SMART CONTRACT AUDIT REPORT

For

Circa Token (Order #FO711C80EA5C5)

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

Prepared on: 12/07/2018

Table of Content

1. Disclaimer

2. Overview of the audit

3. Attacks made to the contract

4. Good things in smart contract

5. Critical vulnerabilities found in the contract

6. Medium vulnerabilities found in the contract

7. Low severity vulnerabilities found in the contract

8. Summary of the audit

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 **2 files Circa.sol** and **CircaICO.sol** it contains approx 566 lines of Solidity code. All the functions and state variables are well commented using the natspec documentation.

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.

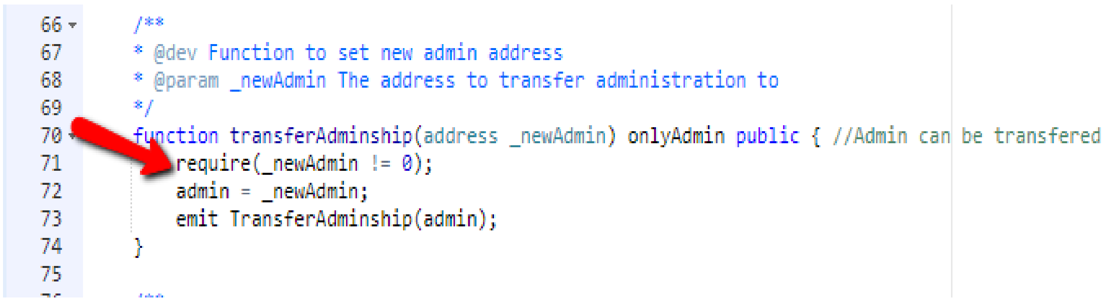
3.5: Forcing ether to a contract

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 Circa.sol

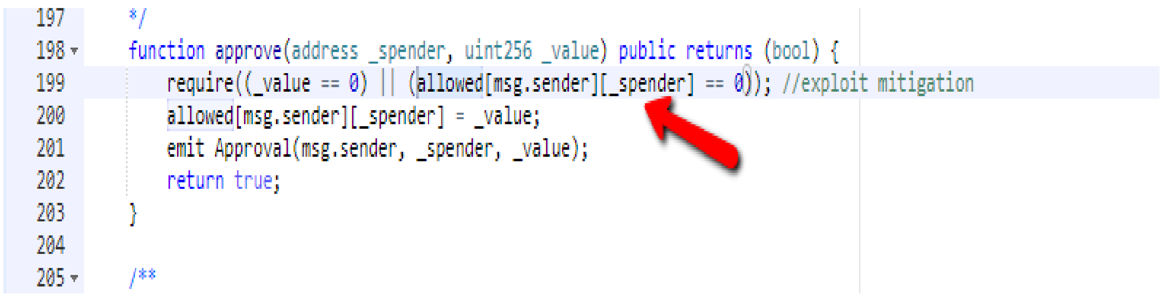
4.1.1 _transferOwnership Function:-



```
/**
* @dev Function to set new admin address
* @param _newAdmin The address to transfer administration to
*/
function transferAdminship(address _newAdmin) onlyAdmin public { //Admin can be transfered
    require(_newAdmin != 0);
    admin = _newAdmin;
    emit TransferAdminship(admin);
}
```

- Here you are checking that the value of "_newOwner" is valid address or not, which a good thing is.

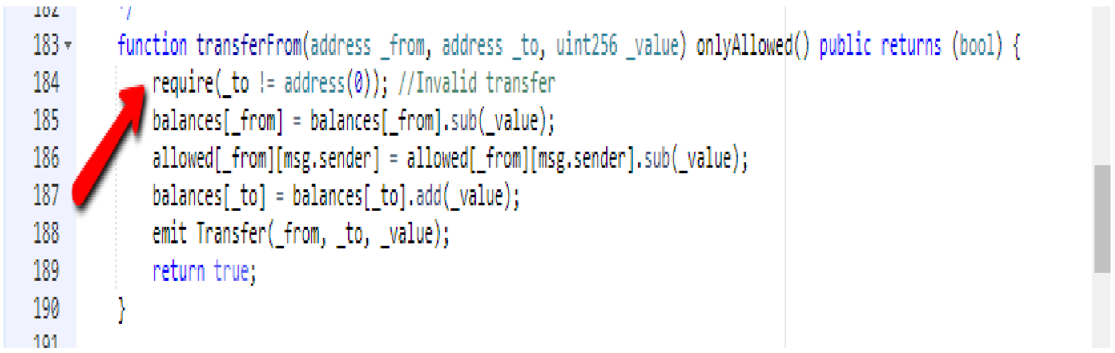
4.1.2 **approve Function:-**



```
*/
function approve(address _spender, uint256 _value) public returns (bool) {
    require((_value == 0) || (allowed[msg.sender][_spender] == 0)); //exploit mitigation
    allowed[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);
    return true;
}

/**
```

