SMART CONTRACT AUDIT REPORT

For

GFT Token (Order #FO71671EBEBA8)

Prepared By: Yogesh Padsala **Prepared For**: GFT Token

Prepared on: 20/06/2018

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 GFTToken.sol. It contains approx 302 lines of Solidity code. All the functions and state variables are not well commented using the natspec documentation. Some functions have comments and some do not.

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 (discussed below in detail).

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.

This contract **is vulnerable** to this attack (discussed below). It **does NOTHING to prevent** the *short address attack* during **ICO** or in an **exchange** (it will also depend if the ICO contract or DApp to check the length of data. If they don't, then short address attacks would drain out this coin from the exchange).

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.

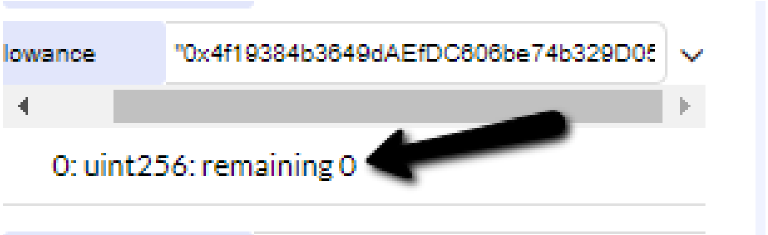
4. Critical vulnerabilities found in the contract

4.1: Underflow & Overflow attack:

=>In your contract some functions accept negative values.

=>Function name: - approve.

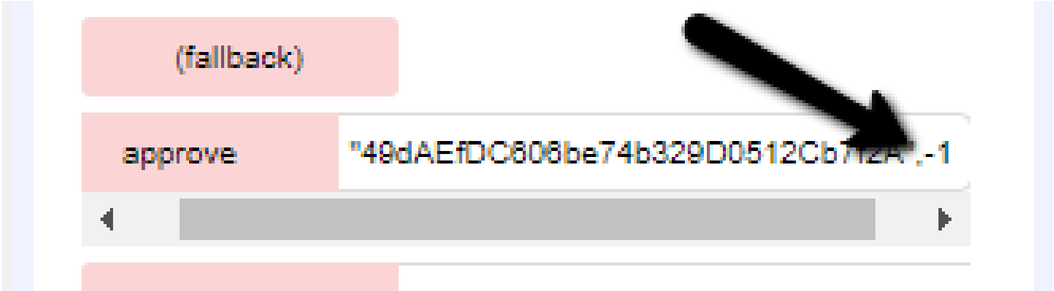
- **Approve**
  - Allowance value in starting.



lowance "0x4f19384b3649dAEfDC606be74b329D05

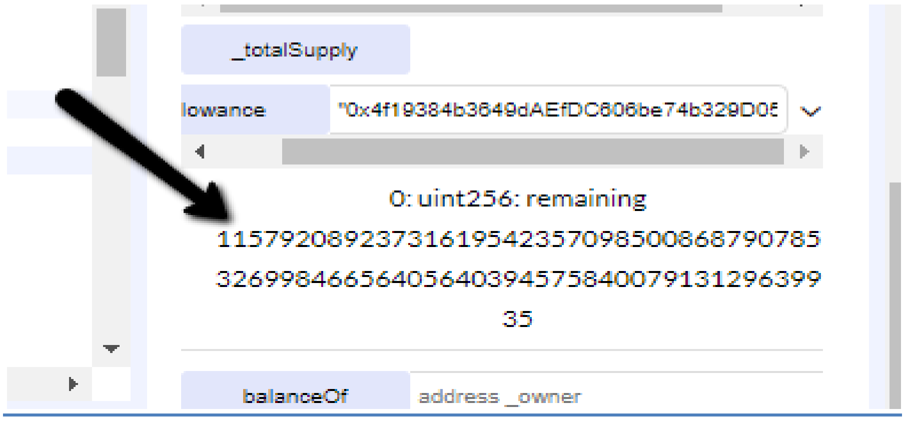
0: uint256: remaining 0

  - Now calling approve function with negative value.



approve "49dAEfDC606be74b329D0512Cb712A",-1

  - Transaction Hash:- https://rinkeby.etherscan.io/tx/0x4ad21be54bf917c3de54ba928bc7d7bcf716757f325237b432f42618466a92f7
  - Allowance after negative approves.

