SMART CONTRACT AUDIT REPORT

For
SHIKOKU INU

Prepared By: Kishan Patel **Prepared For**: SHIKOKU INU

Prepared on: 05/06/2021

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 906 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.

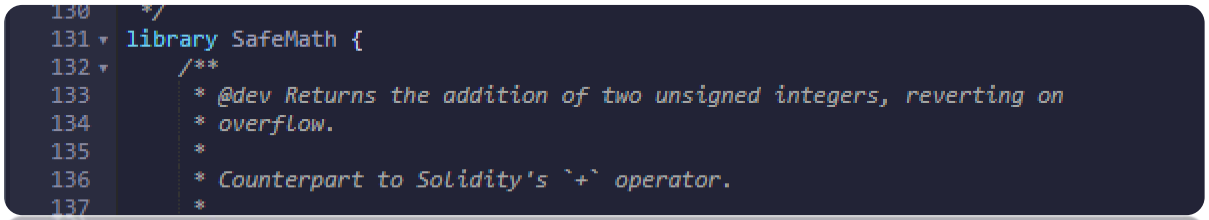
- **Forcing Ethereum to a contract**

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:-

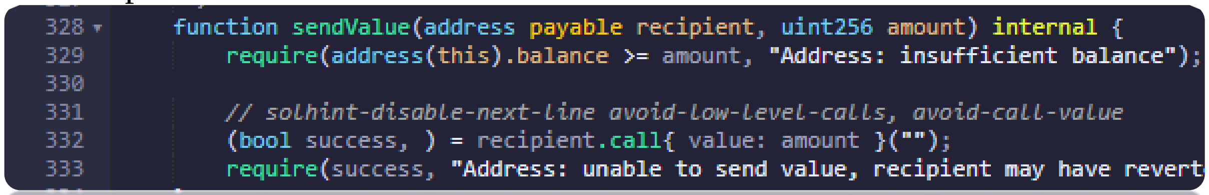
- You are using SafeMath library it is a good thing. This protects you from underflow and overflow attacks.



```
    */
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
```

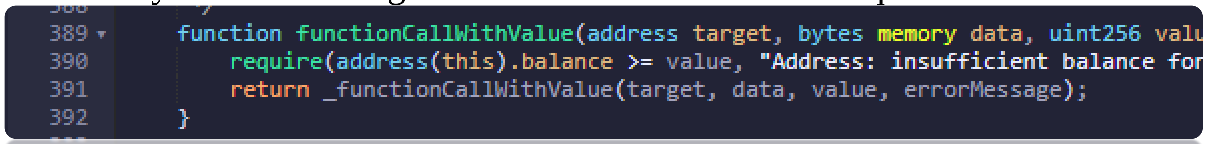
• Good required condition in functions:-

- 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.



