Tron BTC Smart Contract 1st Audit Report

Introduction 3
Scope of Audit 3
Check Vulnerabilities 3
Techniques and Methods 5
Structural Analysis 5
Static Analysis 5
Code Review / Manual Analysis 5
Gas Consumption 5
Tools and Platforms used for Audit 5
Issue Categories 6
High Severity Issues 6
Medium Severity Issues 6
Low Severity Issues 6
Informational 6
Number of issues per severity 6
Contract Details 7
Issues Found – Code Review / Manual Testing 8
High Severity Issues 8
Medium Severity Issues 8
Low Severity Issues 8
Informational 8
Closing Summary 9
Disclaimer 10

Scope of Audit

The scope of this audit was to analyze and document TronBTC smart contract codebase for quality, security, and correctness.

Check Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- 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
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

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.

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 the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities 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 the 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 to 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 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 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

TYPE	HIGH	MEDIUM	LOW	INFORMATIONAL
OPEN	5	4	2	4
CLOSED	0	0	0	0

Introduction

During the period of **February 1st, 2021 to February 4th, 2021** - Quillhash Team performed a security audit for Tron BTC smart contracts.

The code for the audit was taken from following the following link**:**

https://ropsten.etherscan.io/address/0x6c0a0323175cc07e340fa6673326fc24566b2ed0#code

A. Contract Details

Name - TronBTCApp

About the Contract:
TronBTC primarily aims to create a value for TBXC utility coin by utilizing the **staking** as well as **referral** model.
However, it should be noted that the BTC part is not yet included and the BTC interaction shall be introduced at later stages.

Moreover, **TBXC** is a zero pre-mine coin and is owned by this contract. Therefore, no single authority has any special privileges of minting or burning the coin.

Issues Found – Code Review / Manual Testing

High Severity Issues

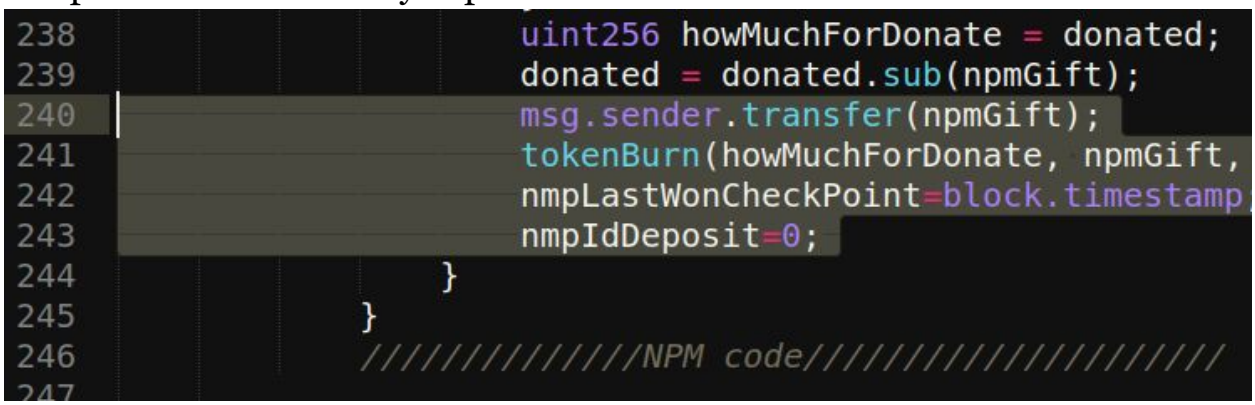
A.1 State variables updated after External Call in "deposit" Function

Line no - 242,243

Description:

The ***deposit function*** in the contract modifies its state variables after the external calls(*at Line 240 & 241*). Although the function checks whether or not the caller is a **contract(***at Line 205***),** it must not violate the ***Check Effects Interaction Pattern*****.**

Moreover, modifying the state of any variable after external calls leads to a potential re-entrancy exploit scenario.



