SMART CONTRACT AUDIT REPORT

For

Bountie Token (Order # FO711C85)

Prepared By: Yogesh Padsala

Prepared on: 22/09/2018

audit@gdo.co.in

Prepared For: Bountie Token

https://www.bountie.io

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 BoutieContract-Mainnet_Live.sol. It contains approx **295** lines of Solidity code including external files. All the functions and state variables are **not** well commented using the natspec documentation; however that does not raise any vulnerability.

The audit was performed by Yogesh Padsala, from GDO Infotech Pvt Ltd. Yogesh has extensive work experience of developing and auditing the smart contracts.

The audit was based on the solidity compiler 0.4.25+commit.59dbf8f1 with optimization enabled compiler in remix.ethereum.org

This audit was also performed the verification of the details in whitepaper, located at: https://www.bountie.io/tokensale/assets/doc/whitepaper-en.pdf

3. Attacks tested on 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

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

3.2: Short address attack

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

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

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

3.5: Forcing ether to a contract

Here, the Smart Contract’s balance has never been used as guard, which mitigated this vulnerability

3.6: Denial Of Service (DOS)

There is no process consuming loops in the contracts which can be used for DoS attacks. Also, there is no progressing state based on external calls, and thus this contract is not prone to DoS.

4. Good things in smart contract

4.1 Ability to change the wallet address

This contract implements CrowdSale_Change_ReceiveWallet function which enables contract owner to change the wallet address. This is very important as this safety standard, and useful in event of loss or theft of the wallet private keys.

4.2 Ability to control the token transfer

This contract implements startTransferToken and stopTransferToken functions which enable admin to place safeguard in event of any security violation event.

4.3 Greater control over ICO process

It is really good thing for admin to be able to start ICO, stop ICO, change wallet address, etc.

4.4 Ability to mint tokens manually.

This contract enables admin to generate new tokens manually. This is really useful in even of owner need to use tokens apart from ICO.

4.4 Good things in the code

- transferFrom function

```
125     function transferFrom( address _from, address _to, uint256 _amount ) public c
126         require( _to != 0x0);
127         require(!lockstatus);
128         require(balances[_from] >= _amount && allowed[_from][msg.sender] >= _amou
129         balances[_from] = (balances[_from]).sub(_amount);
130         allowed[_from][msg.sender] = (allowed[_from][msg.sender]).sub(_amount);
131         balances[_to] = (balances[_to]).add(_amount);
```

Address “_to” parameter is checked, as well as allowance of caller is checked before doing the transfer.

- approve function

```
137     function approve(address _spender, uint256 _amount)public onlyFinishedICO ret
138         require(!lockstatus);
139         require( _spender != 0x0);
140         allowed[msg.sender][_spender] = _amount;
141         emit Approval(msg.sender, _spender, _amount);
```

Address "_spender" is checked for validity, and lockstatus also checked.

- Transfer function

```
152     function transfer(address _to, uint256 _amount)public onlyFinishedICO returns
153         require(!lockstatus);
154         require( _to != 0x0);
155         require(balances[msg.sender] >= _amount && _amount >= 0);
156         balances[msg.sender] = (balances[msg.sender]).sub(_amount);
157         balances[_to] = (balances[_to]).add(_amount);
158         emit Transfer(msg.sender, _to, _amount);
```

Validity of address "_to" is checked. It performs all the required validations first before doing the transfer event.

- mintContract function

```
205     function mintContract(address from, address receiver, uint256 tokenQuantity
206         require(tokenQuantity > 0);
207         mintedTokens = mintedTokens.add(tokenQuantity);
208         uint256 novumShare = tokenQuantity * 4 / 65;
209         uint256 userManagement = tokenQuantity * 31 / 65;
210         balances[novumAddress] = balances[novumAddress].add(novumShare);
211         balances[owner] = balances[owner].add(userManagement);
212         _totalsupply = _totalsupply.add(tokenQuantity * 100 / 65);
213         balances[receiver] = balances[receiver].add(tokenQuantity);
```

Balance is sent to the receiver address only afte doing all the validations.

- fallback function

```
194     function () public payable onlyICO {
195         require(msg.value != 0 && msg.sender != 0x0);
196         require(!stopped && msg.sender != owner);
197         uint256 tokenPrice = calculatePrice();
198         uint256 tokenQuantity = (msg.value).mul(tokenPrice);
199         ETHcollected = ETHcollected.add(msg.value);
200         require(ETHcollected <= maxCap);
201         mintContract(address(this), msg.sender, tokenQuantity);
```

Ether value is checked. ICO should not be stopped and caller should not be owner. Also, ether collected must not be more than max cap.

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: Old compiler version

=> You are using Solidity version is 0.4.24.

=> Solidity latest version is 0.4.25.