- In this function, you are checking the old allowance value, and also comparing it with 0 value, which is a good thing.

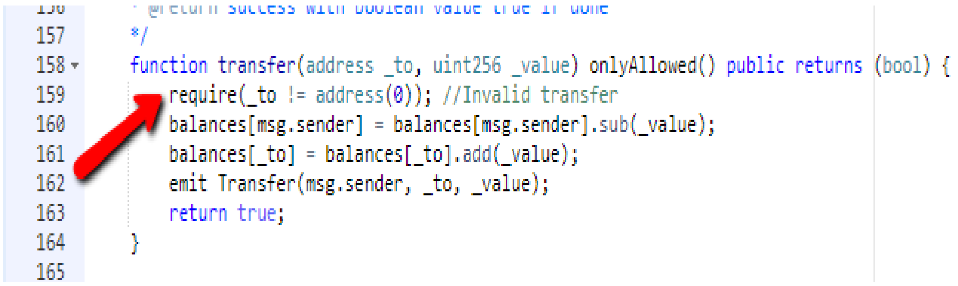
4.1.3 **transferFrom function:-**



```
function transferFrom(address _from, address _to, uint256 _value) onlyAllowed() public returns (bool) {
    require(_to != address(0)); //Invalid transfer
    balances[_from] = balances[_from].sub(_value);
    allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
    balances[_to] = balances[_to].add(_value);
    emit Transfer(_from, _to, _value);
    return true;
}
```

- In this function you are checking address of variable “to” which is a good thing.

4.1.4 **transfer Function:-**

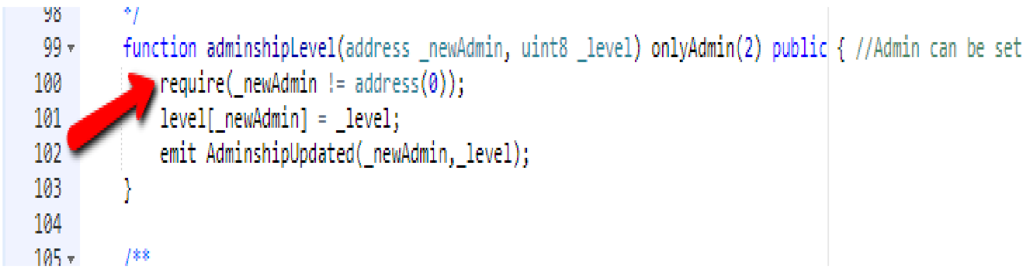


```
 * @return success with boolean value true if done
 */
function transfer(address _to, uint256 _value) onlyAllowed() public returns (bool) {
    require(_to != address(0)); //Invalid transfer
    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(_value);
    emit Transfer(msg.sender, _to, _value);
    return true;
}
```

- In this function you are checking address of variable “to” which is a good thing.