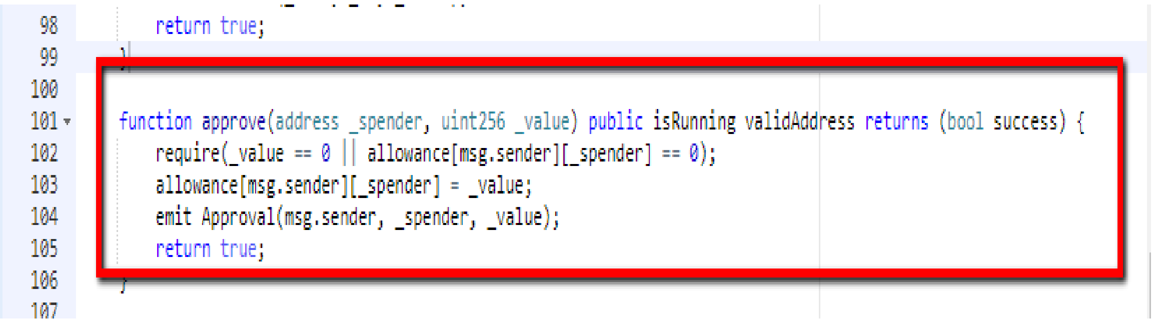


lowance "0x4f19384b3649dAEfDC606be74b329D05

```
0: uint256: remaining
115792089237316195423570985008687907853269984665640564039457584007913129639935
```

balanceOf address _owner

Solution:-



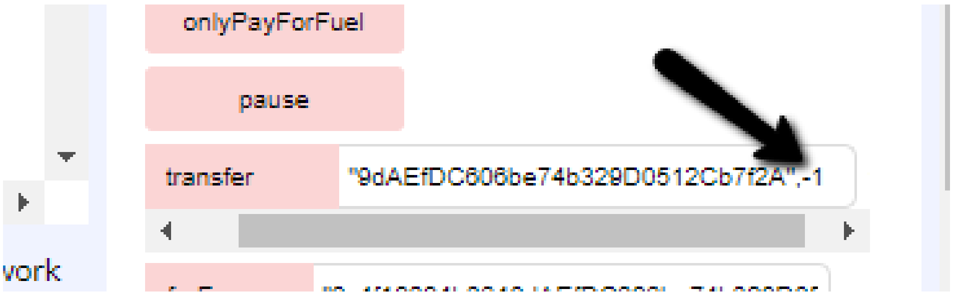
```
    return true;
}

function approve(address _spender, uint256 _value) public isRunning validAddress returns (bool success) {
    require(_value == 0 || allowance[msg.sender][_spender] == 0);
    allowance[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);
    return true;
}
```

- In approve functions you have to put one condition.

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

- By this way, users get approval of not more than their balance.
- In debug_setTotalCreated function, you have to take care of variable "_value" value when you call it.

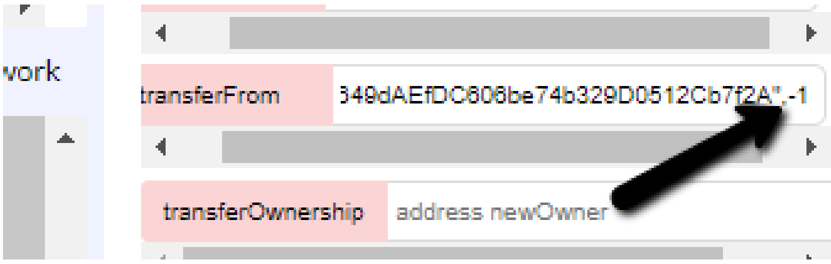
❖ **Transfer**



transfer "9dAEfDC606be74b329D0512Cb7f2A",-1

- Transaction Hash :- https://rinkeby.etherscan.io/tx/0x0c765723cbb835d3de9097acdd0041afaf5ea71ee2712dd79a613ac3733c82b8.

❖ **transferFrom**



transferFrom 349dAEfDC606be74b329D0512Cb7f2A",-1

transferOwnership address newOwner

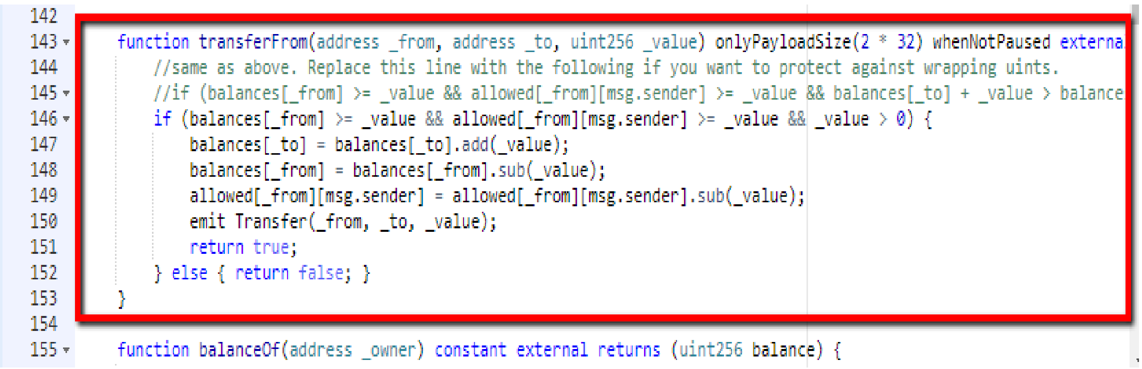
- Transaction Hash :- https://rinkeby.etherscan.io/tx/0xc7563471b4e31aa6744f8917e608188262f015dc5d26e17a41b85ed9c7595b47.
- **Solution For Transfer and Transferfrom**

=>You have to take care of the amount value before calling the smart contract in your application.

