SMART CONTRACT AUDIT REPORT

For

Help Coins

Prepared By: Kishan Patel

Prepared For: HCS team

Prepared on: 14/05/2021

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 604 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 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

- ### Compiler version is static:-

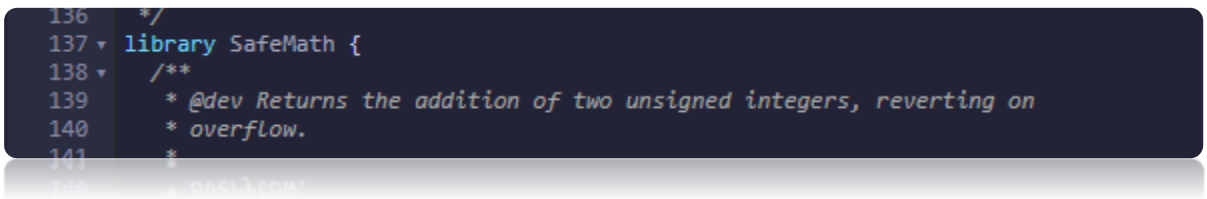
  => In this file you have put "pragma solidity 0.5.16;" which is a good way to define compiler version.

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

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

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



- **Good required condition in functions:-**
  - Here you are checking that newOwner address value should be proper and valid address.

```
337  function _transferOwnership(address newOwner) internal {
338    require(newOwner != address(0), "Ownable: new owner is the zero address");
339    emit OwnershipTransferred(_owner, newOwner);
340    _owner = newOwner;
341  }
```

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

```
521  function _transfer(address sender, address recipient, uint256 amount) internal {
522    require(sender != address(0), "BEP20: transfer from the zero address");
523    require(recipient != address(0), "BEP20: transfer to the zero address");
524
525    _balances[sender] = _balances[sender].sub(amount, "BEP20: transfer amount exce
```

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

```
539  function _mint(address account, uint256 amount) internal {
540    require(account != address(0), "BEP20: mint to the zero address");
541
542    totalSupply = totalSupply.add(amount);
```

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

```
558  function _burn(address account, uint256 amount) internal {
559    require(account != address(0), "BEP20: burn from the zero address");
560
```

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

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

- **Critical vulnerabilities found in the contract**

=> No critical vulnerabilities found

- **Medium vulnerabilities found in the contract**

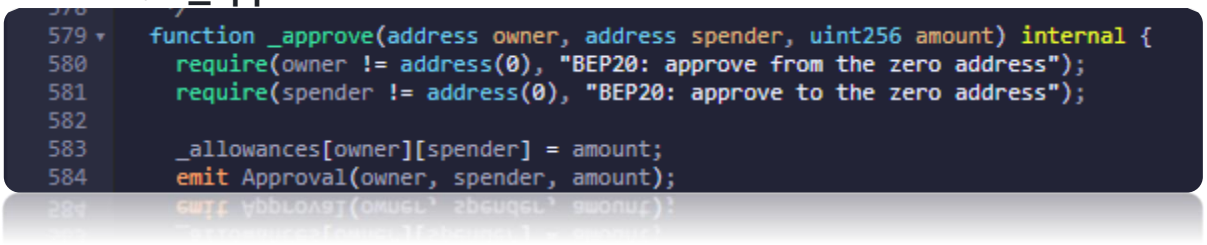
=> No Medium vulnerabilities found

- # 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**



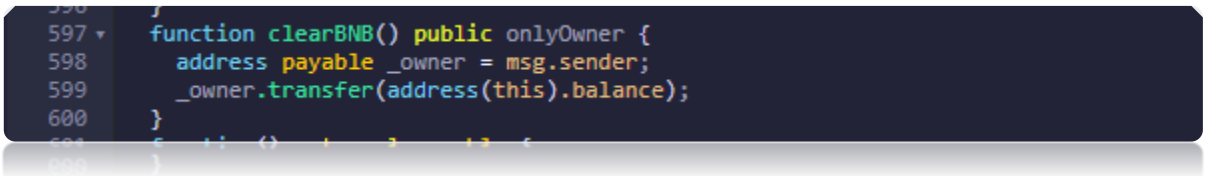
```
579 ▾ function _approve(address owner, address spender, uint256 amount) internal {
580     require(owner != address(0), "BEP20: approve from the zero address");
581     require(spender != address(0), "BEP20: approve to the zero address");
582
583     _allowances[owner][spender] = amount;
584     emit Approval(owner, spender, amount);
```

      - Here you can check that amount is not more than balance of owner.

  - ## 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**



```
597 ▾ function clearBNB() public onlyOwner {
598     address payable _owner = msg.sender;
599     _owner.transfer(address(this).balance);
600 }
```

      - 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**

backdoor to still fund from this contract.

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 ;)). .

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