Global Digital Opportunities

SMART CONTRACT AUDIT REPORT

For

The Token (Order # FO711C89)

Prepared By: Yogesh Padsala

Prepared For: The Token

Prepared on: 29/09/2018

audit@gdo.co.in

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 following five files:

- TheToken.sol
- TokenSale.sol
- SplitPayment.sol
- SafeMath.sol
- Ownable.sol

It contains approx **1090** lines of Solidity code. All the functions and state variables are well commented using the natspec documentation.

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 according to the basic_flow.2018-09-24.txt file provided.

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.

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 the smart contract

4.1 Higher degree of the crowsale control

This contract implements various functions which enables owner to do various tasks as such as pause, unpause, extract ether, etc.

4.2 Admin can process ACH payments

This contract implements feature which enables owner to process token distribution manually if they receive payments through ACH.

4.3 Split payment facility

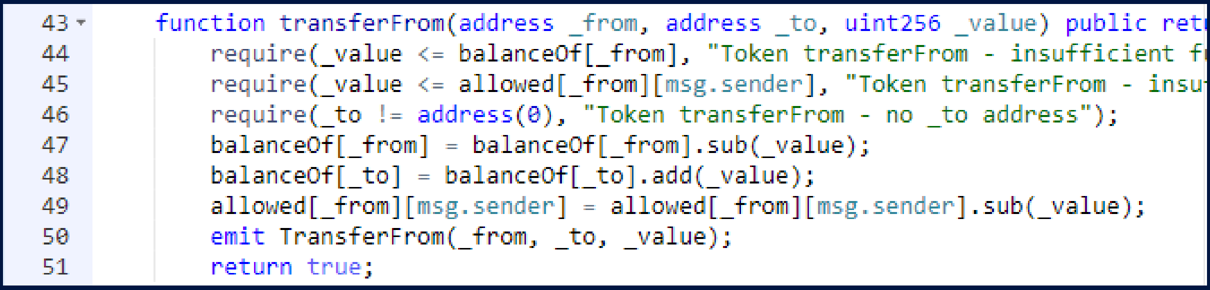
This contract has facility to add multiple payees and split the fund received in the crowdsale contract.

4.4 Good validations

This contract processes loop with good validations as well as functions are having good require conditions. The dorequire() function is really good as it emits event in any error or issue, so client can get notified for it.

4.4 Good things in the code

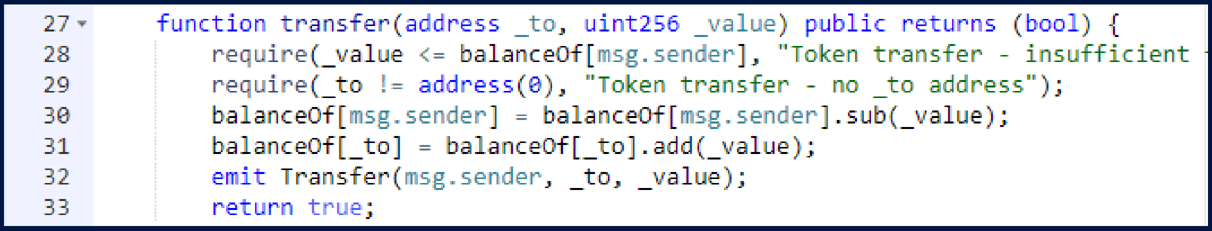
- transferFrom function - TheToken.sol



```
function transferFrom(address _from, address _to, uint256 _value) public ret
    require(_value <= balanceOf[_from], "Token transferFrom - insufficient f
    require(_value <= allowed[_from][msg.sender], "Token transferFrom - insuf
    require(_to != address(0), "Token transferFrom - no _to address");
    balanceOf[_from] = balanceOf[_from].sub(_value);
    balanceOf[_to] = balanceOf[_to].add(_value);
    allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
    emit TransferFrom(_from, _to, _value);
    return true;
```