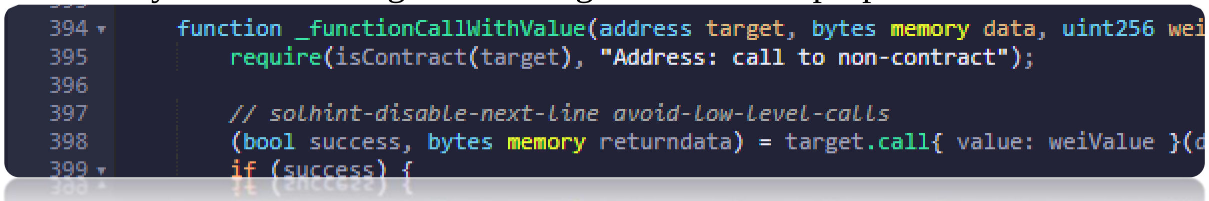
```
    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");

        // 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.



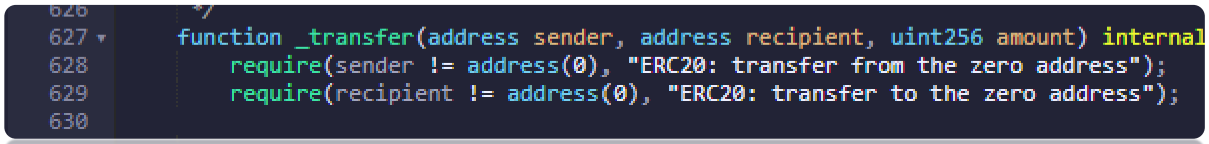
```
    function functionCallWithValue(address target, bytes memory data, uint256 valu
        require(address(this).balance >= value, "Address: insufficient balance for
        return _functionCallWithValue(target, data, value, errorMessage);
    }
```

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



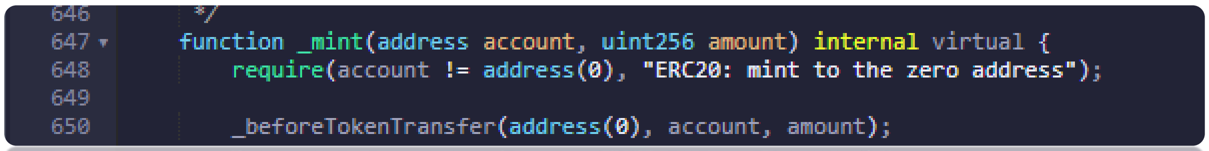
```
    function _functionCallWithValue(address target, bytes memory data, uint256 wei
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{ value: weiValue }(d
        if (success) {
```

- Here you are checking that the sender and recipient addresses value should be correct and valid.



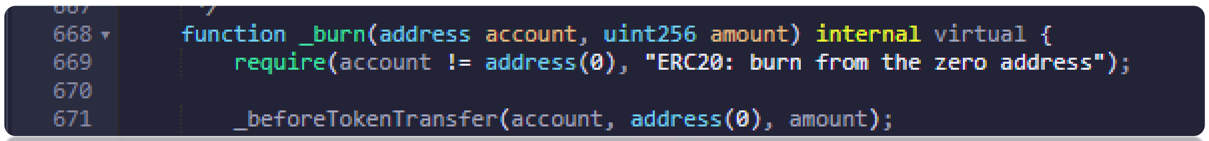
```
     */
    function _transfer(address sender, address recipient, uint256 amount) internal
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");
```

- Here you are checking that the account address value should be correct and valid.



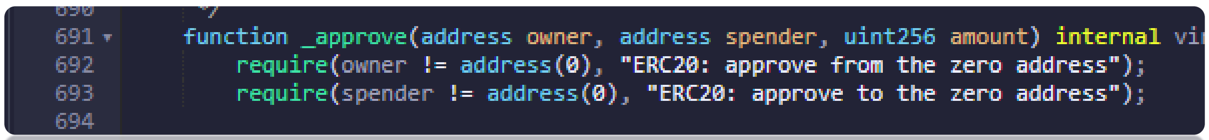
```
     */
    function _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: mint to the zero address");

        _beforeTokenTransfer(address(0), account, amount);
```

- Here you are checking that the account address value should be correct and valid.



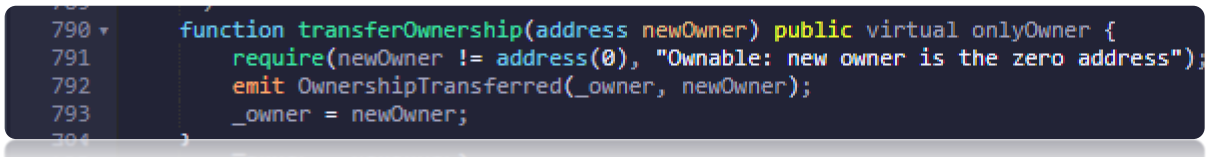
```
     */
    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero address");