Recommendation:

Follow the **Check Effect Interaction Pattern.**

A.2 State variables updated after External Call in "tokenBurn" Function

Line no - 269

Description:

The ***tokenBurn function*** includes externals calls *at Line 265 & 267*

However, a state variable, i.e., "***donated",*** is modified after the external calls , thus leading to a potential re-entrancy exploit scenario.

```
function tokenBurn(uint256 howMuchForDonate, uint256 _npmGift, uint256
    // token burn caused by new deposits
    uint256 tokenToBurn;
    uint256 howMuchToBuyAtDex;
    howMuchToBuyAtDex = donated.mul(NMP_DONATED_PURCHASE_PERCENT).div(1
    tokenToBurn = justswap.trxToTokenTransferInput.value(howMuchToBuyAt
    globalTokensBurned = globalTokensBurned + tokenToBurn;
    PlatformTokenApp.burn(address(this), tokenToBurn);
    emit EvtTokensSwapBurn(msg.sender,howMuchForDonate, howMuchToBuyAtD
    donated = 0;
}
```

Recommendation:

[Check Effect Interaction Pattern](#) must be followed and implemented.

A.3 Contract State updated after External Call in "reinvest" Function

Line no - 297

Description:

The ***reinvest function*** in the contract includes externals calls *at Line 293 & 295*

However, the contract's state is modified after the external calls which is not considered as a best practice and might lead to an exploit scenario.

```
282 ▾   function reinvest() external {
283       require(!isItContract(msg.sender) && msg.sender == tx.origin);
284       updateProfits(msg.sender);
285       Investor storage investor = investors[msg.sender];
286       uint256 depositAmount = investor.profitGained;
287       require(address(this).balance >= depositAmount);
288       investor.profitGained = 0;
289       investor.trxDeposit = investor.trxDeposit.add(depositAmount/2)
290       investor.reinvested += depositAmount;
291       emit EvtReinvest(msg.sender, depositAmount);
292       payUplines(depositAmount, investor.upline);
293       platformAddress.transfer(depositAmount.div(10));
294       if(globalInvested > INITIAL_GROWTH_NUMBER*10**TOKENS_DECIMAL)
295             issueTokens(msg.sender, depositAmount.div(2).div(10));
296       }
297       investor.lastWithdrawTime = now;
298   }
```

Recommendation:

Reinvest should follow the [Check Effect Interaction Pattern](#).

A. 4 Invalid Identifier in Deposit Function. Leads to PARSE ERROR

Line no - 206

Description:

The require statement(*at line 206*), in the **deposit function** of the contract

includes an unknown identifier called "**trx".**

This will lead to a parse error while contract compilation

```
204     function deposit(address _upline, uint256 _userType) externa
205         require(!isItContract(msg.sender) && msg.sender == tx.or
206         require(msg.value >= 100 trx, "ma");
207         require(startTheProject==1,"ns");
```

Recommendation:-

Any invalid and unknown identifiers/keywords should be removed from the

contract.

A.5 "transferProfitGainedContract" function modifies Contract State after External Call

Line no - 330,331

Description:

The ***transferProfitGainedContract function*** in the contract includes externals calls *at Line 329*

However, the function modifies the state of some State Variables*(at Line **330,331**)* after the external calls. This is not considered as a best practice and might lead to an exploit scenario.

```
325                 investor.howmuchPaid = investor.howmuchPaid.add(payout);
326                 investor.profitGained = investor.profitGained.sub(payout);
327                 investor.trxDeposit = investor.trxDeposit.sub(payout/2);
328                 investor.trxDeposit = investor.trxDeposit.add(payout.div(4));
329                 msg.sender.transfer(payout.mul(3).div(4)); // 75% to user
330                 investor.lastWithdrawTime = now;
331                 donated += payout.div(4); // 25% percent
332                 emit EvtWithdrawn(msg.sender, payout);
333             }
334         }
```

Recommendation:

Reinvest should follow the [Check Effect Interaction Pattern](#).

Medium Severity Issues

A.6 burnTokensAmount function doesn't follow the Check Effects interaction pattern.

Line no - 196

Description:

The **burnTokensAmount** function updates the contract state, at ***Line 200,*** after making an external call.

Although the external call is made to the ***PlatformTokenApp*** itself, it is always considered a better practice to update states before any external call.

```
196     function burnTokensAmount(uint256 _amount) public {
197         Investor storage investor = investors[msg.sender];
198         require(investor.userType==2 || investor.userType==3, "!muau
199         PlatformTokenApp.burn(msg.sender, _amount);
200         investor.tokensBurned = investor.tokensBurned.add(_amount);
201         emit EvtTokensBurn(msg.sender,_amount,  investor.userType);
202     }
```

Recommendation:

[Check Effect Interaction Pattern](#) must be followed and implemented.

A.7 issueTokens function doesn't follow the Check Effects interaction pattern.