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

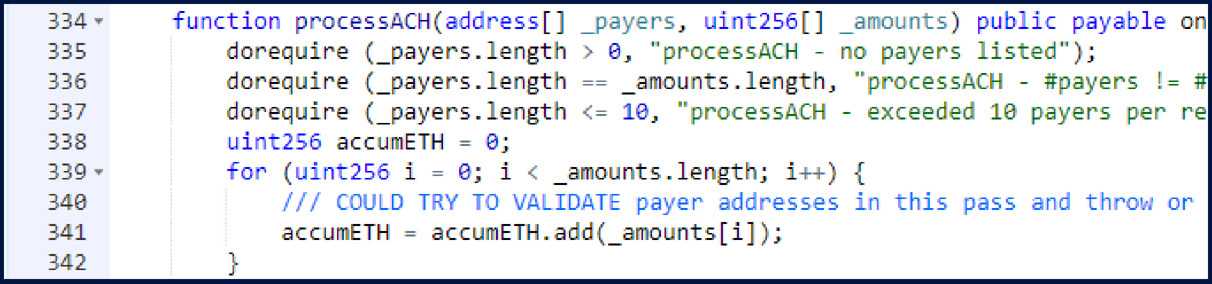
- Transfer function - TheToken.sol



```
    function transfer(address _to, uint256 _value) public returns (bool) {
        require(_value <= balanceOf[msg.sender], "Token transfer - insufficient
        require(_to != address(0), "Token transfer - no _to address");
        balanceOf[msg.sender] = balanceOf[msg.sender].sub(_value);
        balanceOf[_to] = balanceOf[_to].add(_value);
        emit Transfer(msg.sender, _to, _value);
        return true;
```

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

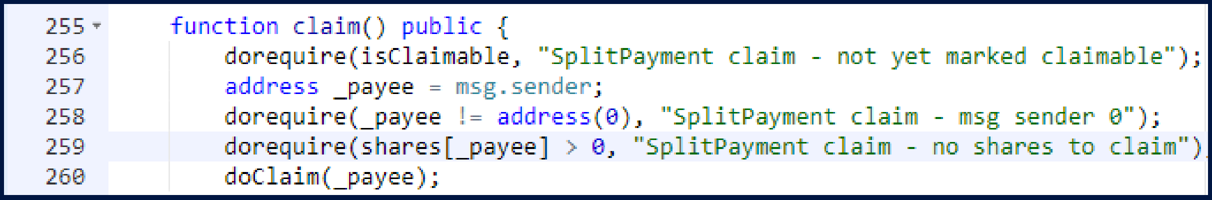
- processACH function - TokenSale.sol



```
    function processACH(address[] _payers, uint256[] _amounts) public payable on
        dorequire (_payers.length > 0, "processACH - no payers listed");
        dorequire (_payers.length == _amounts.length, "processACH - #payers != #
        dorequire (_payers.length <= 10, "processACH - exceeded 10 payers per re
        uint256 accumETH = 0;
        for (uint256 i = 0; i < _amounts.length; i++) {
            /// COULD TRY TO VALIDATE payer addresses in this pass and throw or
            accumETH = accumETH.add(_amounts[i]);
        }
```

Validity of _payer array is checked and most importantly the length of the array is restricted to 10 which is really good. Because uncontrolled for loop execution cause really bad things.

- claim function - SplitPayment.sol



```
    function claim() public {
        dorequire(isClaimable, "SplitPayment claim - not yet marked claimable");
        address _payee = msg.sender;
        dorequire(_payee != address(0), "SplitPayment claim - msg sender 0");
        dorequire(shares[_payee] > 0, "SplitPayment claim - no shares to claim")
        doClaim(_payee);
```

It is required if it is claimable and also the validity of the _payee is checked.

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 in all the files

=> You are using Solidity version is 0.4.24.

=> Solidity latest version is 0.4.25.

7.2 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.

SplitPayment.sol: #78, #73, #76, #79

TokenSale.sol: #122, #124, #125, #263, #299, #119, #168, #181, #124, #182

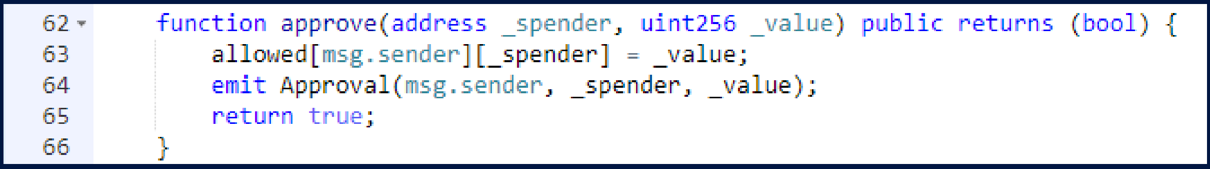
Please implement Safemath at those places.

7.3 issues in approve function – TheToken.sol

=> 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.

