ZOMBIEFINANCE

SMART-CONTRACT AUDIT REPORT

BY SOLIDSECT

CONTRACT ADDRESS:

https://bscscan.com/address/0xe2a6428fd332287b0470965e16350d3cc1736e3e#code

Website: https://zombiefinance.net

Telegram: https://t.me/zombiefinance

Twitter: https://twitter.com/zombie_finance

Medium: https://medium.com/zombie-finance

Contents

Disclaimer

This audit makes no statements or warranties about the utility or safety of the code, the suitability or regulatory regime of the business model, or any other statements about the fitness of the contracts regarding their purpose or bug-free status. This audit documentation is for discussion purposes only.

AUDIT SUMMARY

Concluding, the inspected code is well-written and performs as expected. There is no back door to steal funds.

Please try to check the address and value of the token externally before sending this to the solidity code.

Our final recommendation is to pay more attention to the visibility of the functions, the hardcoded address, and the mapping, since it is quite important to define who is supposed to execute the functions and to follow best practices regarding the use of 'assert,' 'require,' etc. (which you are already doing, so great job on that).

Best Work: The address validation and value validation is done properly.

Suggestions: Please add address validations at some place, and try to use the static version of solidity; check the amount in the approve function.

TOKEN ALLOCATION

TOKEN ALLOCATION AND DISTRIBUTION

10.000.000 Initial Token Supply

8.000.000 Token Allocated for Presale and Exchange Listings - Verified.

1.000.000 Token Allocated for Development and Team - Verified.

1.000.000 Token Allocated for Marketing and Events - Verified.

TOKEN LOCK + BURN

%40 of PancakeSwap LP Tokens Burned - Verified.

%100 of JulSwap Dex LP Tokens Locked for 265 Years - Verified.

MECHANICS (on each transaction)

%3 Reflected Back to Current Holders - Verified.

%2 Burned Forever - Verified.

Audit Overview

The project has 1 file. It contains approx. 730 lines of Solidity code. All functions and state variables are well-commented using natspec documentation, which that does not create any vulnerability.

Executed Attacks 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. Yet, an underflow happens when you try to subtract 0 minus a number larger 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 Open Zeppelin's Safe Math to mitigate this attack, and 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 does not check the length of the address of the sender, the Tron's virtual machine will 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 up due to this (please see below). It is highly recommended to call functions after checking validity of the address.

Visibility and 'Delegate Call' Misuse

This is also known as Parity Hack, which occurs during misuse of a Delegate Call. **No such issues were found** in this smart contract, and the visibility was also properly addressed. There are some places where there is no visibility defined. The 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 that 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 Tron 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 mitigates this vulnerability.

Forcing BNB to a Contract

While implementing 'selfdestruct' in a smart contract, it sends all the BNB 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, a hacker can bypass the 'required' conditions. Here, the smart contract's balance has never been used as a guard, which mitigates this vulnerability.

Positive Traits in The Smart Contract

This function checks that the balance of the contract is larger or equal to the amount. You are also checking that the token is successfully transferred to the recipient's address.

```
303 ▾   function sendValue(address payable recipient, uint256 amount) internal {
304         require(address(this).balance >= amount, "Address: insufficient balance");
305
306         // solhint-disable-next-line avoid-low-level-calls, avoid-call-value
307         (bool success, ) = recipient.call{ value: amount }("");
308         require(success, "Address: unable to send value, recipient may have revert
```

This function checks that the contract has more than or a balance equal to the amount value.

```
364 ▾   function functionCallWithValue(address target, bytes memory data, uint256 valu
365         require(address(this).balance >= value, "Address: insufficient balance for
366         return _functionCallWithValue(target, data, value, errorMessage);
367     }
```

This function checks that the target address is a proper contract address.

```
369 ▾   function _functionCallWithValue(address target, bytes memory data, uint256 we
370         require(isContract(target), "Address: call to non-contract");
371
372         // solhint-disable-next-line avoid-low-level-calls
```

This function checks that the new owner address value is a proper valid address.

```
450 ▾   function transferOwnership(address newOwner) public virtual onlyOwner {
451         require(newOwner != address(0), "Ownable: new owner is the zero address")
452         emit OwnershipTransferred(_owner, newOwner);
453         _owner = newOwner;
454     }
```

This function checks that this function is not called by the address which is excluded.

```
554 ▾   function deliver(uint256 tAmount) public {
555         address sender = _msgSender();
556         require(!_isExcluded[sender], "Excluded addresses cannot call this functi
557         (uint256 rAmount,,,,,) = _getValues(tAmount);
558         _rOwned[sender] = _rOwned[sender].sub(rAmount);
```

This function checks that the _t Amount value is than or equal to the _t total amount (total token value).

```
563 ▾   function reflectionFromToken(uint256 tAmount, bool deductTransferFee) public
564         require(tAmount <= _tTotal, "Amount must be less than supply");
565 ▾       if (!deductTransferFee) {
566             (uint256 rAmount,,,,,) = _getValues(tAmount);
567             return rAmount;
```

This function checks that the _r amount value is less than or equal to the _r total amount (total reflections value).

```
574 ▾   function tokenFromReflection(uint256 rAmount) public view returns(uint256) {
575         require(rAmount <= _rTotal, "Amount must be less than total reflections")
576         uint256 currentRate =  _getRate();
577         return rAmount.div(currentRate);
578     }
```

This function checks that an account address is not already excluded from a reward, and that an account address is not the Uniswap router address.

```
580 ▾   function excludeAccount(address account) external onlyOwner() {
581         require(account != 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D, 'We can not
582         require(!_isExcluded[account], "Account is already excluded");
583 ▾       if(_rOwned[account] > 0) {
584             _tOwned[account] = tokenFromReflection(_rOwned[account]);
```

