

SMART CONTRACT AUDIT REPORT

For

Troncase

Prepared By: Kishan Patel

Prepared on: 11/07/2020

Prepared For: Troncase.pro

Table of Content

- Disclaimer
- Overview of the audit
- Attacks made to the contract
- Good things in smart contract
- Critical vulnerabilities found in the contract
- Medium vulnerabilities found in the contract
- Low severity vulnerabilities found in the contract
- Summary of the audit

• **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**

This is re-check of this contract. The project has 1 file. It contains approx 488 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 -1 instead of 2^{256} . 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.

```
2 library SafeMath {
3
4     function add(uint256 a, uint256 b) internal pure returns (uint256) {
5         uint256 c = a + b;
6         require(c >= a, "SafeMath: addition overflow");
7     }
8 }
```

- **Compiler version is fixed:-**

=> In this file you have put “pragma solidity 0.5.10;” which is a good way to define compiler version.

=> Solidity source files indicate the versions of the compiler they can be compiled with. Pragma solidity >=0.5.10; // bad: compiles 0.5.10 and above
pragma solidity 0.5.10; //good: compiles 0.5.10 only

=> If you put(>=) symbol then you are able to get compiler version 0.5.10 and above. But if you don't use(^/>=) symbol then you are able to use only 0.5.10 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.

=> Use latest version of solidity.

- **Good required condition in functions:-**

- Here you are checking that to projectAddr and marketingAddr is not contract address.

```
89
90 ▾   constructor(address payable marketingAddr, address payable projectAddr, address
91       require(!isContract(marketingAddr) && !isContract(projectAddr));
92       marketingAddress = marketingAddr;
93       projectAddress = projectAddr;
```

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

```
182   }
183
184       require(totalAmount > 0, "User has no dividends");
185
186       uint contractBalance = address(this).balance;
187       if (contractBalance < totalAmount) {
188           uint contractBalance = address(this).balance;
```

- Here you are checking that msg.sender is not contract address and msg.value is bigger than INVEST_MIN_AMOUNT(100 trx) and less than DEPOSITS_MAX(4000000 trx) and user can deposits 300 times. **Please correct error message for user deposits.**

```
272 ▾   function invest(address referrer) public payable {
273       require(!isContract(msg.sender) && msg.sender == tx.origin);
274
275       require(msg.value >= INVEST_MIN_AMOUNT && msg.value <= INVEST_MAX_AMOUNT,
276
277       User storage user = users[msg.sender];
278
279       require(user.deposits.length < DEPOSITS_MAX, "Maximum 100 deposits from ad
```

- **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')**

```
90 constructor(address payable marketingAddr, address payable projectAddr, address payable adminAddr) public payable {
91     require(!isContract(marketingAddr) && !isContract(projectAddr));
92     marketingAddress = marketingAddr;
93     projectAddress = projectAddr;
```

- 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')**

```
271
272 function invest(address referrer) public payable {
273     require(!isContract(msg.sender) && msg.sender == tx.origin);
274
275     require(msg.value >= INVEST_MIN_AMOUNT && msg.value <= INVEST_MAX_AMOUNT,
```

- 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: - withdraw

```
160     if (user.deposits[i].start > user.checkpoint) {
161
162         dividends = (uint(user.deposits[i].amount).mul(userPercentRate+
163             .mul(block.timestamp.sub(uint(user.deposits[i].start)))
164             .div(TIME_STEP));
165
166     } else {
167
168         dividends = (uint(user.deposits[i].amount).mul(userPercentRate+
169             .mul(block.timestamp.sub(uint(user.checkpoint)))
170             .div(TIME_STEP));
171     }
```

• Function: - getUserAvailable

```
244
245         dividends = (uint(user.deposits[i].amount).mul(userPercentRate+
246             .mul(block.timestamp.sub(uint(user.deposits[i].start)))
247             .div(TIME_STEP));
248
249     } else {
250
251         dividends = (uint(user.deposits[i].amount).mul(userPercentRate+
252             .mul(block.timestamp.sub(uint(user.checkpoint)))
253             .div(TIME_STEP));
254     }
```

○ 7.3: Uncheck return response of transfer method:-

=> I have found that you are transferring fund to address using a transfer method.

=> It is always good to check the return value or response from a function call.

=> Here are some functions where you forgot to check a response.

=> I suggest, if there is a possibility then please check the response.

• Function: - withdraw

```
205     user.checkpoint = uint32(block.timestamp);
206
207     msg.sender.transfer(totalAmount);
208
209     totalWithdrawn = totalWithdrawn.add(totalAmount);
```

- Here you are calling transfer method 1 time. It is good to check that the transferFrom is successfully done or not.

- **Function: - invest**

```
305 marketingAddress.transfer(marketingFee);  
306 projectAddress.transfer(projectFee);  
307 adminAddress.transfer(adminFee);  
308 networkAddress.transfer(network);
```

- Here you are calling transfer method 4 time. It is good to check that the transferFrom is successfully done or not.

• Summary of the Audit

Overall the code is well and performs well. There is no back door to steal fund.

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 ;)).

- **Good Point:** Code performance and quality is good.
Address validation and value validation is done properly.
- **Suggestions:** Please try to check address value in function, use safeMath library in some function and check return response of transfer and transferFrom method.