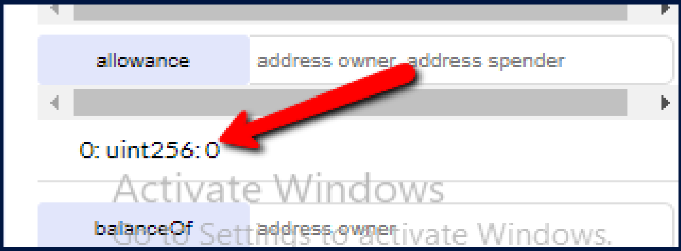
=>But it is good practice to check balance in approve function.

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



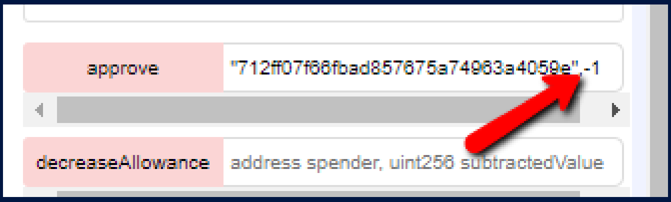
```
function approve(address _spender, uint256 _value) public returns (bool) {
    allowed[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);
    return true;
}
```

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



```
allowance | address owner, address spender
0: uint256: 0
balanceOf | address owner
```

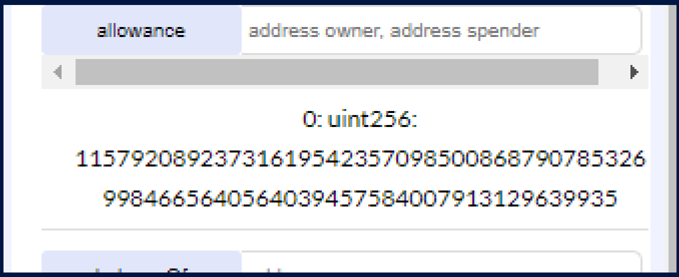
  - Calling approve with negative value.



```
approve | "712ff07f66fbad857675a74963a4059e",-1
decreaseAllowance | address spender, uint256 subtractedValue
```

  - Transaction hash
    https://rinkeby.etherscan.io/tx/0x0591dff5738c74019a2c2f653a2fe15c11503be3cb822e17a922f4164208c79d

- Allowance after approve



```
0: uint256:
115792089237316195423570985008687907853269984665640564039457584007913129639935
```

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 Revert Instead of Require

Although this not a big issue and your contract is susceptible to any risks. But at some places, if (condition) {revert();} is used instead of require(condition);

Both does the same job, but use require() for better code readability

TokenSale.sol: #275, #283, #290, #311, #302

7.5 High Loop Iterations Possibilities – SplitPayment.sol

Function addPayees() at line number #164 is called only by admin and admin can only input fair amount of entries in the array.

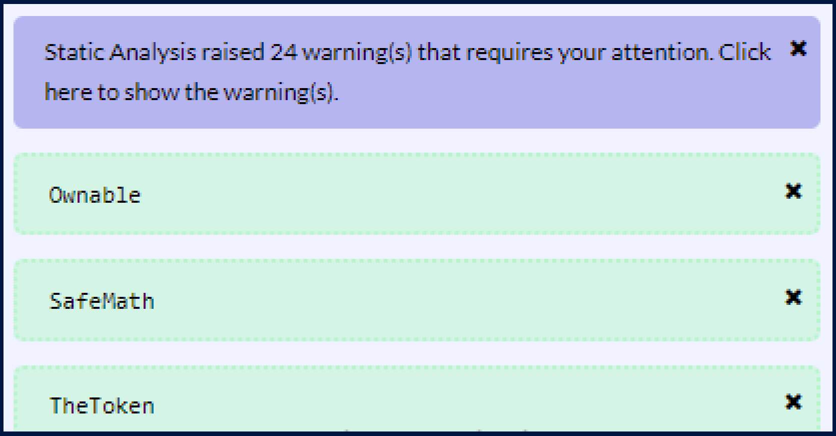
But, it is possibilities, even in error, the input array hold many entries which causes to eat out all the gas and hit the block maximum gas limit.

So, it is good to limit the array iterations to minimum in the code.

8. Summary of the Audit

Overall the code performs good data validations as well as meets the calculations according to the basic flow document.

The compiler also displayed 24 warnings:



Static Analysis raised 24 warning(s) that requires your attention. Click here to show the warning(s).

Ownable

SafeMath

TheToken

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.

There are many places in the contract where many variables are marked as "Private". One thing to understand is that, making variables private, does not make a them invisible. Miners have access to all contracts' code and data. Developers must account for the lack of privacy in Ethereum

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