BuckSwap

SMART CONTRACT AUDIT REPORT
For
BucksMaster

Audited by: Rug Doctor **Date:** 27/04/2021

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 1013 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.

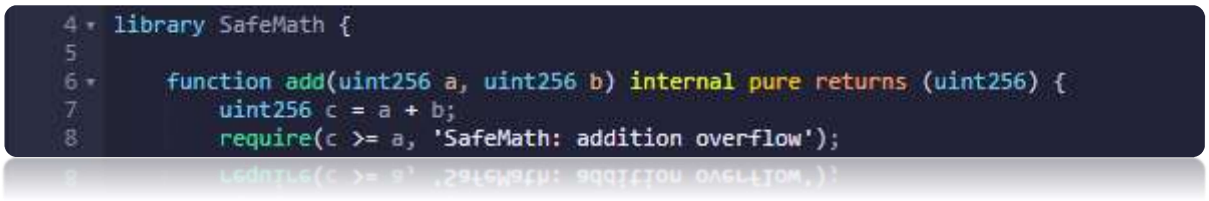
Use of "require" function in this smart contract mitigated this vulnerability.

- ## 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

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



```
4  library SafeMath {
5
6      function add(uint256 a, uint256 b) internal pure returns (uint256) {
7          uint256 c = a + b;
8          require(c >= a, 'SafeMath: addition overflow');
```

- ### Compiler version is fixed:-

=> In this file you have put "pragma solidity 0.6.12;" 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 ^0.6.12; // 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.

- ## Good required condition in functions:-
  - Here you are checking that balance this contract should be bigger or equal to amount value and method of recipient should be successfully called.

```
155    function sendValue(address payable recipient, uint256 amount) internal {
156        require(address(this).balance >= amount, 'Address: insufficient balance');
157
158
159        (bool success, ) = recipient.call{value: amount}('');
160        require(success, 'Address: unable to send value, recipient may have revert
```

  - Here you are checking that a balance of this contract is equal or bigger than value.

```
187    function functionCallWithValue(
188        address target,
189        bytes memory data,
190        uint256 value,
191        string memory errorMessage
192    ) internal returns (bytes memory) {
193        require(address(this).balance >= value, 'Address: insufficient balance for
```

  - Here you are checking that a target address is contract address.

```
197    function _functionCallWithValue(
198        address target,
199        bytes memory data,
200        uint256 weiValue,
201        string memory errorMessage
202    ) private returns (bytes memory) {
203        require(isContract(target), 'Address: call to non-contract');
```

  - Here you are checking that newOwner address is proper and valid address.

```
350    function _transferOwnership(address newOwner) internal {
351        require(newOwner != address(0), 'Ownable: new owner is the zero address');
352        emit OwnershipTransferred(_owner, newOwner);
353        _owner = newOwner;
354    }
```

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

```
463    function _transfer(
464        address sender,
465        address recipient,
466        uint256 amount
467    ) internal {
468        require(sender != address(0), 'BEP20: transfer from the zero address');
469        require(recipient != address(0), 'BEP20: transfer to the zero address');
```

- Here you are checking that account address is proper and valid address.

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

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

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

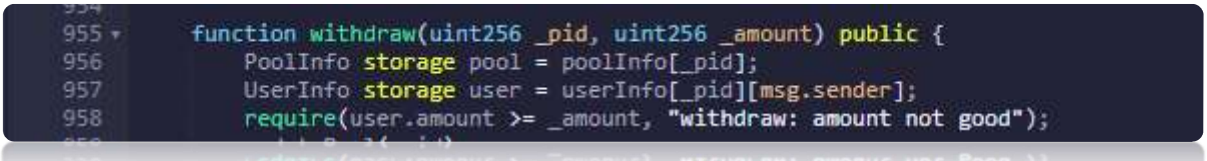
```
function _approve(
    address owner,
    address spender,
    uint256 amount
) internal {
    require(owner != address(0), 'BEP20: approve from the zero address');
    require(spender != address(0), 'BEP20: approve to the zero address');
```

- Here you are checking that _depositFeeBP value should be less than or equl to 10000.

```
function add(
    uint256 _allocPoint,
    IBEP20 _lpToken,
    uint16 _depositFeeBP,
    bool _withUpdate
) public onlyOwner {
    require(
        _depositFeeBP <= 10000,
        "add: invalid deposit fee"
    );
```

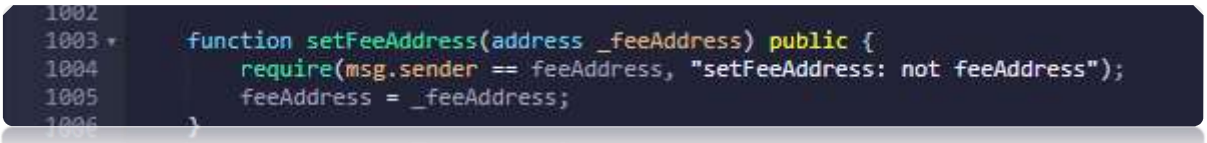
```
function set(
    uint256 _pid,
    uint256 _allocPoint,
    uint16 _depositFeeBP,
    bool _withUpdate
) public onlyOwner {
    require(
        _depositFeeBP <= 10000,
        "set: invalid deposit fee basis points"
    );
```

- Here you are checking that user can not withdraw more amounts then their balance.



```
function withdraw(uint256 _pid, uint256 _amount) public {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    require(user.amount >= _amount, "withdraw: amount not good");
```

- Here you are checking that only current feeAddress account can call this function.



```
function setFeeAddress(address _feeAddress) public {
    require(msg.sender == feeAddress, "setFeeAddress: not feeAddress");
    feeAddress = _feeAddress;
}
```

- **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: Short address attack:-

=> This is not big issue in solidity, because now a days is increased In the new solidity version. But it is good practice to Check for the short address.

=> As you are using latest version of solidity it's not mandatory.

=> In some functions you are not checking the value of Address parameter

- **Function: - constructor ('_devaddr', '_feeAddress')**

```
787     constructor(
788         BucksToken _bucks,
789         address _devaddr,
790         address _feeAddress,
791         uint256 _bucksPerBlock,
792         uint256 _startBlock
```

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

- **Function: - dev ('_devaddr')**

```
997     function dev(address _devaddr) public {
998         require(msg.sender == devaddr, "dev: not dev");
999         devaddr = _devaddr;
1000    }
```

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

- **Function: - setFeeAddress ('_feeAddress')**

```
1003    function setFeeAddress(address _feeAddress) public {
1004        require(msg.sender == feeAddress, "setFeeAddress: not feeAddress");
1005        feeAddress = _feeAddress;
1006    }
```

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

```
495     function _approve(
496         address owner,
497         address spender,
498         uint256 amount
499     ) internal {
500         require(owner != address(0), 'BEP20: approve from the zero address');
501         require(spender != address(0), 'BEP20: approve to the zero address');
```

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

7.3: Uncheck 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: - safeBucksTransfer**

```
987     function safeBucksTransfer(address _to, uint256 _amount) internal {
988         uint256 bucksBal = bucks.balanceOf(address(this));
989         if (_amount > bucksBal) {
990             bucks.transfer(_to, bucksBal);
991         } else {
992             bucks.transfer(_to, _amount);
```

  - Here you are calling transfer method of bucks contract 2 times. 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 is written in good and secured way, validation is done properly, SafeMath library is used.
- **Suggestions:** If possible check user address value is proper or not in all function, user balance in approve function, and return response of transfer method.