SMART CONTRACT AUDIT REPORT

For

BallerMoon

Prepared By: Kishan P.

Prepared For: BallerMoon Foundation

Prepared on: 19/07/2021

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 787 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
- ### Compiler version is static:-

  => In this file you have put "pragma solidity 0.8.4;" which is a good way to define compiler version.

  => Solidity source files indicate the versions of the compiler they can be compiled with. Pragma solidity >=0.8.4; // bad: compiles 0.8.4 and above pragma solidity 0.8.4; //good: compiles 0.8.4 only

  => If you put(>=) symbol then you are able to get compiler version 0.8.4 and above. But if you don't use(^/>=) symbol then you are able to use only 0.8.4 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.

- ### Good required condition in functions:-
  - Here you are checking that the newOwner address value is a proper valid address.

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

  - Here you are checking that the msg.sender should not be _previouseOwner, and current time should be bigger than _lockTime.

```
73
74    function unlock() public virtual {
75        require(_previousOwner == msg.sender, "You don't have permission to unlock"
76        require(block.timestamp > _lockTime , "Contract is still locked");
77        emit OwnershipTransferred(_owner, _previousOwner);
```

- 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");
    (, uint256 tFee, uint256 tLiquidity, uint256 tCommunity, uint256 tBurn) =
    uint256 currentRate =  getRate();
```

- 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();
    return rAmount / currentRate;
```

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

```
function excludeFromReward(address account) public onlyOwner {
    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.

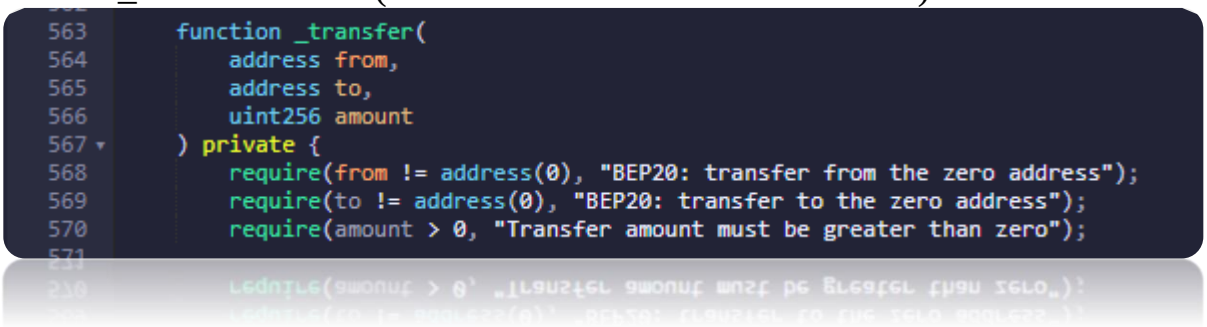
```
function includeInReward(address account) external onlyOwner {
    require(_isExcluded[account], "Account is already included");

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

- 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), "BEP20: approve from the zero address");
    require(spender != address(0), "BEP20: approve to the zero address");
```

- Here you are checking that addresses values of sender and recipient are proper, an amount should be bigger than 0, and amount value should be equal or less than _maxTxAmount (Maximum amount to transfer token).



```
563     function _transfer(
564         address from,
565         address to,
566         uint256 amount
567     ) private {
568         require(from != address(0), "BEP20: transfer from the zero address");
569         require(to != address(0), "BEP20: transfer to the zero address");
570         require(amount > 0, "Transfer amount must be greater than 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

