SMART CONTRACT AUDIT REPORT

For

Smart Doge

Prepared By: Kishan Patel

Prepared For: Smart Doge

Prepared on: 02/08/2021

Table of Content

• 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.

• Overview of the audit

The project has 1 file. It contains approx 236 lines of Solidity code. All the functions and state variables are well commented using the natspec documentation, but that does not create any vulnerability.

• 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.

• 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, but all the functions have strong validations, which prevented this attack.

• Short address attack

If the token contract has enough amounts of tokens and the buy function doesn’t check the length of the address of the sender, the Eth’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.

• Visibility & Delegate call

It is also known as, The Parity Hack, which occurs while misuse of Delegate call.

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.

• Reentrancy / TheDAO hack

Reentrancy occurs in this case: any interaction from a contract (A) with another contract (B) and any transfer of Eth 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.

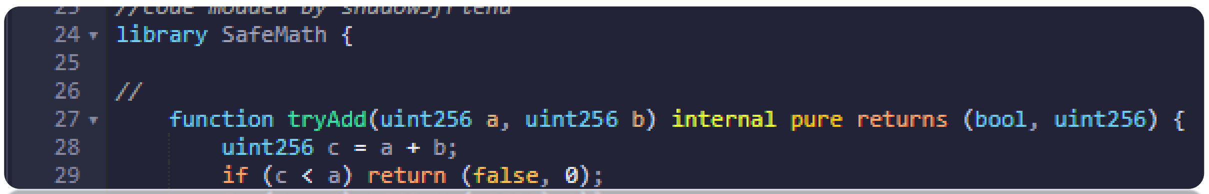
- **Forcing Ethereum to a contract**

While implementing “selfdestruct” in smart contract, it sends all the Eth 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.

• Good things in smart contract

• SafeMath library:-

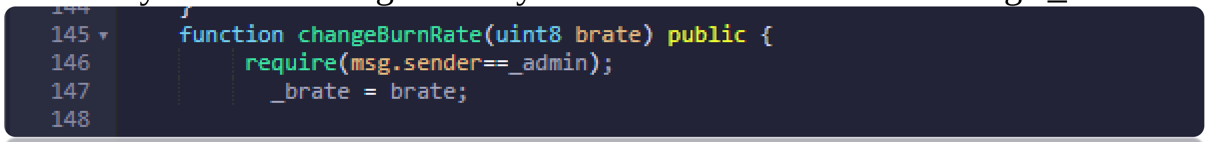
- You are using SafeMath library it is a good thing. This protects you from underflow and overflow attacks.



```
library SafeMath {

//
    function tryAdd(uint256 a, uint256 b) internal pure returns (bool, uint256) {
        uint256 c = a + b;
        if (c < a) return (false, 0);
```

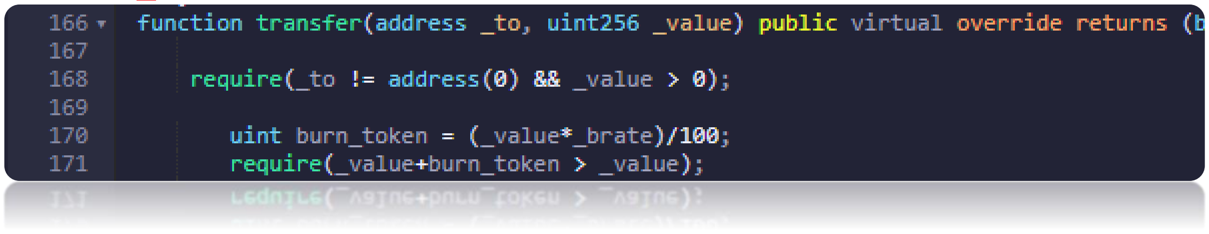
• Good required condition in functions:-

- Here you are checking that only owner of the contract can change _brate value.



