SMART CONTRACT AUDIT REPORT

For

PITcoin (Order #F081778D0E745)

Prepared By: Kishan Patel

Prepared For: aqoleg

Prepared on: 10/07/2020

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

The project has 1 file. It contains approx 296 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 Ethereum'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 Ether 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 ether to a contract

While implementing “selfdestruct” in smart contract, it sends all the ether 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.

```
17  library Math {
18      /// @return uint256 = a + b
19      function add(uint256 a, uint256 b) internal pure return
20          uint256 c = a + b;
21          assert(c >= a);
22          return c;
23      }
24
25      /// @return uint256 = a - b
26      function sub(uint256 a, uint256 b) internal pure return
```

- **Good required condition in functions:-**
  - Here you are checking that after subtracting tokens from totalSupply totalSupply is not going to zero.

```
118     function sell(uint256 _tokens) external {
119         // w*pr = T + T*ppt/m - P + R
120         // w*pr = T-t + (T-t)*(ppt + f*m/(T-t))/m - (P - ((t-f) + t
121
122         uint256 fee = _tokens.div(10);
123         uint256 withdraw = _tokens.sub(fee);
124
125         balanceOf[msg.sender] = balanceOf[msg.sender].sub(_tokens);
126         totalSupply = totalSupply.sub(_tokens);
127         require(totalSupply != 0, "zero total supply");
128         emit Transfer(msg.sender, address(0), _tokens);
```

- Here you are checking that after adding refDividends to dividends. A dividend is not zero.

```
140     function withdraw() external {
141         // w*pr = T + T*ppt/m - P + R
142         // (w - (d+r)/pr)*pr = T + T*ppt/m - (P + d) + (R - r)
143
144         uint256 dividends = dividendsOf(msg.sender);
145         payoutsOf[msg.sender] = payoutsOf[msg.sender].signedAdd(d
146
147         dividends = dividends.add(refDividendsOf[msg.sender]);
148         refDividendsOf[msg.sender] = 0;
149
150         require(dividends != 0, "zero dividends");
151
```

- Here you are checking that after adding refDividends to allDividends. A allDividends is not zero.

```
158     function reinvest() external {
159         // w*pr = T + T*ppt/m - P + R
160         // w*pr = T+d+r + (T+d+r)*ppt/m - (P + d + (d+r)*ppt/m
161
162         uint256 dividends = dividendsOf(msg.sender);
163
164         uint256 allDividends = dividends.add(refDividendsOf[ms
165         refDividendsOf[msg.sender] = 0;
166
167         require(allDividends != 0, "zero dividends");
```

- Here you are checking that if ref address is not zero (or correct address value) then check it is not a sender and ref address has greater balance than refRequirement.

```
181     function buy(address _ref) public payable {
182         // w*pr = T + T*ppt/m - P + R
183         // with ref
184         // in*pr = t + f + r
185         // (w + in)*pr = T+t + (T+t)*(ppt + f*m/T)/m - (P + t*(ppt + f*m/T
186         // no ref
187         // in*pr = t + f
188         // (w + in)*pr = T+t + (T+t)*(ppt + f*m/T)/m - (P + t*(ppt + f*m/T
189         // first
190         // in*pr = t
191         // (w + in)*pr = T+t + (T+t)*ppt/m - (P + t*ppt/m) + R
192
193         uint256 tokens = msg.value.mul(price);
194         uint256 fee = tokens.div(10);
195         tokens = tokens.sub(fee);
196
197         if (_ref != address(0)) {
198             require(_ref != msg.sender, "_ref is sender");
199             require(balanceOf[_ref] >= refRequirement, "small balance");
```

- Here you are checking that a spender address is in a correct format.

```
243     function approve(address _spender, uint256 _value) public returns (bool) {
244         require(_spender != address(0), "zero _spender");
245
246         allowance[msg.sender][_spender] = _value;
247         emit Approval(msg.sender, _spender, _value);
248
249         return true;
250     }
```

- Here you are checking that to address is in a correct format.

```
270     function send(address _from, address _to, uint256 _value) private {
271         // w*pr = T + T*ppt/m - P + R
272         // newPpt = ppt + f*m/(T-v-f)
273         // w*pr = T-f + (T-f)*newPpt/m - (P - (v+f)*ppt/m + v*newPpt/m) + R
274
275         require(_to != address(0), "zero _to");
276         uint256 fee = _value.div(20);
```

• 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: Compiler version not fixed

=> In this file you have put “pragma solidity ^0.6.6;” which is not a good way to define compiler version.

=> Solidity source files indicate the versions of the compiler they can be compiled with. Pragma solidity ^0.6.6; // bad: compiles ^0.6.0 and above pragma solidity 0.6.6; //good: compiles 0.6.6 only

=> If you put(^) symbol then you are able to get compiler version 0.6.6 and above. But if you don’t use(^) symbol then you are able to use only 0.6.6

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.

=> Solidity latest version is 0.6.11.

7.2: 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 some functions you are not checking the value of address parameter:-

- **function:- clean ('_contract')**

```
113     function clean(address _contract, uint256 _value) external {
114         Erc20(_contract).transfer(msg.sender, _value);
115     }
116
```

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

- **function:- dividendsOf('_owner')**

```
253     function dividendsOf(address _owner) public view returns (uint256) {
254         // dividendsOf = balanceOf*profitPerToken/multiplicator - payoutsOf
255
256         uint256 a = balanceOf[_owner].mul(profitPerToken).div(multiplicator);
257         int256 b = payoutsOf[_owner];
258         // a - b
```

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

- **function:- send('_from')**

```
270     function send(address _from, address _to, uint256 _value) private {
271         // w*pr = T + T*ppt/m - P + R
272         // newPpt = ppt + f*m/(T-v-f)
273         // w*pr = T-f + (T-f)*newPpt/m - (P - (v+f)*ppt/m + v*newPpt/m) + R
274
275         require(_to != address(0), "zero _to");
276         uint256 fee = _value.div(20);
277         uint256 cost = _value.add(fee);
278
```

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

• 7.3: vulnerabilities in approve function:-

=> This is not a big issue and your contract is not susceptible to any risks because you are checking the balance and allowance in every function.

=>But it is good practice to check balance in approve function.

=>So here, negative value also gets accepted in this function for allowance.

```
242     /// @dev ERC20
243     function approve(address _spender, uint256 _value) public returns (bool) {
244         require(_spender != address(0), "zero _spender");
245
246         allowance[msg.sender][_spender] = _value;
247         emit Approval(msg.sender, _spender, _value);
248
249         return true;
250     }
```

- **Solution:-**
  1) require (_value <= balanceOf [_spender]);

• Summary of the Audit

Overall the code is well and performs good.

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 and check address.