Line no - 169

Description:

The ***issueTokens*** function updates the contract state (at Line ***177,184,191,192)*** after making an external call.

Although the external call is made to the ***PlatformTokenApp*** itself, it is always considered a better practice to update states before any external call.

Recommendation:

Check Effect Interaction Pattern must be followed and implemented.

A.8 setContractOwner function lacks a Zero Address Check.

Line no - 93

Description:

The **setContractOwner** function doesn't validate the **_contractOwner** address passed as a parameter.

```
93      function setContractOwner(address _contractOwner) external {
94          require(msg.sender == contractOwner, "!co");
95          contractOwner = _contractOwner;
96      }
```

Recommendation:

The new owner's address must be checked with a **require** statement.

require(_contractOwner != address(0),"Invalid address passed");

A.9 Unused Local Variable

Line no - 166

Description:
The **register function** includes a local variable called **_upline** which Is never used within the function.

```
163     function register(address _addr, address _upline) private{
164       Investor storage investor = investors[_addr];
165       investor.upline = _upline;
166       address _upline1 = _upline;
```

Recommendation:
Since there is no specific use of this variable, it must be removed.

Low Severity Issues

A.10 setNmpRandomDevider function doesn't emit any event

Line no - 103

Description:
The ***setNmpRandomDevider function*** modifies the state of some very crucial arithmetic state variables but doesn't emit any event after the updation of those variables.
Since there is no event emitted on updating this variable, it might be difficult to track it off-chain.

***Recommendation*:**
An event should be fired after changing the imperative arithmetic variables.

A.11 Return Value of External call is never used
Line no - 81

Description:
The return value of the externals made in the following line is never used:

```
79          PAIR_ADDRESS = exchangeAddress;
80          justswap = IJustswapExchange(PAIR_ADDRESS);
81          IERC20(myTokenAddress).approve(exchangeAddress, 10000000000000000000);
82          PlatformTokenApp = myTokenAddress;
```

Recommendation:
Effective use of all return values must be ensured within the contract.

Informational

A.12 External visibility should be preferred

Line no - 196

Description:
Those functions that are never called throughout the contract should be marked as ***external*** visibility instead of ***public*** visibility.

This will effectively result in Gas Optimization as well.

Therefore, the following function must be marked as **external** within the contract:

```
function burnTokensAmount(uint256 _amount) public {
    Investor storage investor = investors[msg.sender];
    require(investor.userType==2 || investor.userType==3, "!muau");
    PlatformTokenApp.burn(msg.sender, _amount);
    investor.tokensBurned = investor.tokensBurned.add(_amount);
    emit EvtTokensBurn(msg.sender,_amount,  investor.userType);
}
```

Recommendation:
Change the visibility specifier to external.

A.13 Constant declaration should be preferred

Line no- 11

Description:

State variables that are not supposed to change throughout the contract should be declared as **constant.**

Recommendation*:*

The following state variables need to be declared as constant:

- ***timeDevider***

A.14 Absence of Error messages in Require Statements

Description:

Some of the ***require statements*** in the contract don't include an error message. While this makes it troublesome to detect the reason behind a particular function revert, it also reduces the readability of the code.

The following lines have require statements without error messages:

- *Line 67*
- *Line 205*
- *Line 273*
- *Line 287*

Recommendation:

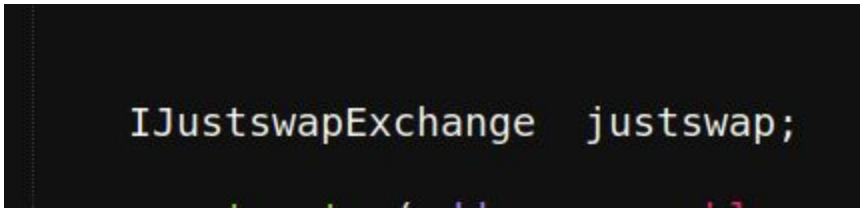
Error messages should be included in every require statement

A.15 Explicit visibility declaration is missing

Line no - 63

Description:

The following element in the contract have not been assigned any visibility explicitly:



Recommendation:
Visibility specifiers should be assigned explicitly in order to avoid ambiguity.

Closing Summary

Overall, smart contracts are very well written and adhere to guidelines.

However, during the process of audit several issues of high, medium as well as low severity which might affect the intended behaviour of the contracts.

Disclaimer

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