better to let users withdraw funds rather than push funds to them automatically.
https://consensys.github.io/smart-contract-best-practices/recommendations/#favor-pull-over-push-for-external-calls

2. 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.
Code Lines: 1014-1073, 1244-1251, 1444-1459.

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

3. 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

4. **Low level calls**

   Low-level calls do not check for code existence or call success. Use of low-level calls should be avoided as they are error-prone.
   Code Lines: 336-348, 433-461, 1358-1364

5. **Do not use EVM assembly. Use of assembly is error-prone and should be avoided. [#304-318, 433-461]**

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 TRIFORCE

## 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


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

```
INFO:Printers:
Compiled with solc
Number of lines: 1663 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 15 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 39
Number of informational issues: 60
Number of low issues: 22
Number of medium issues: 17
Number of high issues: 5
ERCs: ERC20
```

Name	# functions	ERCS	ERC20 info	Complex code	Features
SafeMath	8			No	
Address	7			No	Send ETH Assembly
IPancakeFactory	1			No	
IPancakePair	1			No	
IPancakeRouter02	8			No	Receive ETH
RewardWallet	1			No	
Balancer	1			No	
IERC20	6	ERC20	No Minting Approve Race Cond.	No	
SafeERC20	6			No	Send ETH Tokens interaction
TRIFORCE	62	ERC20	No Minting Approve Race Cond.	Yes	Receive ETH Send ETH Tokens interaction
TRIFORCE_POOLS	26			No	Tokens interaction