4.2 CircaICO.sol

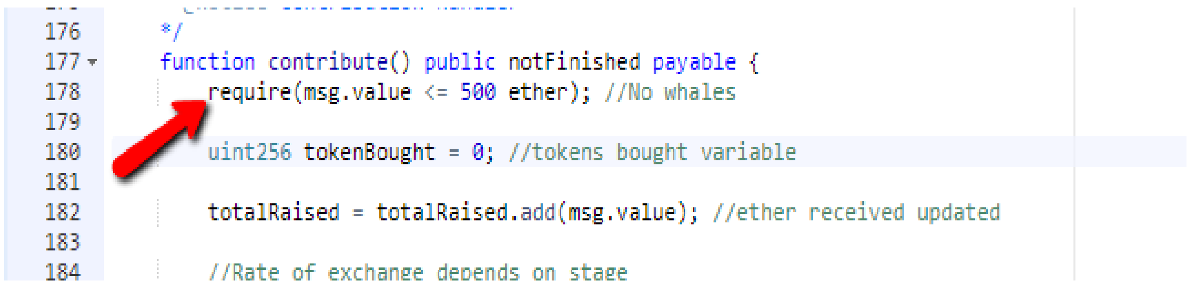
4.2.1 transferOwnership Function:-



```
*/
function adminshipLevel(address _newAdmin, uint8 _level) onlyAdmin(2) public { //Admin can be set
    require(_newAdmin != address(0));
    level[_newAdmin] = _level;
    emit AdminshipUpdated(_newAdmin,_level);
}

/**
```

- Here you are checking that the value of “_newOwner” is valid address or not, which a good thing is.

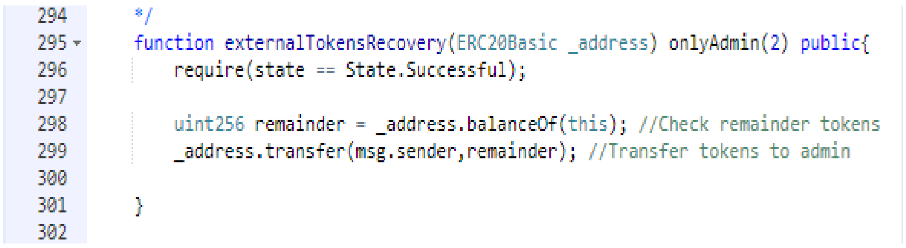
4.2.2 contribute function:-



```
*/
function contribute() public notFinished payable {
    require(msg.value <= 500 ether); //No whales

    uint256 tokenBought = 0; //tokens bought variable

    totalRaised = totalRaised.add(msg.value); //ether received updated

    //Rate of exchange depends on stage
```

- In this function you give limitation to msg.value. it is good things.

4.2.3 externalTokensRecovery function:-



```
*/
function externalTokensRecovery(ERC20Basic _address) onlyAdmin(2) public{
    require(state == State.Successful);

    uint256 remainder = _address.balanceOf(this); //Check remainder tokens
    _address.transfer(msg.sender,remainder); //Transfer tokens to admin

}
```

- This is good things that your contract also accept external token.

5. Critical vulnerabilities found in the contract

=> No critical vulnerabilities found

6. Medium vulnerabilities found in the contract

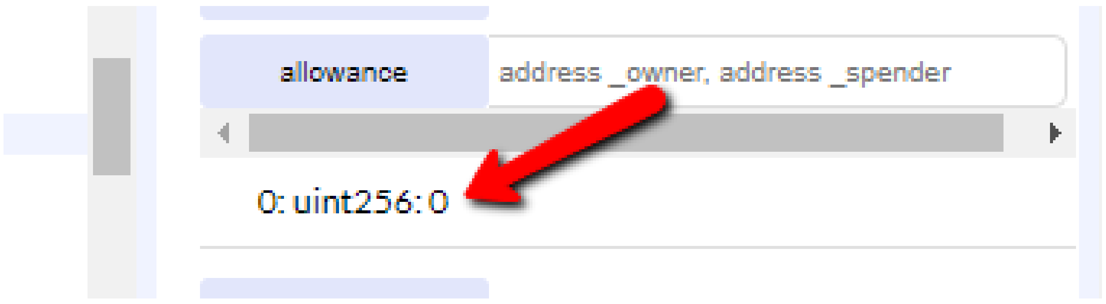
6.1 Circa.sol

6.1.1: Underflow & Overflow attack:

=>In your contract, some functions accept negative value.

