BuckSwap

SMART CONTRACT AUDIT REPORT
For
BucksToken

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

Table of Content

- Disclaimer
- Overview of the audit
- Attacks made to the contract
- Good things in smart contract
- Critical vulnerabilities found in the contract
- Medium vulnerabilities found in the contract
- Low severity vulnerabilities found in the contract
- Summary of the audit

• 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 656 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.

- ### 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.

```
212     function sendValue(address payable recipient, uint256 amount) internal {
213         require(address(this).balance >= amount, 'Address: insufficient balance');
214
215
216         (bool success, ) = recipient.call{value: amount}('');
217         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.

```
244     function functionCallWithValue(
245         address target,
246         bytes memory data,
247         uint256 value,
248         string memory errorMessage
249     ) internal returns (bytes memory) {
250         require(address(this).balance >= value, 'Address: insufficient balance for
```

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

```
254     function _functionCallWithValue(
255         address target,
256         bytes memory data,
257         uint256 weiValue,
258         string memory errorMessage
259     ) private returns (bytes memory) {
260         require(isContract(target), 'Address: call to non-contract');
```

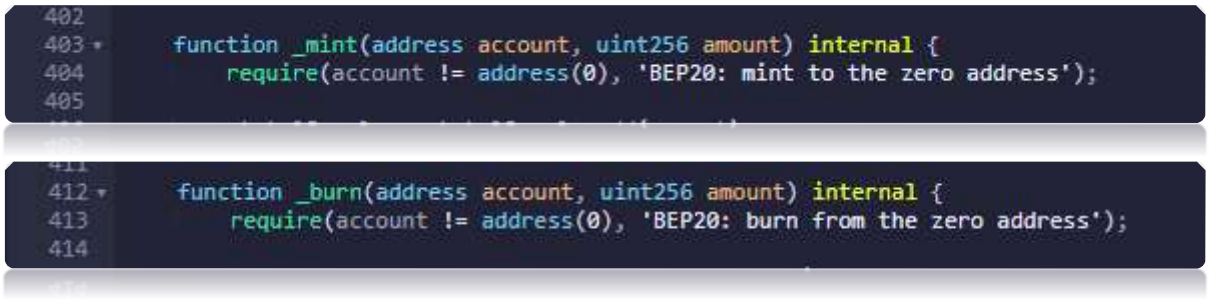
  - Here you are checking that newOwner address is proper and valid address.

```
53
54      function _transferOwnership(address newOwner) internal {
55          require(newOwner != address(0), 'Ownable: new owner is the zero address');
56          emit OwnershipTransferred(_owner, newOwner);
57          _owner = newOwner;
58      }
```

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

```
389     function _transfer(
390         address sender,
391         address recipient,
392         uint256 amount
393     ) internal {
394         require(sender != address(0), 'BEP20: transfer from the zero address');
395         require(recipient != address(0), 'BEP20: transfer to the zero address');
```

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



```
402
403     function _mint(address account, uint256 amount) internal {
404         require(account != address(0), 'BEP20: mint to the zero address');
405
```



```
412     function _burn(address account, uint256 amount) internal {
413         require(account != address(0), 'BEP20: burn from the zero address');
414
```

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

```
421     function _approve(
422         address owner,
423         address spender,
424         uint256 amount
425     ) internal {
426         require(owner != address(0), 'BEP20: approve from the zero address');
427         require(spender != address(0), 'BEP20: approve to the zero address');
428
```

- **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**

```
421     function _approve(
422         address owner,
423         address spender,
424         uint256 amount
425     ) internal {
426         require(owner != address(0), 'BEP20: approve from the zero address');
427         require(spender != address(0), 'BEP20: approve to the zero address');
```

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

• 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 balance in approve function, and try to implement safeMath library in BucksToken contract.