7.2 Style guide violation

Although this is not a security vulnerability, but function and event names usually start with a lower- and uppercase letter respectively:

```
function foo(); // good

event LogFoo(); // good
```

Violating the style guide decreases readability and leads to confusion. We noticed the names of the functions do not adhere with that.

Line number #250, #256, #261, #267

7.3 issues in approve function

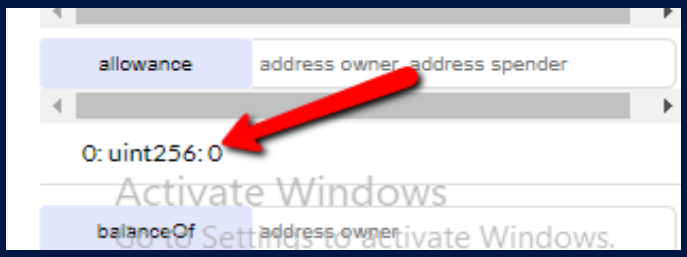
=> This is not a big issue and your contract is not susceptible to any risks because you are checking balance and allowance in every function.

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

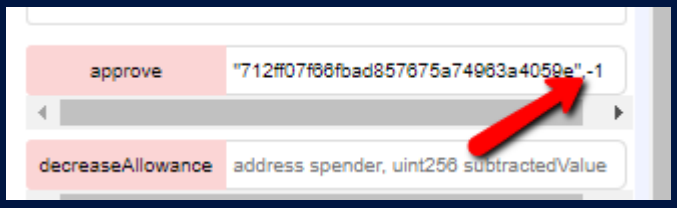
=>So here, negative value also gets accepted in this function for allowance. So allowance of any user goes wrong.

```
137     function approve(address _spender, uint256 _amount)public returns (bool suc
138         require(!lockstatus);
139         require( _spender != 0x0);
140         allowed[msg.sender][_spender] = _amount;
141         emit Approval(msg.sender, _spender, _amount);
```

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

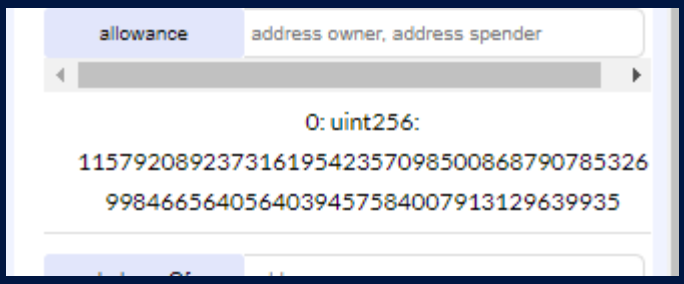


  - Calling approve with negative value.



  - Transaction hash
    https://rinkeby.etherscan.io/tx/0x24e2bd4ed5505900e43f283a3b7e0842ec1274efb7ea2db3d32325f336136cd0

- Allowance after approve



Solution:

- 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.

7.4 Unchecked Math

Safemath library is included, which is good thing. But at some place, it is not used.

This is not a big issue, as validations are done well. But it is good practice to use it at all the mathematical calculations.

Line numbers :- #212, #228, #209, #240, #232, #163, #212, #208, #236, #185, #240, #208, #228

Please implement Safemath at those places.

7.5 Implicit visibility level

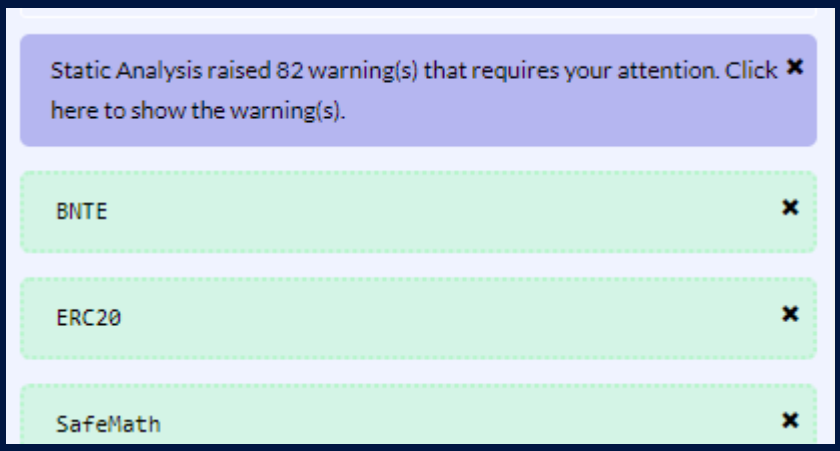
Although this not a big issue, but the visibility is not specified for variable at line number #77 and #78.

It is good practice to specify appropriate visibility to every state members.

8. Summary of the Audit

Overall the code performs good data validations as well as meets the calculations required in the whitepaper.

The compiler also displayed 82 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.

Please 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.

It is also encouraged to run bug bounty program and let community help to further polish the code to the perfection.