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).

```
SOLIDITY_VISIBILITY :33
SOLIDITY_OVERPOWERED_ROLE :19
SOLIDITY_PRAGMAS_VERSION :1
SOLIDITY_PRIVATE_MODIFIER_DONT_HIDE_DATA :7
SOLIDITY_EXTRA_GAS_IN_LOOPS :2
SOLIDITY_ADDRESS_HARDCODED :4
SOLIDITY_GAS_LIMIT_IN_LOOPS :3
SOLIDITY_UNCHECKED_CALL :5
SOLIDITY_DIV_MUL :1
SOLIDITY_SHOULD_RETURN_STRUCT :3
SOLIDITY_TX_ORIGIN :1
SOLIDITY_SAFEMATH :3
SOLIDITY_USING_INLINE_ASSEMBLY :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.

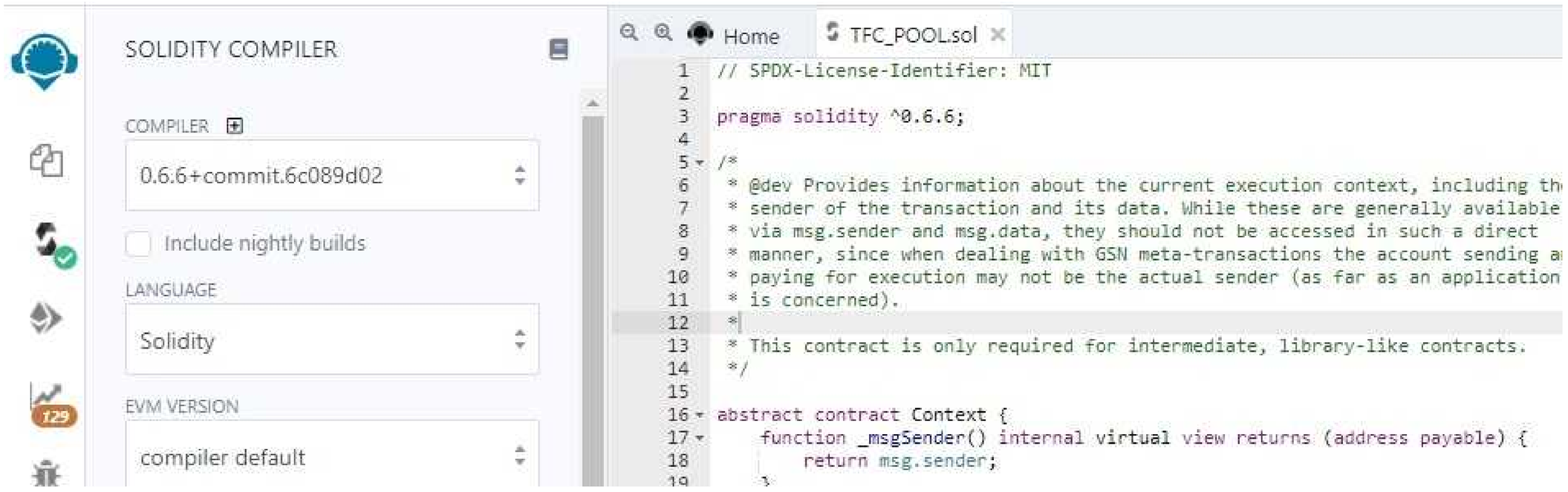
Mythril

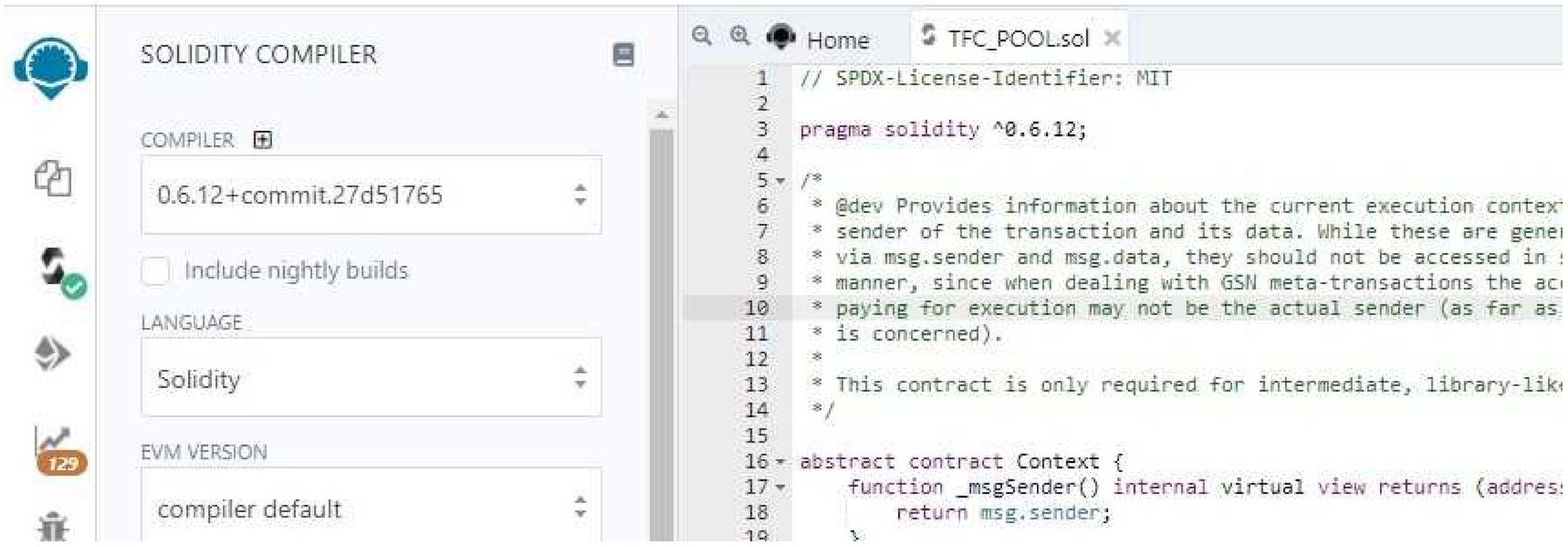
Mythril is a security analysis tool for EVM bytecode. It detects security vulnerabilities in smart contracts built for Ethereum. It uses symbolic execution, SMT solving and taint analysis to detect a variety of security vulnerabilities.
Mythril did not detect any high severity issue. All the considerable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

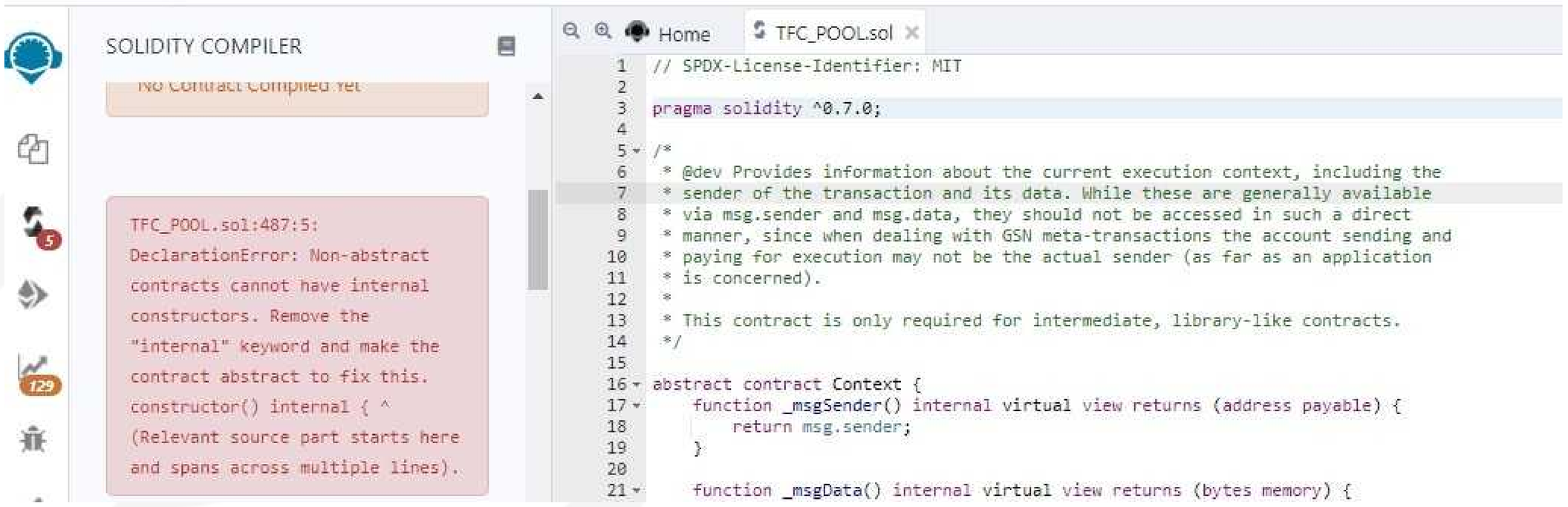
Remix IDE

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 TFC POOL smart contract was tested for various compiler versions. The smart contract compiled without any error on explicitly setting compiler version **0.6.6 to 0.6.12**.







It is recommended to use any fixed compiler version from 0.6.6 to 0.6.12.

The contract failed to compile from version less than 0.7.0 onwards as few functionalities in solidity have been deprecated.

The contract was successfully deployed. And the gas consumption was found to be normal.

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 **ERC20/BEP20** guidelines. No critical or major vulnerabilities were found in the audit. Several issues of **medium and 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.

All the medium severity and 12 low severity issues were fixed by the TRIFORCE TEAM.

Disclaimer

QuillHash audit is not a security warranty, investment advice, or an endorsement of the **TFC Protocol 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 **TFC Team** put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

hello@quillhash.com