Global Digital Opportunities

GDO Infotech Private Limited
Amreli – Surat, Gujarat, INDIA - 365410
Web: www.gdo.co.in **Email**: contact@gdo.co.in
Ph: +91 9712144344

SMART CONTRACT AUDIT REPORT

For

Topscoin (TPC) (Order # FO711C80EA5C5)

Prepared By: Yogesh Padsala

Prepared For: Topscoin.net

Prepared on: 28/07/2018

Table of Content

1. 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.

2. Overview of the audit

The project has 1 file TPCToken.sol. It contains approx 66 lines of Solidity code. All the functions and state variables are **not** well commented using the natspec documentation, but that does not create any vulnerability.

3. 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.

3.1: 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 not** 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.

3.2: 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.

3.3: Visibility & Delegatecall

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

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.

3.4: 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.

3.5: 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.

4. Good things in smart contract

4.1 _transfer Function:-

```
21
22    function _transfer(address _from, address _to, uint _value) internal {
23        require(_to != 0x0);
24        require(balanceOf[_from] >= _value);
25        require(balanceOf[_to] + _value > balanceOf[_to]);
26        uint previousBalances = balanceOf[_from] + balanceOf[_to];
27        balanceOf[_from] -= _value;
28        balanceOf[_to] += _value;
29        Transfer(_from, _to, _value);
30        assert(balanceOf[_from] + balanceOf[_to] == previousBalances);
31    }
32
```

- Here you are checking address of receiver and also balance of sender.
- This is good validations to prevent underflow and overflow attacks.

4.2 transferFrom Function:-

```
36
37    function transferFrom(address _from, address _to, uint256 _value) public returns (bool success) {
38        require(_value <= allowance[_from][msg.sender]);
39        allowance[_from][msg.sender] -= _value;
40        _transfer(_from, _to, _value);
41        return true;
42    }
```

- Here you are checking the allowance value of sender, which is a good thing

4.3 burn function:-

```
48
49    function burn(uint256 _value) public returns (bool success) {
50        require(balanceOf[msg.sender] >= _value);
51        balanceOf[msg.sender] -= _value;
52        totalSupply -= _value;
53        Burn(msg.sender, _value);
54        return true;
55    }
56
```

- Here you are checking balance of caller then this function allow sender to burn token, which is a good thing

4.4 burnFrom Function:-

```
57    function burnFrom(address _from, uint256 _value) public returns (bool success) {
58        require(balanceOf[_from] >= _value);
59        require(_value <= allowance[_from][msg.sender]);
60        balanceOf[_from] -= _value;
61        allowance[_from][msg.sender] -= _value;
62        totalSupply -= _value;
63        Burn(_from, _value);
64        return true;
65    }
```

- In this function, you first check the balance of "_from" address.
- Then allowance of msg.sender for "_from" address.
- If these both conditions would pass then and then token would be burned from address, which is a good validations.

5. Critical vulnerabilities found in the contract

=> No critical vulnerabilities found

6. Medium vulnerabilities found in the contract

=> No Medium vulnerabilities found

7. Low severity vulnerabilities found

7.1 vulnerabilities in approve function

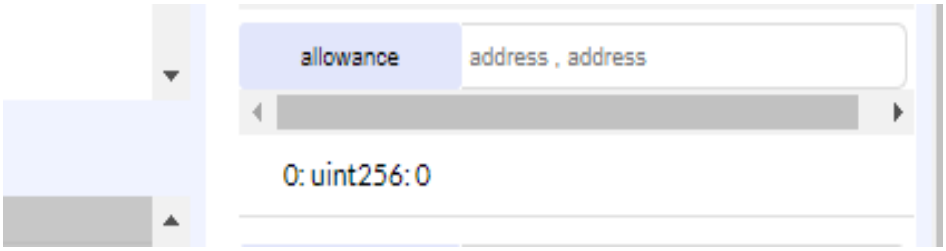
=> This is not a big issue and your contract is not susceptible to any risks because you are checking balance and allowance in every function.

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

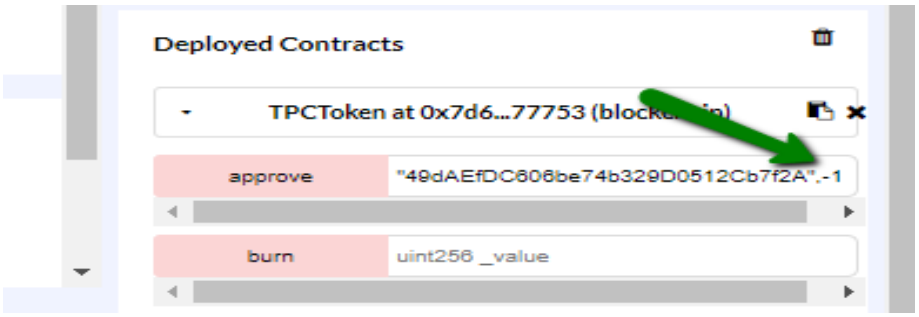
=>So here, negative value also gets accepted in this function for allowance. So allowance of any user goes wrong.

```
43
44    function approve(address _spender, uint256 _value) public returns (bool success) {
45        allowance[msg.sender][_spender] = _value;
46        return true;
47    }
48
```

