SMART CONTRACT AUDIT REPORT

For

TREES Token

About the Auditor: Kishan Patel works as an independent auditor with combine 5 years experience in Ethereum and Binance blockchain at Fiverr. He specialized in auditing solidity code, finding bugs and vulnerability. He has audited more than 500 smart contracts including SAFEMOON, SuperTron, MintableBEP20 to name a few.

SAFETREES

Prepared By: Kishan Patel

Prepared For: SAFETREES Project

Prepared on: 30th May 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 1182 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 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.

• 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 ethereum hands over control to that contract (B).

This makes it possible for B to call back into A before this interaction is completed.

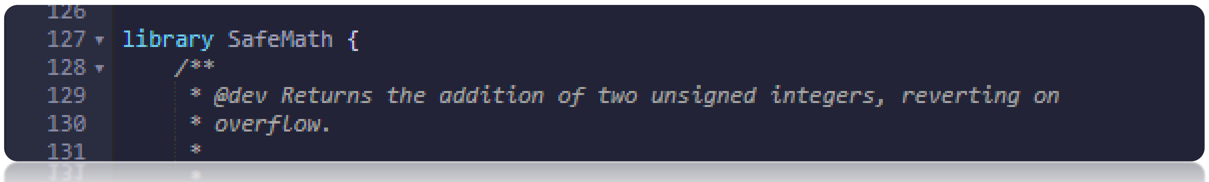
Use of “require” function in this smart contract mitigated this vulnerability.

- **Forcing Ethereum to a contract**

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

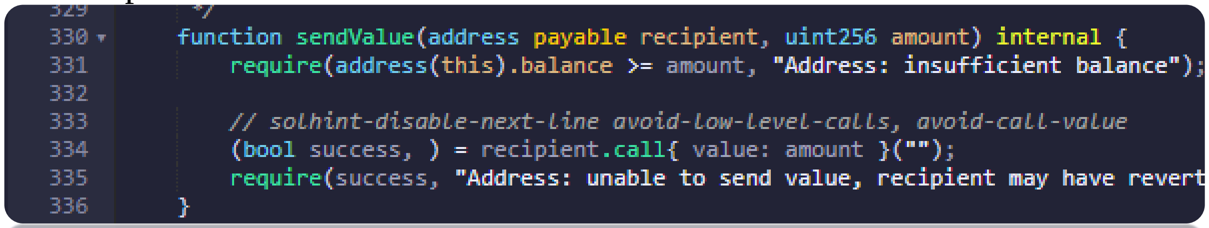
Good things in smart contract

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



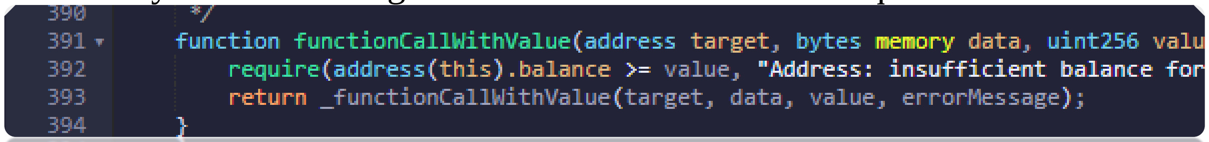
```
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
```

- **Good required condition in functions:-**
    - Here you are checking that balance of the contract is bigger or equal to the amount value and checking that token is successfully transferred to the recipient's address.



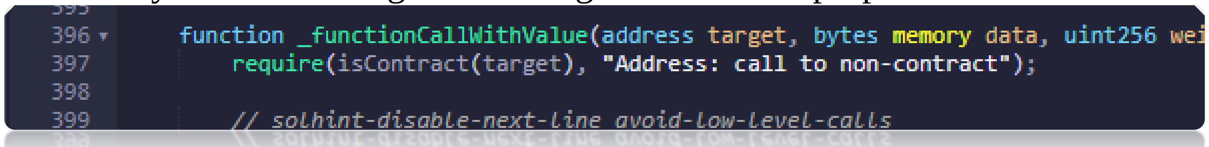
```
     */
    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");

        // solhint-disable-next-line avoid-low-level-calls, avoid-call-value
        (bool success, ) = recipient.call{ value: amount }("");
        require(success, "Address: unable to send value, recipient may have revert
    }
```

- Here you are checking that the contract has more or equal balance then value.