        _beforeTokenTransfer(account, address(0), amount);
```

- Here you are checking that the owner and spender addresses value should be correct and valid.



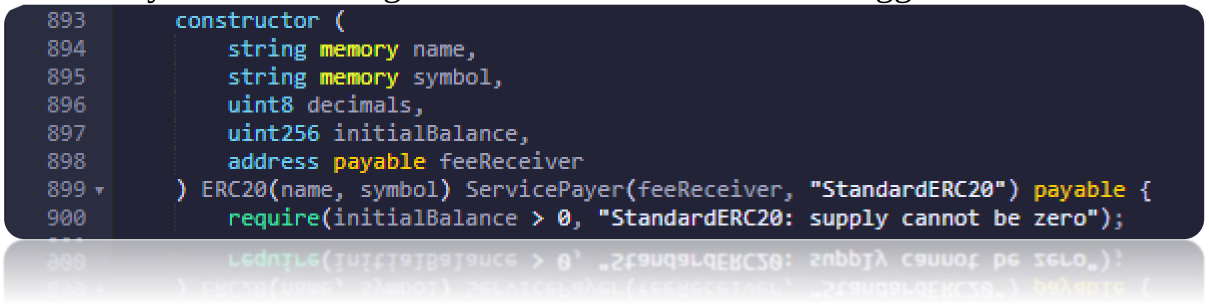
```
     */
    function _approve(address owner, address spender, uint256 amount) internal vi
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");
```

- Here you are checking that 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 initialBalance should be bigger than 0.



```
    constructor (
        string memory name,
        string memory symbol,
        uint8 decimals,
        uint256 initialBalance,
        address payable feeReceiver
    ) ERC20(name, symbol) ServicePayer(feeReceiver, "StandardERC20") payable {
        require(initialBalance > 0, "StandardERC20: supply cannot be zero");
```

- **Critical vulnerabilities found in the contract**

=> No Critial vulnerabilities found

- **Medium vulnerabilities found in the contract**

=> No Medium vulnerabilities found

- **Low severity vulnerabilities found**

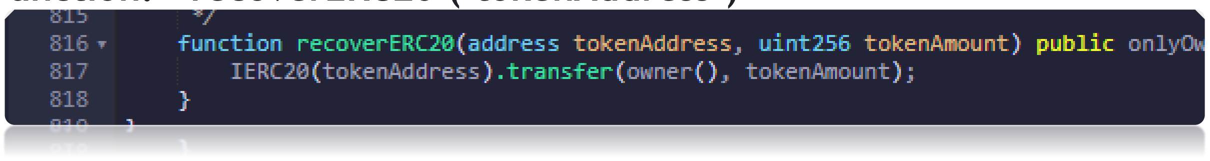
  - **7.1: Short address attack:-**

=> 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.

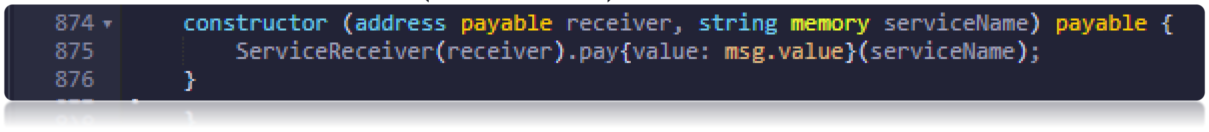
Function: - recoverERC20 ('tokenAddress')



```
    */
    function recoverERC20(address tokenAddress, uint256 tokenAmount) public onlyOw
        IERC20(tokenAddress).transfer(owner(), tokenAmount);
    }
}
```

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

Function: - constructor (‘receiver’)



