SMART CONTRACT AUDIT REPORT

For

Troncase

Prepared By: Kishan Patel

Prepared For: Troncase.io

Prepared on: 03/12/2020

Table of Content

• Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

• Overview of the audit

The project has 1 file. It contains approx 485 lines of Solidity code. All the functions and state variables are well commented using the natspec documentation, but that does not create any vulnerability.

• Attacks made to the contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

- ## Over and under flows

An overflow happens when the limit of the type variable uint256, 2 ** 256, is exceeded. What happens is that the value resets to zero instead of incrementing more. On the other hand, an underflow happens when you try to subtract 0 minus a number bigger than 0. For example, if you subtract 0 - 1 the result will be = 2 ** 256 instead of -1. This is quite dangerous.

This contract **does** check for overflows and underflows by using OpenZeppelin's SafeMath to mitigate this attack, but all the functions have strong validations, which prevented this attack.

- ## Short address attack

If the token contract has enough amount of tokens and the buy function doesn't check the length of the address of the sender, the Tron's virtual machine will just add zeros to the transaction until the address is complete.

Although this contract **is not vulnerable** to this attack, but there are some point where users can mess themselves due to this (Please see below). It is highly recommended to call functions after checking validity of the address.

- ## Visibility & Delegate call

It is also known as, The Parity Hack, which occurs while misuse of Delegate call.

No such issues found in this smart contract and visibility also properly addressed. There are some places where there is no visibility defined. Smart Contract will assume "Public" visibility if there is no visibility defined. It is good practice to explicitly define the visibility, but again, the contract is not prone to any vulnerability due to this in this case.

- ## Reentrancy / TheDAO hack

Reentrancy occurs in this case: any interaction from a contract (A) with another contract (B) and any transfer of Tron hands over control to that contract (B).

This makes it possible for B to call back into A before this interaction is completed.

Use of “require” function in this smart contract mitigated this vulnerability.

- ## Forcing tron to a contract

While implementing “selfdestruct” in smart contract, it sends all the tron to the target address. Now, if the target address is a contract address, then the fallback function of target contract does not get called. And thus Hacker can bypass the “Required” conditions. Here, the Smart Contract’s balance has never been used as guard, which mitigated this vulnerability.

- # Good things in smart contract
  - ### SafeMath library:-
    - You are using SafeMath library it is a good thing. This protects you from underflow and overflow attacks.

```
451
452 library SafeMath {
453
454     function add(uint256 a, uint256 b) internal
455         uint256 c = a + b;
456         require(c >= a, "SafeMath: addition over
457
```

  - ### Compiler version is fixed:-
    - This is good practice to depend on one solidity version.
    - If you put(^) symbol then you are able to get compiler version 0.4.25 and above. But if you don’t use (^) symbol then you are able to use only 0.4.25 version. And if there are some changes come in the compiler and you use the old version then some issues may come at deploy time.
    - Please check you are using latest version of solidity.

```
1  pragma solidity 0.5.10;
2
3  contract Troncase {
4      using SafeMath for uint;
5
```

- **Good required condition in functions:-**
    - Here you are checking that to projectAddr and marketingAddr is not contract address.

```
63
64      constructor(address payable marketingAddr, address payable projectAddr, addres
65          require(!isContract(marketingAddr) && !isContract(projectAddr));
66          marketingAddress = marketingAddr;
67          projectAddress = projectAddr;
```

    - Here you are checking that msg.sender is not contract addres and msg.value is bigger than INVEST_MIN_AMOUNT(100 trx) and less than DEPOSITS_MAX(4000000 trx) and user can deposits 100 times.

```
72
73      function invest(address referrer) public payable {
74          require(!isContract(msg.sender) && msg.sender == tx.origin);
75
76          require(msg.value >= INVEST_MIN_AMOUNT && msg.value <= INVEST_MAX_AMO
77
78          User storage user = users[msg.sender];
79
80          require(user.deposits.length < DEPOSITS_MAX, "Maximum 100 deposits fr
```