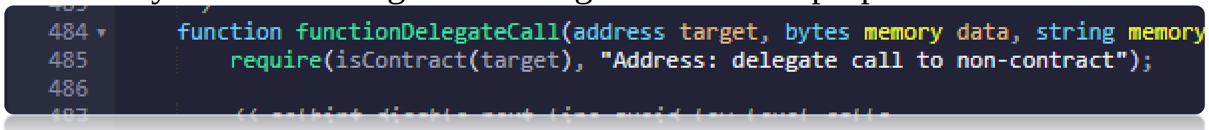
```
     */
    function functionCallWithValue(address target, bytes memory data, uint256 valu
        require(address(this).balance >= value, "Address: insufficient balance for
        return _functionCallWithValue(target, data, value, errorMessage);
    }
```

- Here you are checking that the target address is a proper contract address or not.



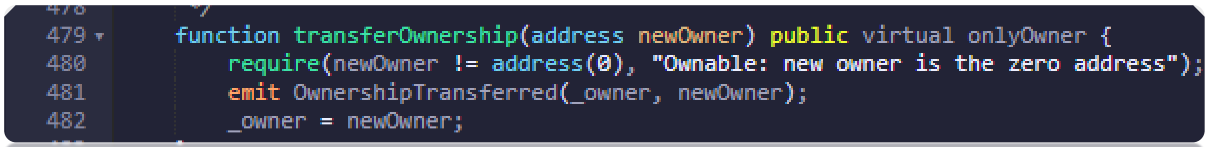
```
    function _functionCallWithValue(address target, bytes memory data, uint256 wei
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
```

- Here you are checking that the target address is a proper contract address or not.



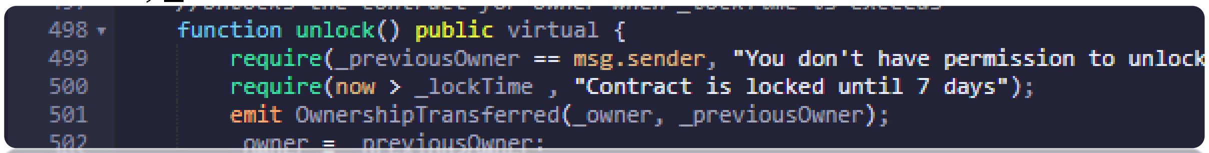
```
    function functionDelegateCall(address target, bytes memory data, string memory
        require(isContract(target), "Address: delegate call to non-contract");

```

- Here you are checking that the newOwner address value is a proper valid address.



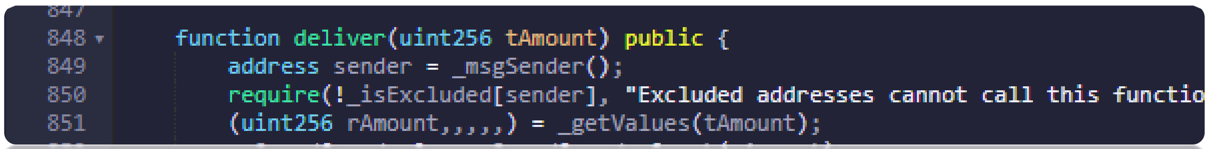
```
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
```

- Here you are checking that msg.sender should not be _previousOwner address value, _lockTime should be less than now.