```
    constructor (address payable receiver, string memory serviceName) payable {
        ServiceReceiver(receiver).pay{value: msg.value}(serviceName);
    }
```

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

7.2: Compiler version is not fixed:-

=> In this file you have put “pragma solidity ^0.7.0;” 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 >=0.7.0; // bad: compiles 0.7.0 and above pragma solidity 0.7.0; //good: compiles 0.7.0 only

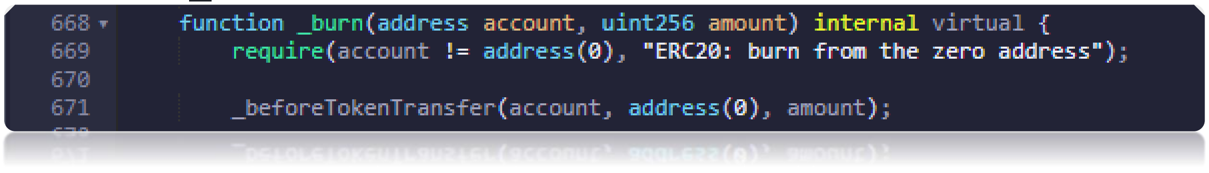
=> If you put(>=) symbol then you are able to get compiler version 0.7.0 and above. But if you don’t use(^/>=) symbol then you are able to use only 0.7.0 version. And if there are some changes come in the compiler and you use the old version then some issues may come at deploy time.

=> Use latest version of solidity.

○ 7.3: Check user balance in _burn and _approve function:-

=> I have found that in burn and approve function user can give more amount value then user has balance.
=> It is necessary to check that user can give less or equal amount value to their balance.
=> There is no validation about user balance. So it is good to check that a user not set approval wrongly.

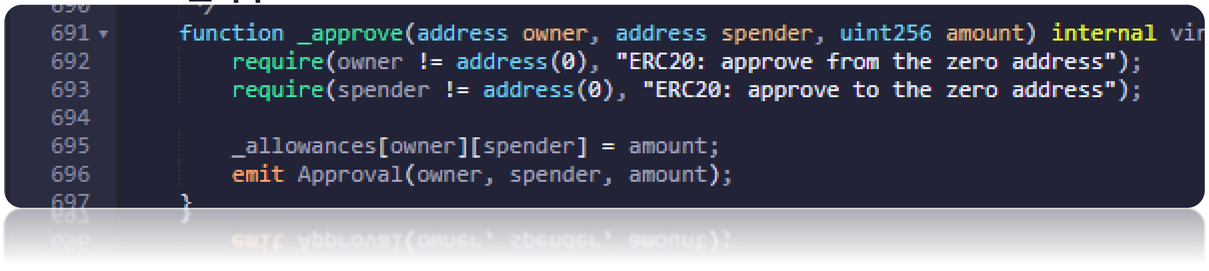
Function: - _burn



```
function _burn(address account, uint256 amount) internal virtual {
    require(account != address(0), "ERC20: burn from the zero address");

    _beforeTokenTransfer(account, address(0), amount);
```

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

Function: - _approve



```
function _approve(address owner, address spender, uint256 amount) internal vir
    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.

7.4: Uncheck return response of transfer method:-

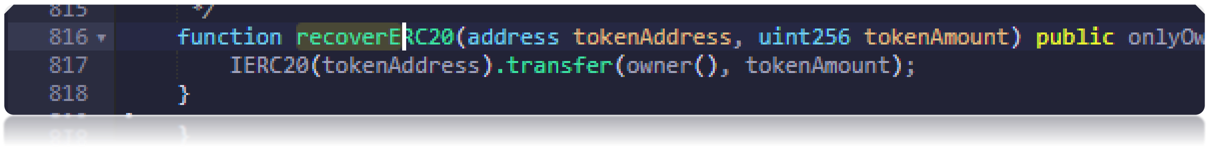
=> I have found that you are transferring fund to address using a transfer method.

=> It is always good to check the return value or response from a function call.

=> Here are some functions where you forgot to check a response.

=> I suggest, if there is a possibility then please check the response.

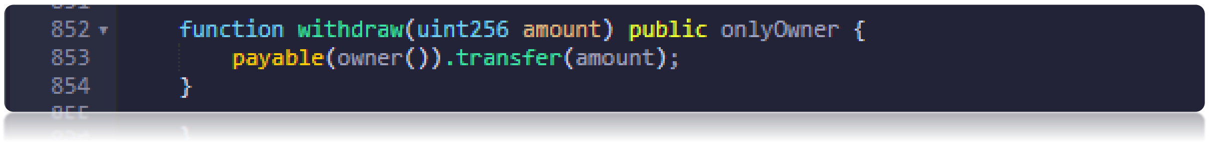
Function: - recoverERC20



```
    function recoverERC20(address tokenAddress, uint256 tokenAmount) public onlyOw
        IERC20(tokenAddress).transfer(owner(), tokenAmount);
    }
```

- Here you are calling transfer method of tokenAddress contract 1 time. It is good to check that the transfer is successfully done or not.

Function: - withdraw



```
    function withdraw(uint256 amount) public onlyOwner {
        payable(owner()).transfer(amount);
    }
```

- Here you are calling transfer method 1 time. It is good to check that the transfer is successfully done or not.

• 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 latest and static version of solidity, check user balance in burn and approve function, and check return response of transfer method.