GDO Infotech Private Limited
Amreli – Surat – Poland – UK – Canada
Web: www.gdo.co.in **Email**: contact@gdo.co.in
Ph: +91 9712144344

SMART CONTRACT AUDIT REPORT

For

Giegle Contract(Order # FO711C80EA5C5)

Prepared By: Yogesh Padsala

Prepared For: Giegle

Prepared on: 17/07/2018

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 GIEGLE_CONTRACT.sol. It contains approx 266 lines of Solidity code. All the functions and state variables are **not** well commented using the natspec documentation, but that does not create any issues. Good comments make contract more readable to other community members.

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** check for overflows and underflows by using OpenZeppelin's SafeMath to mitigate this attack. However it has some concerns, which are discussed below.

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:-

```
157 ▾   function transfer(address _to, uint256 _value) public isRunning returns (bool success) {
158         require(_to != address(0));
159         require(balanceOf[msg.sender] >= _value);
160         require(balanceOf[_to].add(_value) >= balanceOf[_to]);
161         balanceOf[msg.sender] = balanceOf[msg.sender].sub(_value);
162         balanceOf[_to] = balanceOf[_to].add(_value);
163         emit Transfer(msg.sender, _to, _value);
164         return true;
165     }
```

- Here you are checking address variable of “to”.
- Here you are checking the balance of sender.

4.2 transferFrom Function:-

```
167 ▾     function transferFrom(address _from, address _to, uint256 _value) public isRunnin
168         require(_from != address(0) && _to != address(0));
169         require(balanceOf[_from] >= _value);
170         require(balanceOf[_to].add(_value) >= balanceOf[_to]);
171         require(allowance[_from][msg.sender] >= _value);
172         balanceOf[_to] = balanceOf[_to].add(_value);
173         balanceOf[_from] = balanceOf[_from].sub(_value);
174         allowance[_from][msg.sender] = allowance[_from][msg.sender].sub(_value);
175         emit Transfer(_from, _to, _value);
176         return true;
177     }
```

- Here you are checking validation of address parameter “to” and “from”.
- Balance and allowance of sender.
- You are also reducing the allowance of sender at the end of function.

4.3 approve function:-

```
178
179 ▾   function approve(address _spender, uint256 _value) public isRunning returns (bool success) {
180         require(_spender != address(0));
181         require(_value <= balanceOf[msg.sender]);
182         require(_value == 0 || allowance[msg.sender][_spender] == 0);
183         allowance[msg.sender][_spender] = _value;
184         emit Approval(msg.sender, _spender, _value);
185         return true;
186     }
187
```

- Here you have put validation about address parameter of “_spender”.
- This function accepts value below the balance of function caller.
- It also checks the first allowance of caller must 0.

4.4 setStage Function:-

```
177   // set new ico stage
178   function setStage(uint256 _stage, uint256 _startDate, uint256 _endDate, uint256 _fund, uint256 _bonus) external
179
180       // current time must be greater then previous ico stage end time
181       require(now > ico.icoEndDate);
182       // current stage must be greater then previous ico stage
183       require(_stage > ico.icoStage);
184       // current time must be less then start new ico time
185       require(now < _startDate);
186       // new ico start time must be less then new ico stage end date
187       require(_startDate < _endDate);
188       // owner must have fund to start the ico stage
189       require(balanceOf[msg.sender] >= _fund);
190
```

- Your validations are good in this function.
- You are checking that starting time is greater than now and endtime. _stage value is bigger than past ico stage and caller is not able to give "_fund" value bigger than his balance, which is good thing.

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 Short address

```
147
148   function _transfer(address _wallet, address _to, uint256 _value) internal {
149       require(_to != address(0));
150       require(balanceOf[_wallet] >= _value);
151       require(balanceOf[_to].add(_value) >= balanceOf[_to]);
152       balanceOf[_wallet]  = balanceOf[_wallet].sub(_value);
153       balanceOf[_to]      = balanceOf[_to].add(_value);
154       emit Transfer(_wallet, _to, _value);
155   }
```

=> This is not a big issue in solidity, but it is good practice to validate address of "_wallet" in function _transfer and "_to" in function sendToken.

7.2 Underflow attack :-

```
196 ▾   function sendToken(address _to, uint _amount) public isOwner isRunning returns(bool) {
197         require(msg.sender != address(0));
198         require(_amount > 0);
199         uint256 stageIs = getStage();
200         require(stageIs != 10);
201
```

=>Again, this is not a big issue. You have put condition require(_amount > 0). But that condition does not prevent negative value. In another words, if I input negative value, then that condition will pass, as all the negative values will be consider as longer positive values in solidity.

7.3 Implicit visibility level

=> This is not a big issue in the solidity, because if you do not put any visibility, then it will automatically take “public”. But it is good practice to specify visibility at every variables and functions.

```
42
43      uint256 fundTeam            = 750000000  * (10 ** uint256(decimals));
44      uint256 fundAdvisors        = 125000000  * (10 ** uint256(decimals));
45      uint256 fundBounty          = 125000000  * (10 ** uint256(decimals));
46      uint256 fundPlatform        = 1450000000 * (10 ** uint256(decimals));
47      uint256[] DPTbonus    = [100, 50, 25, 0];
48
49      address owner;
50      address walletTeam;
51      address walletAdvisors;
52      address walletBounty;
53      address walletPlatform;
```

Solution:-

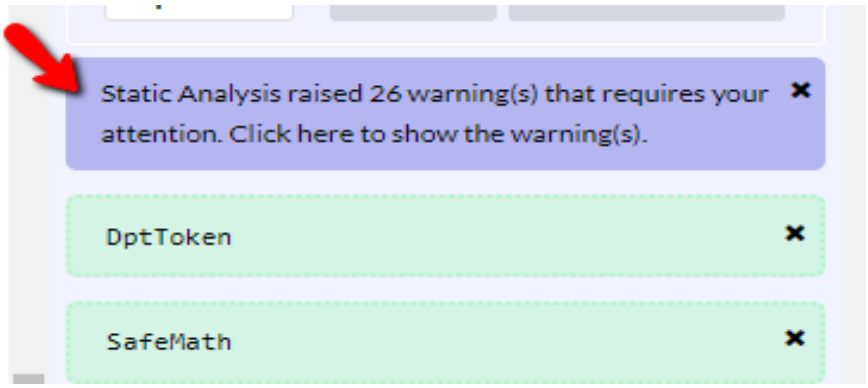
1) For #43.
    a. uint256 public fundTeam = 750000000 * (10 ** uint256(decimals));
2) For #44
    a. uint256 public fundAdvisors = 125000000 * (10 ** uint256(decimals));
3) For #45.
    a. uint256 public fundBounty = 125000000 * (10 ** uint256(decimals));

4) For #46.
    a. uint256 public fundPlatform = 1450000000 * (10 ** uint256(decimals));
5) For #47.
    a. uint256[] public DPTbonus = [100, 50, 25, 0];
6) For #49
    a. address public owner;
7) For #50
    a. address public walletTeam;
8) For #51
    a. address public walletAdvisors;
9) For #52
    a. address public walletBounty;
10) For #53
    a. address public walletPlatform;

8. Summary of the Audit

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

The compiler also displayed 26 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.

Even this code is for Gigle contract, so please do not forget to change the name of contract to GIEGLE, or something appropriate (unless you purposefully want to keep it this way).

```
29
30  contract DptToken {
31      using SafeMath for uint256;
32
33      string  public name     = "Giegle";
34      string  public symbol   = "GIEGLE";
35      uint256 public decimals = 18;
36
```

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 ;)).