4.2: Short address attack

=>In your contract, some functions do not check the value of address variable.

=>Function name: - transferFrom, approve,transfer.



```
function transferFrom(address _from, address _to, uint256 _value) onlyPayloadSize(2 * 32) whenNotPaused externa
    //same as above. Replace this line with the following if you want to protect against wrapping uints.
    //if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && balances[_to] + _value > balance
    if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && _value > 0) {
        balances[_to] = balances[_to].add(_value);
        balances[_from] = balances[_from].sub(_value);
        allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
        emit Transfer(_from, _to, _value);
        return true;
    } else { return false; }
}

function balanceOf(address _owner) constant external returns (uint256 balance) {
```

- You are not checking the value of “_from” and “_to” variable in transferFrom.
- You are not checking value of “_to” variable in transfer function.
- You are not checking value of “_spender” variable in approve function.
- Anyone can request these function with short address.

Solution:-

- Add only one line in these functions.
- **require(address parameter != address(0));**

5. Medium vulnerabilities found in the contract

=> Congratulations. No such vulnerabilities found.

6. Low severity vulnerabilities found

6.1: Compiler version not fixed

=> In this file you have put "pragma solidity ^0.4.24;" which is not good way to define compiler version. Please remove caret (^) symbol unless it is really intended.

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

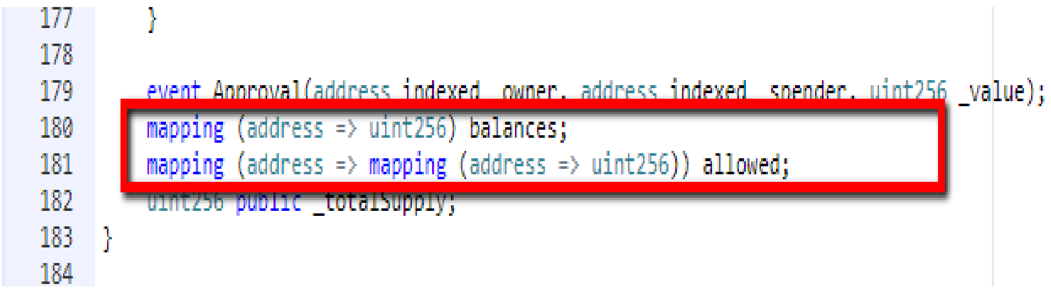
pragma solidity ^0.4.24; // bad: compiles w 0.4.24 and above

pragma solidity 0.4.24; // good : compiles w 0.4.24 only

=> If you put (^) symbol then you are able to get compiler version 0.4.24 and above. But if you don't use (^) symbol then you are able to use only 0.4.24 version. And if there is some changes come in compiler and you use old version then some issue may come at deploy time.

6.2: Implicit visibility level

=>This is not a big issue in solidity. So, if you do not define any visibility level then it automatically takes public.



```
177     }
178
179     event Approval(address indexed _owner, address indexed _spender, uint256 _value);
180     mapping (address => uint256) balances;
181     mapping (address => mapping (address => uint256)) allowed;
182     uint256 public _totalSupply;
183 }
184
```

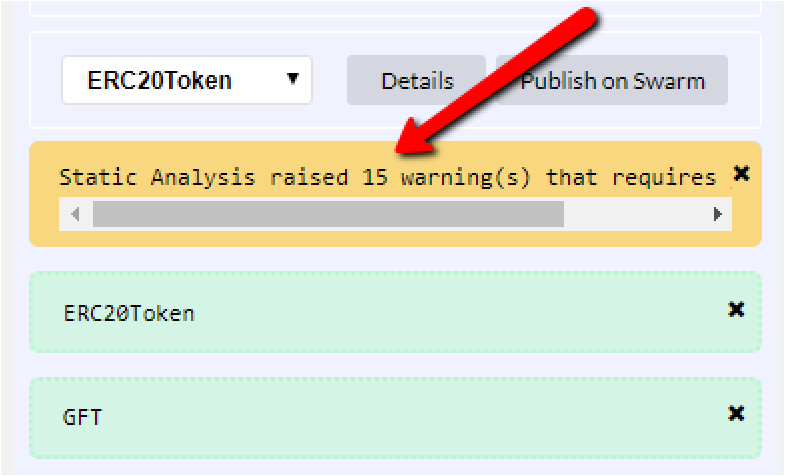
=>But it is good practice to specify visibility at every variables and functions.

=> Lines numbers #180, #181

7. Summary of the Audit

Overall the code is well commented, apart from some functions. But that does not trigger any vulnerabilities.

The compiler also displayed 15 warnings:



```
Static Analysis raised 15 warning(s) that requires
ERC20Token
GFT
```

Now, we checked those warnings, which were due to their static analysis, and which includes like gas errors and all.

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

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 ;)).

Try to check the address and value of token externally before sending to the solidity code.

You are using constant function for viewing the information it's ok now because constant is alias of the view. But it's good thing to use view function for viewing smart contract information. For more details: https://ethereum.stackexchange.com/questions/25200/solidity-what-is-the-difference-between-view-and-constant/25202