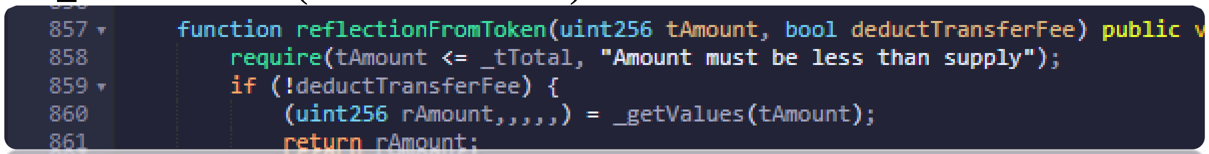
```
    function unlock() public virtual {
        require(_previousOwner == msg.sender, "You don't have permission to unlock
        require(now > _lockTime , "Contract is locked until 7 days");
        emit OwnershipTransferred(_owner, _previousOwner);
        _owner = _previousOwner;
```

- Here you are checking that this function is not called by the address which is excluded.



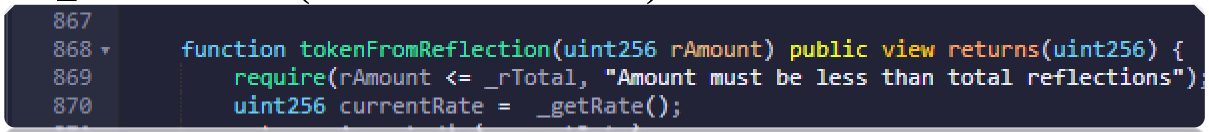
```
    function deliver(uint256 tAmount) public {
        address sender = _msgSender();
        require(!_isExcluded[sender], "Excluded addresses cannot call this functio
        (uint256 rAmount,,,,,) = _getValues(tAmount);
```

- Here you are checking that tAmount value should be less than or equal to the _tTotal amount (Total token value).



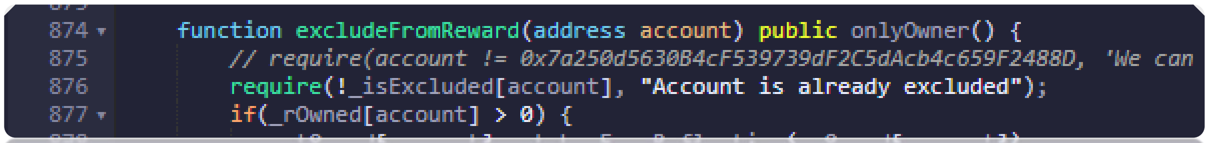
```
    function reflectionFromToken(uint256 tAmount, bool deductTransferFee) public v
        require(tAmount <= _tTotal, "Amount must be less than supply");
        if (!deductTransferFee) {
            (uint256 rAmount,,,,,) = _getValues(tAmount);
            return rAmount;
```

- Here you are checking that rAmount value should be less than or equal to the _rTotal amount (Total reflections value).



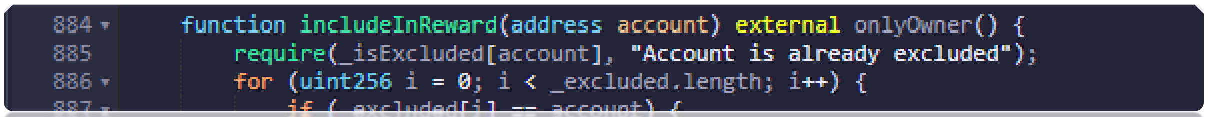
```

    function tokenFromReflection(uint256 rAmount) public view returns(uint256) {
        require(rAmount <= _rTotal, "Amount must be less than total reflections");
        uint256 currentRate =  _getRate();
```

- Here you are checking that account address is not already excluded from a reward.



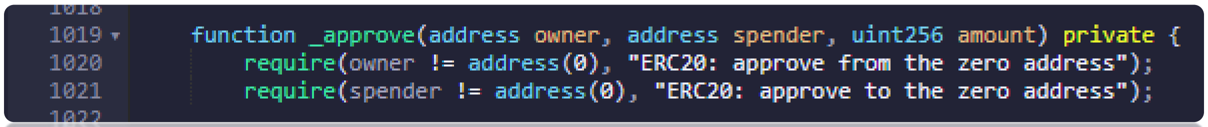
```
function excludeFromReward(address account) public onlyOwner() {
    // require(account != 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D, 'We can
    require(!_isExcluded[account], "Account is already excluded");
    if(_rOwned[account] > 0) {
```

- Here you are checking that an account address is not already included for reward.