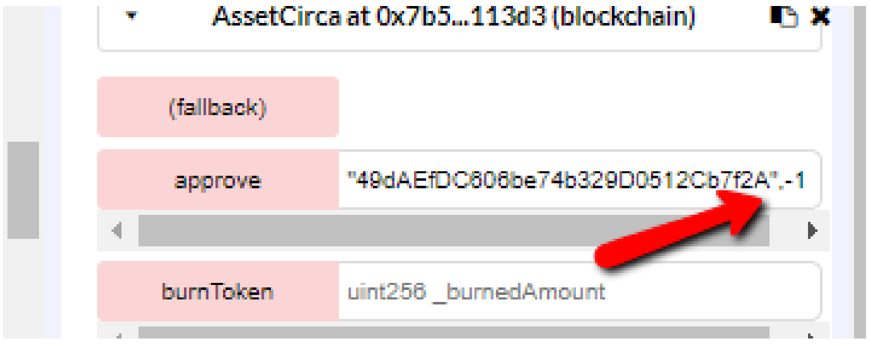
=>Function name: - approve,

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



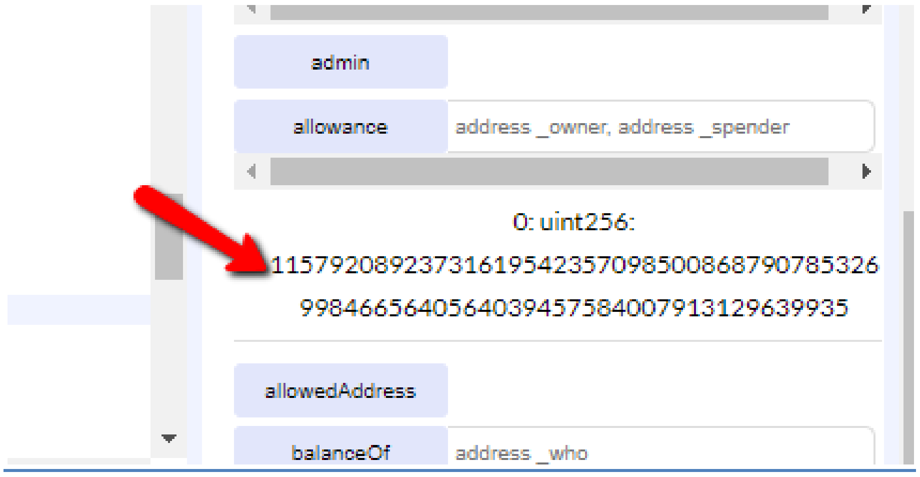
```
0: uint256: 0
```

  - Now calling approve function with negative value.



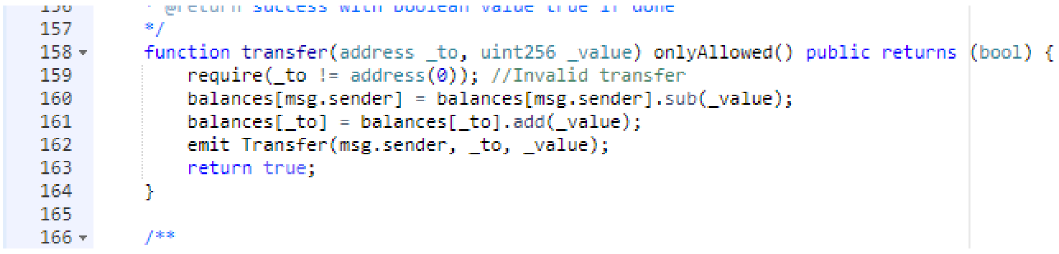
"49dAEfDC606be74b329D0512Cb7f2A",-1

- Transaction Hash:- [https://rinkeby.etherscan.io/tx/0x2942ec99dec62a9b15dc1e21ef762ad44446713b3097e0c209d7e58802067589.](https://rinkeby.etherscan.io/tx/0x2942ec99dec62a9b15dc1e21ef762ad44446713b3097e0c209d7e58802067589)
- Allowance after negative approves.



```
0: uint256:
115792089237316195423570985008687907853269984665640564039457584007913129639935
```

