SMART CONTRACT AUDIT REPORT V1

For

SaferICO

Prepared By: SFI Team

Prepared for: ...

Prepared on: 17/06/2021

Table of Content

• Disclaimer

The audit makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about the 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 650 lines of Solidity code. All the functions and state variables are well commented on using the Nat spec 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 minuses 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, 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, there are some points where users can mess themselves due to this (Please see below). It is highly recommended to call functions after checking the validity of the address.

- ## Visibility & Delegate call

It is also known as, The Parity Hack, which occurs while misusing 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 / The DAO 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.

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

- ## Forcing Ethereum to a contract

While implementing “self-destruct” in a 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 the target contract does not get called. And thus, Hackers can bypass the “Required” conditions. Here, the Smart Contract’s balance has never been used as a guard, which mitigated this vulnerability.

• Good things in smart contract

- ### Compiler version is static: -

  => In this file, you have put “pragma solidity 0.5.16;” which is a good way to define the compiler version.

  => Solidity source files indicate the versions of the compiler they can be compiled with. Pragma solidity ^0.5.16; // bad: compiles 0.5.16 and above pragma solidity 0.5.16; //good: compiles 0.5.16 only

```
pragma solidity 0.5.16;
```

  => If you put the (^) symbolthen you are able to get compiler version 0.5.16 and above. But if you don’t use the (^) symbol then you are able to use only 0.5.16 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 the latest version of solidity.

- ### SafeMath library: -
  - SFI are using SafeMath library it is a good thing. This protects SFI from underflow and overflow attacks.

```
library SafeMath {
  /**
   * @dev Returns the addition of two unsigned integers, reverting on
   * overflow.
   *
   * Counterpart to Solidity's `+` operator.
   *
   * Requirements:
   * - Addition cannot overflow.
   */
```

Good required condition in functions: -

- Here you are checking that newOwner address value should be a proper and valid address.

```
function _transferOwnership(address newOwner) internal {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
  }
```

- Here you are adding a lock and unlock function with the possibility of knowing the time of the lock.

```
function geUnlockTime() public view returns (uint256) {
        return _lockTime;
    }

    function lock(uint256 time) public onlyOwner {
        _previousOwner = _owner;
        _owner = address(0);
        _lockTime = time;
        emit OwnershipTransferred(_owner, address(0));
    }

    function unlock() public {
        require(_previousOwner == msg.sender, "You don't have permission to unlock");
        require(now > _lockTime , "Contract is locked until 0 days");
        emit OwnershipTransferred(_owner, _previousOwner);
        _owner = _previousOwner;
    }
```

- Here you are using functions that can be used for airdrop or giveaway or bounty distributions, the first function can be used to burn token by transferring them to the burn address, the second function can be used to transfer the token to many addresses with a different amount, the last one can be used to transfer the token to many addresses with the same amount.

```
function transfer(address recipient, uint256 amount) external returns (bool)
{
    _transfer(_msgSender(), recipient, amount);
    return true;
  }
   function multiTransfer(address[] memory recipient, uint256[] memory
amount) public returns (bool success) {
        require(recipient.length <= 200000, "Too many recipients");

        for(uint256 i = 0; i < recipient.length; i++)
        {

            _transfer(_msgSender(),recipient[i], amount[i]);
```

```
        }

        return true;
    }
  function multiTransferSingleValue(address[] memory recipient, uint256
_value) public returns (bool success) {
        uint256 toSend = _value * 10**18;

        require(recipient.length <= 200000, "Too many recipients");

        for(uint256 i = 0; i < recipient.length; i++) {
            _transfer(_msgSender(),recipient[i], toSend);
        }

        return true;
    }
```

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

```
function _transfer(address sender, address recipient, uint256 amount)
internal {
    require(sender != address(0), "BEP20: transfer from the zero address");
    require(recipient != address(0), "BEP20: transfer to the zero address");

    _balances[sender] = _balances[sender].sub(amount, "BEP20: transfer amount
exceeds balance");
    _balances[recipient] = _balances[recipient].add(amount);
    emit Transfer(sender, recipient, amount);
  }
```

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

```
function _approve(address owner, address spender, uint256 amount) internal {
    require(owner != address(0), "BEP20: approve from the zero address");
    require(spender != address(0), "BEP20: approve to the zero address");

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

- Here you are checking that account address value should be proper and valid address, burn function you can use it or use transfer function both will work

```
function _burn(address account, uint256 amount) internal {
    require(account != address(0), "BEP20: burn from the zero address");

    _balances[account] = _balances[account].sub(amount, "BEP20: burn amount
exceeds balance");
    _totalSupply = _totalSupply.sub(amount);
    emit Transfer(account, address(0), amount);
  }
```

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

```
function _mint(address account, uint256 amount) internal {
    require(account != address(0), "BEP20: mint to the zero address");

    _totalSupply = _totalSupply.add(amount);
    _balances[account] = _balances[account].add(amount);
    emit Transfer(address(0), account, amount);
  }
```

- **Critical vulnerabilities found in the contract**

=> No critical vulnerabilities found

- **Medium vulnerabilities found in the contract**

=> No Medium vulnerabilities found

- # Low severity vulnerabilities found

  - ## 7.1: 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**
      - Here you can check that amount is not more than balance of owner.

```
function _approve(address owner, address spender, uint256 amount) internal {
    require(owner != address(0), "BEP20: approve from the zero address");
    require(spender != address(0), "BEP20: approve to the zero address");

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

  - ## 7.2: Unchecked return value or response: -

    => 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: - clearBNB**

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

```
function clearBNB() public onlyOwner {
    address payable _owner = msg.sender;
    _owner.transfer(address(this).balance);
  }
  function() external payable {

  }

}
```

- # Summary of the Audit

Overall, the code is well and performs well. **There is**

no backdoor to still fund from this contract.

Please try to check the address and value of the token externally before sending it 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 execute the functions and to follow best practices regarding the use of assert, require, etc. (which you are doing ;))...

- **Note:** Please focus on a check balance of owner in approves function, check return response of transfer method call, and use the latest version of solidity.