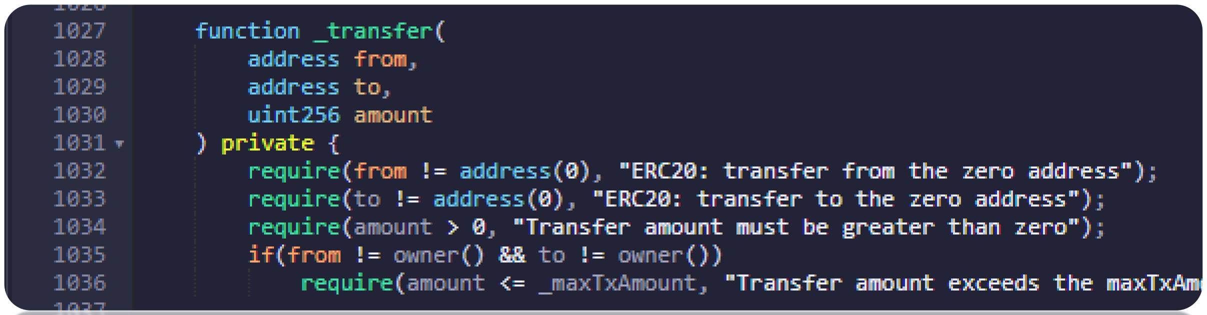
```
function includeInReward(address account) external onlyOwner() {
    require(_isExcluded[account], "Account is already excluded");
    for (uint256 i = 0; i < _excluded.length; i++) {
        if (_excluded[i] == account) {
```

- Here you are checking that owner and spender addresses value are proper addresses.



```
function _approve(address owner, address spender, uint256 amount) private {
    require(owner != address(0), "ERC20: approve from the zero address");
    require(spender != address(0), "ERC20: approve to the zero address");
```

- Here you are checking that addresses values of from and to are proper, an amount should be bigger than 0 and less than _maxTxAmount (Maximum amount to transfer token.



```
function _transfer(
    address from,
    address to,
    uint256 amount
) private {
    require(from != address(0), "ERC20: transfer from the zero address");
    require(to != address(0), "ERC20: transfer to the zero address");
    require(amount > 0, "Transfer amount must be greater than zero");
    if(from != owner() && to != owner())
        require(amount <= _maxTxAmount, "Transfer amount exceeds the maxTxAm
```

• Critical vulnerabilities found in the contract

=> No Critial vulnerabilities found

• Medium vulnerabilities found in the contract

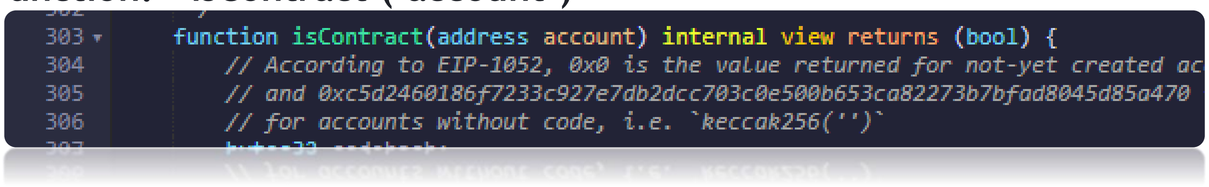
=> No Medium vulnerabilities found

- ## Low severity vulnerabilities found

  - ### 7.1: Short address attack:-

=> This is not a big issue in solidity, because of a new release of the solidity version. But it is good practice to check for the short address.
=> After updating the version of solidity it’s not mandatory.
=> In some functions you are not checking the value of Address parameter here I am showing only necessary functions.