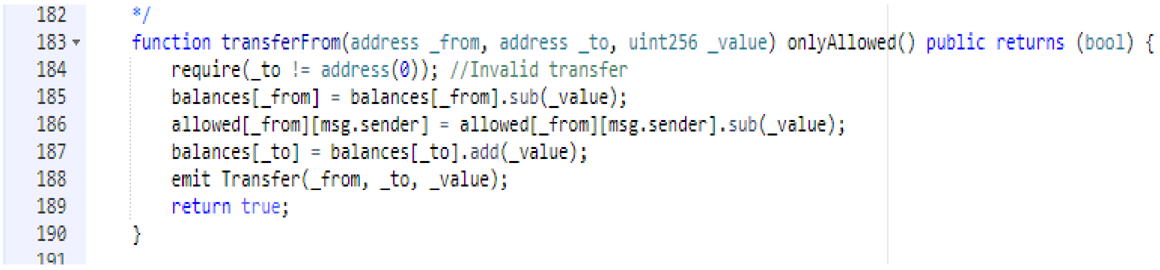
- **Transfer function**



```
     * @return success with boolean value true if done
     */
    function transfer(address _to, uint256 _value) onlyAllowed() public returns (bool) {
        require(_to != address(0)); //Invalid transfer
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(_value);
        emit Transfer(msg.sender, _to, _value);
        return true;
    }

    /**
```

- It is a good practice to put condition that user can spend only that amount that which they have.

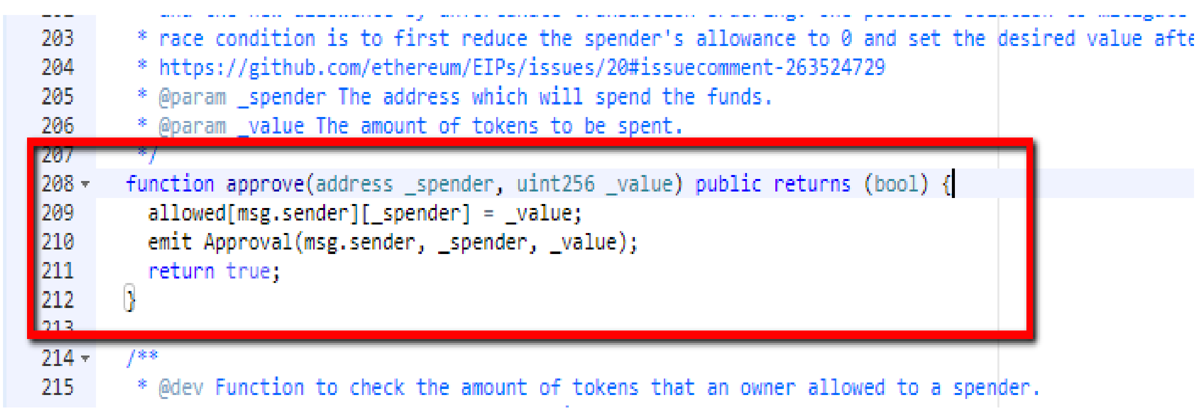
- **Transferform Function:-**



```
    */
    function transferFrom(address _from, address _to, uint256 _value) onlyAllowed() public returns (bool) {
        require(_to != address(0)); //Invalid transfer
        balances[_from] = balances[_from].sub(_value);
        allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
        balances[_to] = balances[_to].add(_value);
        emit Transfer(_from, _to, _value);
        return true;
    }
```

  - It is a good practice to put condition that user can spend only that amount that which they have.

Solution:-



```
     * race condition is to first reduce the spender's allowance to 0 and set the desired value afte
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     * @param _spender The address which will spend the funds.
     * @param _value The amount of tokens to be spent.
     */
    function approve(address _spender, uint256 _value) public returns (bool) {
      allowed[msg.sender][_spender] = _value;
      emit Approval(msg.sender, _spender, _value);
      return true;
    }

    /**
     * @dev Function to check the amount of tokens that an owner allowed to a spender.
```

- In approve, transfer and transferform functions you have to put one condition.