- **Underflow and overflow attack**
    - Allowance before approve

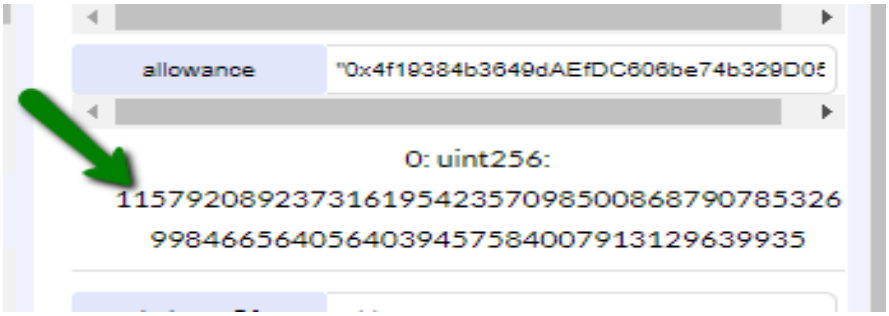


    - Calling approve with negative value.

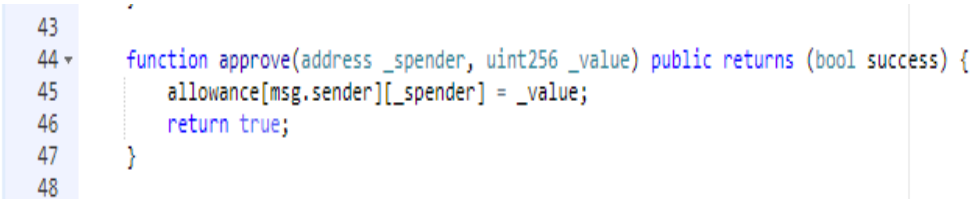


    - Transaction hash
        - https://rinkeby.etherscan.io/tx/0xee5b6c9931f9cd0d7df27f04be218b8228479952152f81317f78832d16a0df94

    - Allowance after approve



Solution:-



```
43
44    function approve(address _spender, uint256 _value) public returns (bool success) {
45        allowance[msg.sender][_spender] = _value;
46        return true;
47    }
48
```

- In approve function you have to put one condition.

  require(_value <= balances[msg.sender]);

- By this way, user only approves the amount which he has in the balance.

7.2: Compiler version not fixed

=> In this file you have put "pragma solidity ^0.4.16;" which is not good way to define compiler version.

=> Solidity source files indicate the versions of the compiler they can be compiled with.

pragma solidity ^0.4.16; // bad: compiles w 0.4.16 and above

pragma solidity 0.4.16; // good : compiles w 0.4.16 only

=> If you put (^) symbol then you are able to get compiler version 0.4.16 and above. But if you don't use (^) symbol then you are able to use only 0.4.16 version. And if there is some changes come in compiler and you use old version then some issue may come at deploy time.

=> And try to use latest version of solidity compiler (0.4.24).

7.3 Short address attack

=> This is not a big issue in the solidity, because nowadays security is increased in new solidity version. But it is good practice to check for the short addresses.

=> In some functions you are not checking value of address parameter

- **function:- transferFrom('_from' and '_to')**

```
37  function transferFrom(address _from, address _to, uint256 _value) public returns (bool success) {
38      require(_value <= allowance[_from][msg.sender]);
39      allowance[_from][msg.sender] -= _value;
40      _transfer(_from, _to, _value);
41      return true;
42  }
```

  - **Solution:-**

    require(_to != 0x0);

    require(_from != 0x0);

- **Function:-burnFrom('_form')**

```
57  function burnFrom(address _from, uint256 _value) public returns (bool success) {
58      require(balanceOf[_from] >= _value);
59      require(_value <= allowance[_from][msg.sender]);
60      balanceOf[_from] -= _value;
61      allowance[_from][msg.sender] -= _value;
62      totalSupply -= _value;
63      Burn(_from, _value);
64      return true;
65  }
```

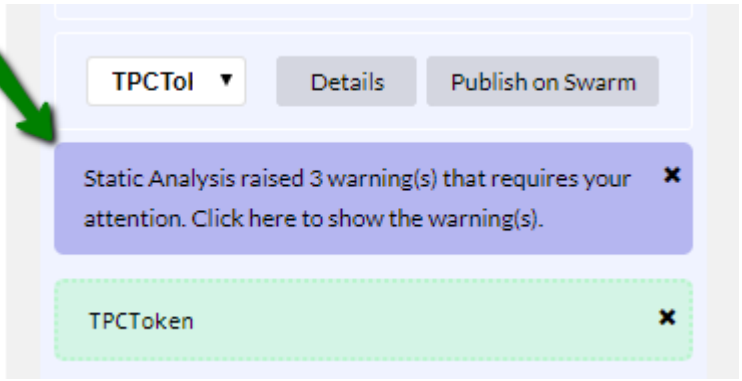
  - **Solution:-**

    require(_from != 0x0);

8. Summary of the Audit

Overall the code is well commented, and performs good data validations.

The compiler also displayed 3 warnings:



Now, we checked those warnings are due to their static analysis, which includes like gas errors and all. So, it is important to supply correct gas values while calling various functions.

Those warnings can be safely ignored as should be taken care while calling the smart contract functions.

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:** You are not using safemath library. It is very good practice to use it in as many methodical calculations as possible.