This function checks that an account address is not already included for a reward.

```
590     function includeAccount(address account) external onlyOwner() {
591         require(_isExcluded[account], "Account is already excluded");
592         for (uint256 i = 0; i < _excluded.length; i++) {
593             if (_excluded[i] == account) {
594                 _excluded[i] = _excluded[_excluded.length - 1];
```

This function checks that the owner and spender address values are proper addresses.

```
603     function _approve(address owner, address spender, uint256 amount) private {
604         require(owner != address(0), "BEP20: approve from the zero address");
605         require(spender != address(0), "BEP20: approve to the zero address");
606
```

This function checks that address values of 'from' and 'to' are proper. An amount should be over zero and less than _MAX_TX_SIZE (maximum amount to transfer token).

```
611     function _transfer(address sender, address recipient, uint256 amount) private
612         require(sender != address(0), "BEP20: transfer from the zero address");
613         require(recipient != address(0), "BEP20: transfer to the zero address");
614         require(amount > 0, "Transfer amount must be greater than zero");
615
```

Critical Vulnerabilities Found

=> No critical vulnerabilities found

Medium-Severity Vulnerabilities Found

=> No medium-severity vulnerabilities found

Low-Severity Vulnerabilities Found

Short address attack:

=> This is not a big issue in Solidity bedue to its latest release. But it is good practice to check for the short address.

=> After updating Solidity, it is no longer mandatory.

=> In some functions, you are not checking the value of the address parameter. The following will show all necessary functions.

Function: isContract ('account')

```
275      */
276     function isContract(address account) internal view returns (bool) {
277         // According to EIP-1052, 0x0 is the value returned for not-yet created ac
278         // and 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470
279         // for accounts without code, i.e. `keccak256('')`
280         bytes32 codehash;
281         bytes32 accountHash = 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7b
```

It is necessary to check the address value of the account. This is because here, you are passing whatever variable is included in the account address from the outside.

Function:- excludeAcoount, includeReward ('account')

```
580 ▾    function excludeAccount(address account) external onlyOwner() {
581          require(account != 0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D, 'We can not
582          require(!_isExcluded[account], "Account is already excluded");
583 ▾        if(_rOwned[account] > 0) {
584              _tOwned[account] = tokenFromReflection(_rOwned[account]);
585          }
```

```
590 ▾    function includeAccount(address account) external onlyOwner() {
591          require(_isExcluded[account], "Account is already excluded");
592 ▾        for (uint256 i = 0; i < _excluded.length; i++) {
593 ▾            if (_excluded[i] == account) {
```

It is necessary to check the address value of the account. This is because here, you are passing whatever variable is included in the account address from the outside.

Function: - _transferBothExcluded ('sender', 'recipient')

```
664 ▾    function _transferBothExcluded(address sender, address recipient, uint256 tAmo
665          uint256 currentRate =  _getRate();
666          (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransfer
667          uint256 rBurn =  tBurn.mul(currentRate);
668          _tOwned[sender] = _tOwned[sender].sub(tAmount);
669          _rOwned[sender] = _rOwned[sender].sub(rAmount);
```

It is necessary to check the address value of the sender and the recipient. This is because here, you are passing whatever variable is included in the sender and recipient addresses from the outside.

Function: _transferStandard, _transferToExcluded, _transferFromExcluded ('sender', 'recipient')

```
632     function _transferStandard(address sender, address recipient, uint256 tAmount)
633         uint256 currentRate =  _getRate();
634         (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransfer
635         uint256 rBurn =  tBurn.mul(currentRate);
```

```
642     function _transferToExcluded(address sender, address recipient, uint256 tAmoun
643         uint256 currentRate =  _getRate();
644         (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransfer
645         uint256 rBurn =  tBurn.mul(currentRate);
646         _rOwned[sender] = _rOwned[sender].sub(rAmount);
647          tOwned[recipient] =  tOwned[recipient].add(tTransferAmount);
```

```
653     function _transferFromExcluded(address sender, address recipient, uint256 tAmo
654         uint256 currentRate =  _getRate();
655         (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransfer
656         uint256 rBurn =  tBurn.mul(currentRate);
657         _tOwned[sender] = _tOwned[sender].sub(tAmount);
```

It is necessary to check the address value of the sender and the recipient. This is because here, you are passing whatever variable is included in the sender and recipient addresses from the outside.

Compiler version is not fixed:
=> In this file, you have to put "pragma solidity ^0.6.4;" which is not a good way to define a compiler version.

=> Solidity source files indicate the versions of the compiler they can be compiled with. Pragma solidity >=0.6.4; // bad: compiles 0.6.4 and above pragma solidity 0.6.4; //good: compiles 0.6.4 only.

=> If you put the (>=) symbol, then you are able to get the Compiler version

0.6.4 and above. But if you don't use the (^/>=) symbol, then you are limited to the 0.6.4 version. And if there are some changes in the Compiler and you use the old version, then issues may arise at deployment time.

=> Use the latest version of Solidity.

Approve can exceed allowance:

=> I have found that in the approve function, the user can give more allowance to a user beyond their balance.

=> It is necessary to check that a user can only give an allowance less or equal to their holdings.

=> There is no validation of the user balance. This is required to ensure that a user cannot set approval incorrectly.

Function: _approve

```
603     function _approve(address owner, address spender, uint256 amount) private {
604         require(owner != address(0), "BEP20: approve from the zero address");
605         require(spender != address(0), "BEP20: approve to the zero address");
606 
607         _allowances[owner][spender] = amount;
608         emit Approval(owner, spender, amount);
```

Here you can check if you have more allowance than balance.

END OF REPORT – 14 PAGES