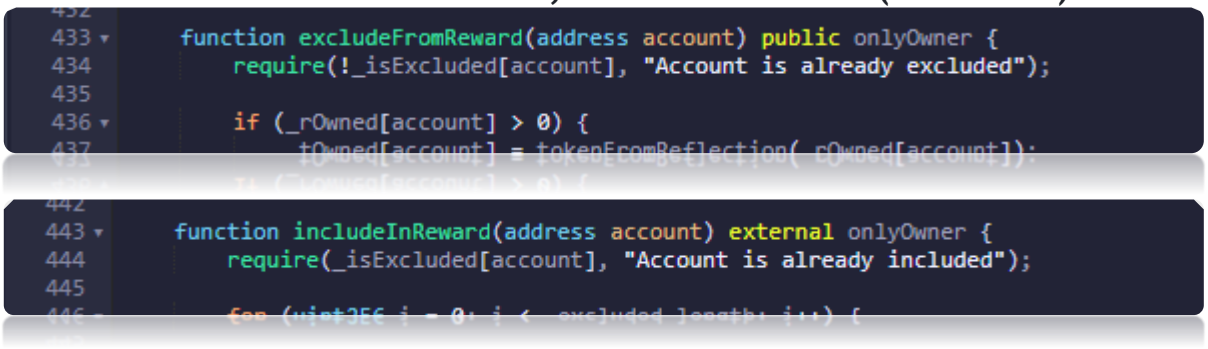
  - ## 7.1: Short address attack:-

=> 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: - excludeFromReward, includeInReward ('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: - _transferStandard, _transferBothExcluded, _transferToExcluded, _transferFromExcluded (‘sender’, ‘recipient’)**

```
709
710     function _transferStandard(address sender, address recipient, uint256 tAmount)
711         (uint256 tTransferAmount, uint256 tFee, uint256 tLiquidity, uint256 tCommu
712         uint256 currentRate = _getRate();
```

```
725     function _transferBothExcluded(address sender, address recipient, uint256 tAmo
726         (uint256 tTransferAmount, uint256 tFee, uint256 tLiquidity, uint256 tCommu
727         uint256 currentRate = _getRate();
728         (uint256 rAmount, uint256 rTransferAmount, uint256 rFee) =  getRValues(tAm
```

```
742     function _transferToExcluded(address sender, address recipient, uint256 tAmoun
743         (uint256 tTransferAmount, uint256 tFee, uint256 tLiquidity, uint256 tCommu
744         uint256 currentRate = _getRate();
745         (uint256 rAmount, uint256 rTransferAmount, uint256 rFee) =  getRValues(tAm
```

```
758     function _transferFromExcluded(address sender, address recipient, uint256 tAmo
759         (uint256 tTransferAmount, uint256 tFee, uint256 tLiquidity, uint256 tCommu
760         uint256 currentRate = _getRate();
761         (uint256 rAmount, uint256 rTransferAmount, uint256 rFee) = _getRValues(tAm
```

  - 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: 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**

```
555     function _approve(address owner, address spender, uint256 amount) private {
556         require(owner != address(0), "BEP20: approve from the zero address");
557         require(spender != address(0), "BEP20: approve to the zero address");
558
559         _allowances[owner][spender] = amount;
560         emit Approval(owner, spender, amount);
561     }
```

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

7.3: Suggestions to add validations:-

=> You have implemented required validation in contract.

=> There are some place where you can improve validation and security of your code.

=> These are all just suggestion it is not bug.

Function: - setAntiWhaleEnabled

```
456
457     function setAntiWhaleEnabled(bool e) external onlyOwner {
458         _isAntiWhaleEnabled = e;
459     }
```

- Here you can check that e value and _isAntiWahleEnabled value is not same then only you can assign it with this you can stop execution of function for same value assignment.

Function: - setAntiWhaleEnabled, setFees, setMaxTxPercent, setMinTokenBalance

```
461     function setAntiWhaleThreshold(uint256 amount) external onlyOwner {
462         _antiWhaleThreshold = amount;
463     }
```

```
473     function setFees(uint256 taxFee, uint256 liquidityFee, uint256 communityFee, u
474         _taxFee = taxFee;
475         _liquidityFee = liquidityFee;
```

```
479
480     function setMaxTxPercent(uint256 maxTxPercent) external onlyOwner {
481         _maxTxAmount = _tTotal * maxTxPercent / 100;
482     }
```

```
484     function setMinTokenBalance(uint256 minTokenBalance) external onlyOwner {
485         _minTokenBalance = minTokenBalance;
486     }
```

- Here you can check that all parameters in all methods should be bigger than 0.

Function: - setSwapandLiquifyEnabled

```
488 ▾   function setSwapAndLiquifyEnabled(bool _enabled) public onlyOwner {
489         swapAndLiquifyEnabled = _enabled;
490         emit SwapAndLiquifyEnabledUpdated(_enabled);
491     }
```

- Here you can check that _enabled value and swapAndLiquifyEnabled value is not same then only you can assign it with this you can stop execution of function for same value assignment.

Function: - setExcludedFromAntiWhale

```
465 ▾   function setExcludedFromAntiWhale(address account, bool e) external onlyOwner
466         _isExcludedFromAntiWhale[account] = e;
467     }
```

- Here you can check that account is not already excluded from _isExcludedFromAntiWhale. It is it already excluded then you don't need to execute this function.

- Not sure but might you need setIncludedInAntiWhale function which can add address value in _isExcludedFromAntiWhale.

Function: - setExcludedFromFee

```
469 ▾   function setExcludedFromFee(address account, bool e) external onlyOwner {
470         _isExcludedFromFee[account] = e;
471     }
```

- Here you can check that account is not already excluded from _isExcludedFromFee. It is it already excluded then you don't need to execute this function.

- Not sure but might you need setIncludedInFee function which can add address value in _isExcludedFromFee.

• 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:** Latest and static version of solidity is used, Code performance and quality is good. Address validation and value validation is done properly.
- **Suggestions:** Please add address validations at some place, check user balance in approve function, and try to add requested suggestions.