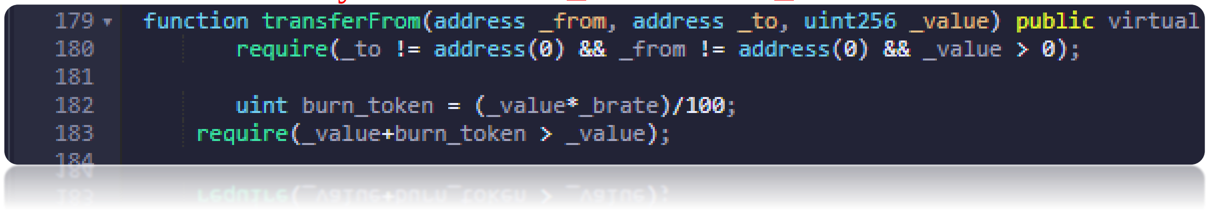
```
    function changeBurnRate(uint8 brate) public {
        require(msg.sender==_admin);
          _brate = brate;

```

- Here you are checking that _to address is value and proper, _value should be bigger than 0, _value should be smaller than burn_token + _value, and msg.sender has more balance than _value + burn_token. Here you are subtracting burn_token from _value while adding to _to address and while subtracting from msg.sender. So please check this logic. I think burn_token should be burned from msg.sender and there you have to do _value + burn_token.



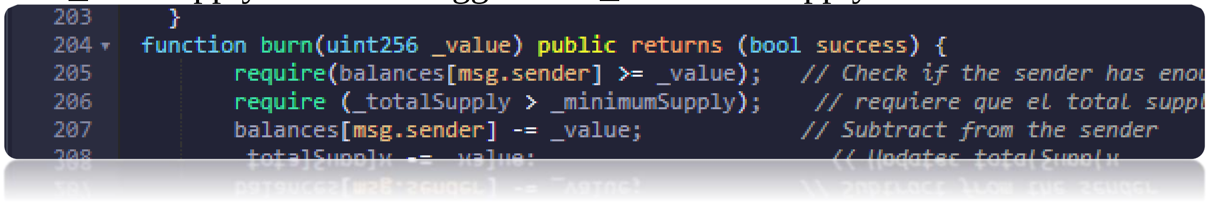
```
function transfer(address _to, uint256 _value) public virtual override returns (b

    require(_to != address(0) && _value > 0);

        uint burn_token = (_value*_brate)/100;
        require(_value+burn_token > _value);
```

- Here you are checking that _to and _from addresses value are proper and valid, _value should be bigger than 0, _value+burn_token should bigger than _value, _value + burn_token should be smaller than _from balance, and msg.sender has allowance from _from address. Here you are subtracting burn_token from _value while adding to _to address and while subtracting from _from address. So please check this logic. I think burn_token should be burned from _from address and there you have to do _value + burn_token.



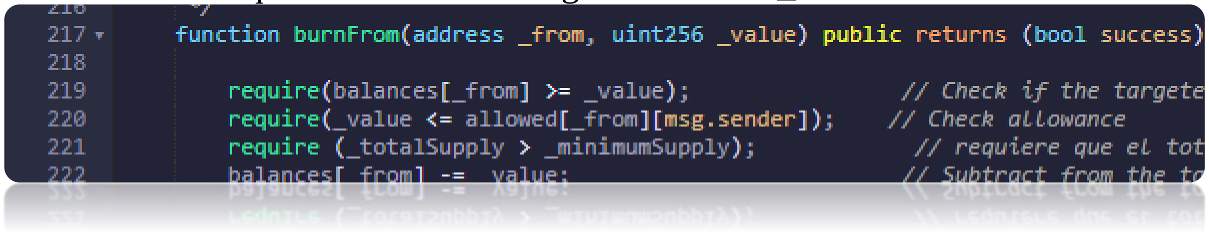
```
function transferFrom(address _from, address _to, uint256 _value) public virtual
        require(_to != address(0) && _from != address(0) && _value > 0);

        uint burn_token = (_value*_brate)/100;
     require(_value+burn_token > _value);
```

- Here you are checking that msg.sender has more or equal balance than _value, _totalSupply should be bigger than _minimumSupply.