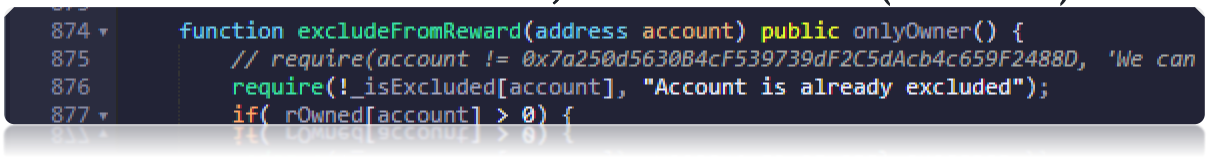
Function: - isContract (‘account’)



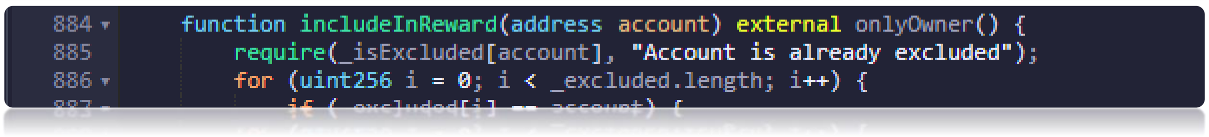
```
    function isContract(address account) internal view returns (bool) {
        // According to EIP-1052, 0x0 is the value returned for not-yet created ac
        // and 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470
        // for accounts without code, i.e. `keccak256('')`
        bytes32 codehash;
```

- It's necessary to check the address value of "account". Because here you are passing whatever variable comes in "account" address from outside.

Function: - excludeFromReward, includeInReward (‘account’)



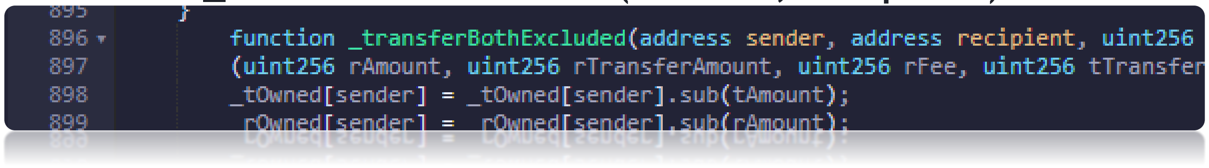
```
    function excludeFromReward(address account) public onlyOwner() {
        // require(account != 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D, 'We can
        require(!_isExcluded[account], "Account is already excluded");
        if( rOwned[account] > 0) {
```



```
    function includeInReward(address account) external onlyOwner() {
        require(_isExcluded[account], "Account is already excluded");
        for (uint256 i = 0; i < _excluded.length; i++) {
            if (_excluded[i] == account) {
```

- It's necessary to check the address value of "account". Because here you are passing whatever variable comes in "account" address from outside.

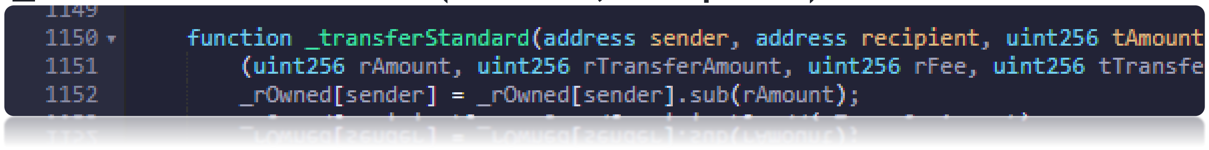
Function: - _transferBothExcluded (‘sender’, ‘recipient’)



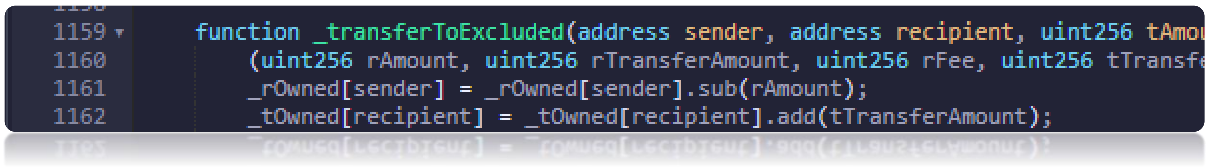
```
    }
        function _transferBothExcluded(address sender, address recipient, uint256
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransfer
        _tOwned[sender] = _tOwned[sender].sub(tAmount);
        _rOwned[sender] = _rOwned[sender].sub(rAmount);
```

- It's necessary to check the addresses value of "sender", "recipient". Because here you are passing whatever variable comes in "sender", "recipient" addresses from outside.