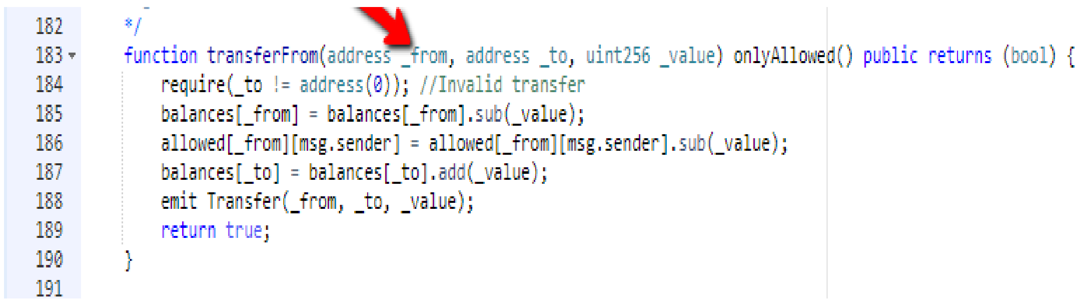
  require(_value <= balances[msg.sender]);

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

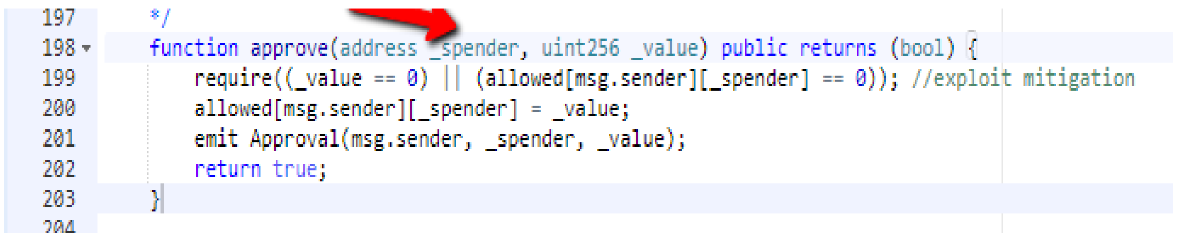
6.1.2: Short address attack

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

=>Function name: - transferFrom("From"), approve("_spender").



```
    */
    function transferFrom(address _from, address _to, uint256 _value) onlyAllowed() public returns (bool) {
        require(_to != address(0)); //Invalid transfer
        balances[_from] = balances[_from].sub(_value);
        allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
        balances[_to] = balances[_to].add(_value);
        emit Transfer(_from, _to, _value);
        return true;
    }
```



```
    */
    function approve(address _spender, uint256 _value) public returns (bool) {
        require((_value == 0) || (allowed[msg.sender][_spender] == 0)); //exploit mitigation
        allowed[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender, _value);
        return true;
    }
```

- You are not checking the value of “from” variable in transferform and “_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));**

6.2 CircaICO.sol

=> No medium vulnerabilities found

7. Low severity vulnerabilities found

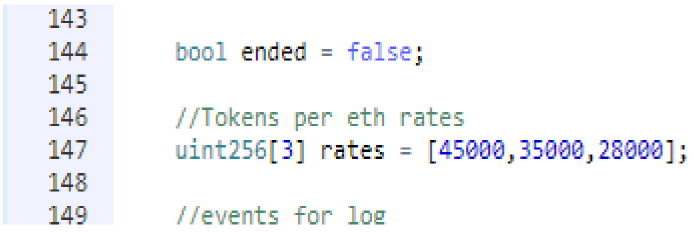
7.1 Circa.sol

=> No low vulnerabilities found

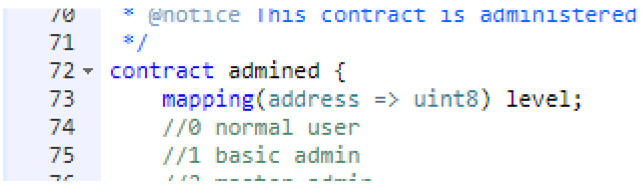
7.2 CircaICO.sol

7.2.1 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.



```
143
144     bool ended = false;
145
146     //Tokens per eth rates
147     uint256[3] rates = [45000,35000,28000];
148
149     //events for log
```



```
70    * @notice This contract is administered
71    */
72  contract admined {
73      mapping(address => uint8) level;
74      //0 normal user
75      //1 basic admin
```