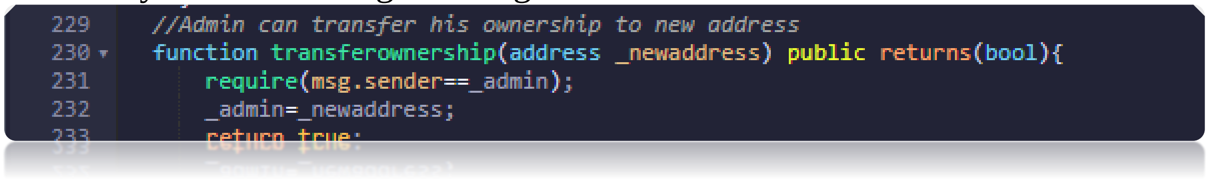
```
    }
function burn(uint256 _value) public returns (bool success) {
        require(balances[msg.sender] >= _value);   // Check if the sender has enou
        require (_totalSupply > _minimumSupply);    // requiere que el total suppl
        balances[msg.sender] -= _value;            // Subtract from the sender
```

- Here you are checking that _from has more or equal balance than _value, _totalSupply should be bigger than _minimumSupply, and _value should be smaller or equal allowance to msg.sender from _from.



```
    function burnFrom(address _from, uint256 _value) public returns (bool success)

        require(balances[_from] >= _value);                // Check if the targete
        require(_value <= allowed[_from][msg.sender]);    // Check allowance
        require (_totalSupply > _minimumSupply);           // requiere que el tot
        balances[_from] -= _value;                        // Subtract from the to
```

- Here you are checking that msg.sender is current owner of the contract.



```
//Admin can transfer his ownership to new address
function transferownership(address _newaddress) public returns(bool){
    require(msg.sender==_admin);
    _admin=_newaddress;
    return true;
```

- **Critical vulnerabilities found in the contract**

=> No Critial vulnerabilities found

- **Medium vulnerabilities found in the contract**

=> No Medium vulnerabilities found

- **Low severity vulnerabilities found**

  - **7.1: Compiler version is not fixed:-**

=> In this file you have put “pragma solidity ^0.8.0;” which is not a good way to define compiler version.

=> Solidity source files indicate the versions of the compiler they can be compiled with. Pragma solidity >=0.8.0; // bad: compiles 0.8.0 and above pragma solidity 0.8.0; //good: compiles 0.8.0 only

=> If you put(>=) symbol then you are able to get compiler version 0.8.0 and above. But if you don’t use(^/>=) symbol then you are able to use only 0.8.0 version. And if there are some changes come in the compiler and you use the old version then some issues may come at deploy time.

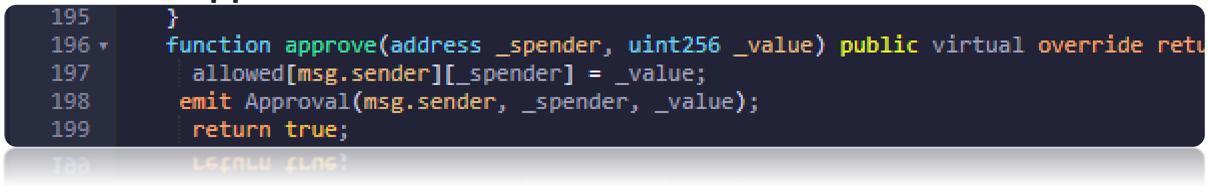
7.2: Check user balance in approve:-

=> I have found that approve function user can give more amounts value than their balance.

=> It is necessary to check that user can give less or equal value to their amount.

=> There is no validation about user balance. So it is good to check that a user not set approval wrongly.

Function: - approve



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

- Here you can check that balance of _spender address should be bigger or equal to _value.

- Here you can check that _spender address is valid and proper.

• Summary of the Audit

Overall the code is well and performs well. The smart contract

has no backdoor in the code (But for security I suggest you to

implement my suggestions).

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 ;)).

- **Good Point:** Code performance is good. Address validation and value validation is done properly.
- **Suggestions:** Please try to use the latest version of solidity, check user balance in approve function.