    - Here you are checking that totalAmount is bigger than 0.

```
207
208             require(totalAmount > 0, "User has no dividends");
209
210             uint contractBalance = address(this).balance;
211             if (contractBalance < totalAmount) {
```

- **Critical vulnerabilities found in the contract**

=> No critical vulnerabilities found

- **Medium vulnerabilities found in the contract**

=> No Medium vulnerabilities found

- # Low severity vulnerabilities found

  - ## 7.1: Short address attack:-

=> This is not big issue in solidity, because now a days is increased In the new solidity version. But it is good practice to Check for the short address.

=> After updating the version of solidity it's not mandatory.

=> In all functions you are not checking the value of address parameter here I am showing only necessary.

- **Function: - constructor ('marketingAddr', 'projectAddr', 'adminAddr')**

```
62      event FeePayed(address indexed user, uint totalAmount);
63
64      constructor(address payable marketingAddr, address payable projectAddr, address payable adminAddr)
65          require(!isContract(marketingAddr) && !isContract(projectAddr));
66          marketingAddress = marketingAddr;
```

  - It's necessary to check the addresses value of "marketingAddr", "projectAddr", and "adminAddr". Because Here you are passing whatever variable comes in "marketingAddr", "projectAddr", and "adminAddr" addresses from outside.

- **Function: - invest ('referrer')**

```
72
73      function invest(address referrer) public payable {
74          require(!isContract(msg.sender) && msg.sender ==
75
76          require(msg.value >= INVEST_MIN_AMOUNT && msg.val
77
```

  - It's necessary to check the address value of "referrer". Because here you are passing whatever variable comes in "referrer" Address from outside.

○ 7.2: Use safeMath library in these functions:-

=> You have implemented safeMath library in a smart contract.

=> Here are some functions where you forgot to use safeMath library.

=> safeMath library protects your smart contract from underflow and overflow an attack.

- **Function: - invest**

```
155         user.deposits.push(Deposit(uint64(msgValue), 0, 
156 
157         totalInvested = totalInvested.add(msgValue);
158         totalDeposits++;
159 
```

- **Function: - withdraw**

```
186                 dividends = (uint(user.deposits[i].amount).mul(userPercentRate+communityBonus+leaderbonus).
187                     .mul(block.timestamp.sub(uint(user.deposits[i].start)))
188                     .div(TIME_STEP);
189 
190             } else {
191 
192                 dividends = (uint(user.deposits[i].amount).mul(userPercentRate+communityBonus+leaderbonus).
193                     .mul(block.timestamp.sub(uint(user.checkpoint)))
```

- **Function: - getUserAvailable**

```
298 
299                 dividends = (uint(user.deposits[i].amount).mul(userPercentRate+communityBonus+leaderbonus).
300                     .mul(block.timestamp.sub(uint(user.deposits[i].start)))
301                     .div(TIME_STEP);
302 
303             } else {
304 
305                 dividends = (uint(user.deposits[i].amount).mul(userPercentRate+communityBonus+leaderbonus).
306                     .mul(block.timestamp.sub(uint(user.checkpoint)))
307                     .div(TIME_STEP);
```

- **Function: - getUserDeposits**

```
414                 amount[index] = uint(user.deposits[i-1].amount);
415                 withdrawn[index] = uint(user.deposits[i-1].withdrawn);
416                 // refback[index] = uint(user.deposits[i-1].refback);
417                 start[index] = uint(user.deposits[i-1].start);
418                 index++;
419             }
```

• Summary of the Audit

Overall the code is well and performs well. **There is no**

backdoor to run funds from smart contract.

Please try to check the address and value of token externally before sending to the solidity code.

Our final recommendation would be to pay more attention to the visibility of the functions , hardcoded address and mapping since it’s quite important to define who’s supposed to executed the functions and to follow best practices regarding the use of assert, require etc. (which you are doing ;)).

• **Note:** Please focus on version of solidity (Use latest),
check addresses.