Solution:-

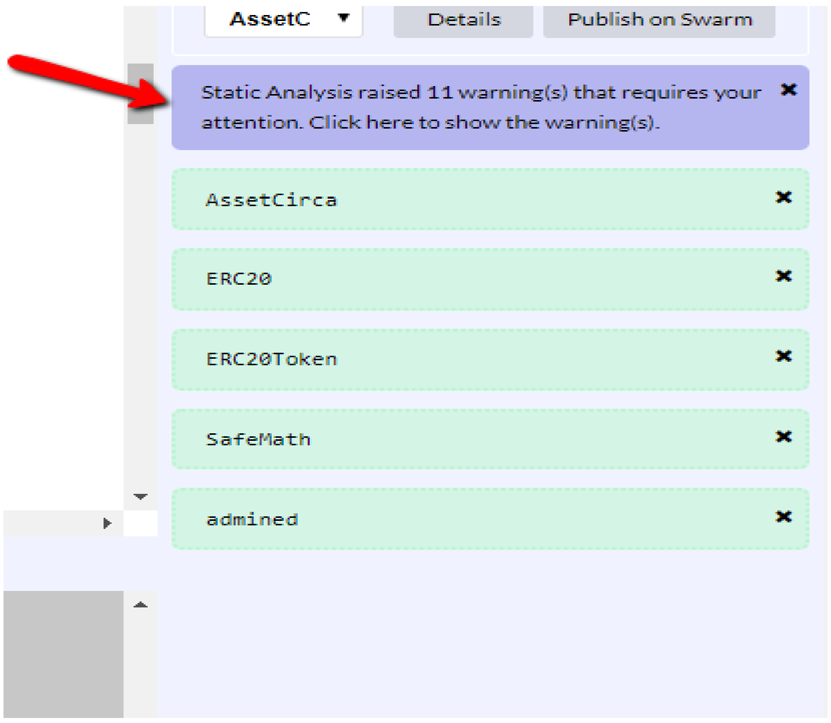
1) For #73.
    a. mapping(address => uint8) public level;
2) For #143.
    a. bool public ended = false;
3) For #147,
    a. uint256[3] public rates = [45000,35000,28000];

8. Summary of the Audit

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

8.1 Circa.sol

The compiler also displayed 11 warnings:



Static Analysis raised 11 warning(s) that requires your attention. Click here to show the warning(s).

AssetCirca

ERC20

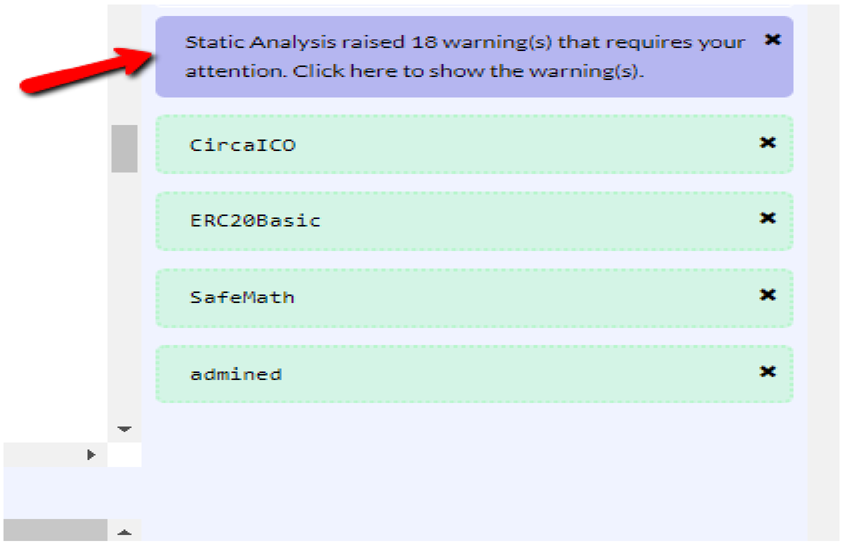
ERC20Token

SafeMath

admined

8.2 CircaICO.sol

The compiler also displayed 18 warnings:



Static Analysis raised 18 warning(s) that requires your attention. Click here to show the warning(s).

CircaICO

ERC20Basic

SafeMath

admined

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.

Main Pointers:

- All works, no problems found in any functions

- There is no existence of code which can be used to steal fund or to do any harmful activities

- No bugs. The code was deployed to the testnet without any errors.

- ETH is accessible during the sale.

- Sale dates are set, but phases can continue if cap is not reached (next stage can be forced if needed). This only can be done by the contract owner.

- 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 ;)).