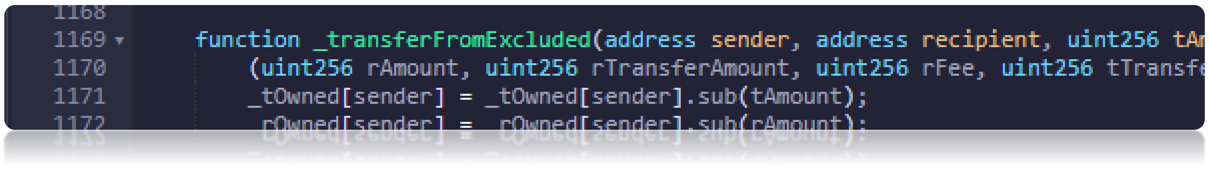
Function: - _transferStandard, _transferToExcluded, _transferFromExcluded (‘sender’, ‘recipient’)



```
    function _transferStandard(address sender, address recipient, uint256 tAmount
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransfe
        _rOwned[sender] = _rOwned[sender].sub(rAmount);
```



```
    function _transferToExcluded(address sender, address recipient, uint256 tAmou
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransfe
        _rOwned[sender] = _rOwned[sender].sub(rAmount);
        _tOwned[recipient] = _tOwned[recipient].add(tTransferAmount);
```



```
    function _transferFromExcluded(address sender, address recipient, uint256 tAm
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransfe
        _tOwned[sender] = _tOwned[sender].sub(tAmount);
        _rOwned[sender] = _rOwned[sender].sub(rAmount);
```

- It's necessary to check the addresses value of "sender", "recipient". Because here you are passing whatever variable comes in "sender", "recipient" addresses from outside.

○ 7.2: Compiler version is not fixed:-

=> In this file you have put “pragma solidity ^0.6.12;” 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 >=0.6.12; // bad: compiles 0.6.12 and above pragma solidity 0.6.12; //good: compiles 0.6.12 only

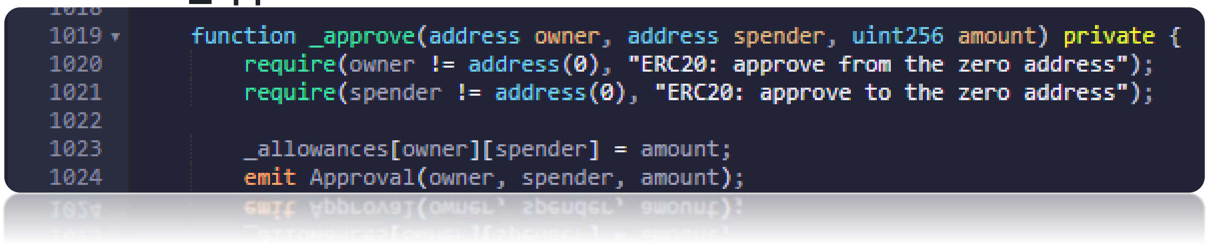
=> If you put(>=) symbol then you are able to get compiler version 0.6.12 and above. But if you don’t use(^/>=) symbol then you are able to use only 0.6.12 version. And if there are some changes come in the compiler and you use the old version then some issues may come at deploy time.

=> Use latest version of solidity.

○ 7.3: Approve given more allowance:-

=> I have found that in approve function user can give more allowance to a user beyond their balance.
=> It is necessary to check that user can give allowance less or equal 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 owner, address spender, uint256 amount) private {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
```

- Here you can check that balance of owner should be bigger or equal to amount value.

• Summary of the Audit

Overall the code is well and performs well. There is no back

door to steal fund.

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 add address validations at some place and also try to use the static version of solidity, check amount in approve function, and check burn functionality.