Smart Contract
Security Audit Report

The SlowMist Security Team received the team's application for smart contract security audit of the ENTERBUTTON on 2022.04.19. The following are the details and results of this smart contract security audit:

Token Name :

ENTERBUTTON

The contract address :

0x3Ecab35B64345bfC472477A653e4A3abE70532D9

The audit items and results :

(Other unknown security vulnerabilities are not included in the audit responsibility scope)

NO.	Audit Items	Result
1	Replay Vulnerability	Passed
2	Denial of Service Vulnerability	Passed
3	Race Conditions Vulnerability	Passed
4	Authority Control Vulnerability	Passed
5	Integer Overflow and Underflow Vulnerability	Passed
6	Gas Optimization Audit	Passed
7	Design Logic Audit	Passed
8	Uninitialized Storage Pointers Vulnerability	Passed
9	Arithmetic Accuracy Deviation Vulnerability	Passed
10	"False top-up" Vulnerability	Passed
11	Malicious Event Log Audit	Passed
12	Scoping and Declarations Audit	Passed

NO.	Audit Items	Result
13	Safety Design Audit	Passed
14	Non-privacy/Non-dark Coin Audit	Passed

Audit Result : Passed

Audit Number : 0X002204220003

Audit Date : 2022.04.19 - 2022.04.22

Audit Team : SlowMist Security Team

Summary conclusion : This is a token contract that contains the tokenVault section. The total amount of contract tokens can be changed, the owner can burn his tokens through the burn function. SafeMath security module is used, which is a recommended approach. The contract does not have the Overflow and the Race Conditions issue.

During the audit, we found the following information:

1. The owner role can lock users' tokens at a specified period of time, but the lock release time is before the lock expires a count of period.
2. The owner role can unlock the time lock to release the locked tokens before the lock expires.

The source code:

```
/**
 *Submitted for verification at Etherscan.io on 2021-09-10
*/

// ----------------------------------------------------------------------------
// ENTERBUTTON Contract
// Name        : ENTERBUTTON
// Symbol      : ENTC
// Decimals    : 18
// InitialSupply : 10,000,000,000 ENTC
// ----------------------------------------------------------------------------
//SlowMist// The contract does not have the Overflow and the Race
pragma solidity 0.5.8;
```

```
interface IERC20 {

    function totalSupply() external view returns (uint256);

    function balanceOf(address account) external view returns (uint256);

    function transfer(address recipient, uint256 amount) external returns (bool);

    function allowance(address owner, address spender) external view returns (uint256);

    function approve(address spender, uint256 amount) external returns (bool);

    function transferFrom(address sender, address recipient, uint256 amount) external returns
(bool);

    event Transfer(address indexed from, address indexed to, uint256 value);

    event Approval(address indexed owner, address indexed spender, uint256 value);
}
//SlowMist// SafeMath security module is used, which is a recommended approach
library SafeMath {

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b <= a, "SafeMath: subtraction overflow");
        uint256 c = a - b;

        return c;
    }

    function mul(uint256 a, uint256 b) internal pure returns (uint256) {

        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
```

```
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b > 0, "SafeMath: division by zero");
        uint256 c = a / b;

        return c;
    }

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b != 0, "SafeMath: modulo by zero");
        return a % b;
    }
}

contract ERC20 is IERC20 {
    using SafeMath for uint256;

    mapping (address => uint256) internal _balances;

    mapping (address => mapping (address => uint256)) private _allowances;

    uint256 private _totalSupply;

    function totalSupply() public view returns (uint256) {
        return _totalSupply;
    }

    function balanceOf(address account) public view returns (uint256) {
        return _balances[account];
    }

    function transfer(address recipient, uint256 amount) public returns (bool) {
        _transfer(msg.sender, recipient, amount);
        //SlowMist// The return value conforms to the EIP20 specification
        return true;
    }

    function allowance(address owner, address spender) public view returns (uint256) {
        return _allowances[owner][spender];
    }
```

```
    function approve(address spender, uint256 value) public returns (bool) {
        _approve(msg.sender, spender, value);
        //SlowMist// The return value conforms to the EIP20 specification
        return true;
    }

    function transferFrom(address sender, address recipient, uint256 amount) public returns
(bool) {
        _transfer(sender, recipient, amount);
        _approve(sender, msg.sender, _allowances[sender][msg.sender].sub(amount));
        //SlowMist// The return value conforms to the EIP20 specification
        return true;
    }

    function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
        _approve(msg.sender, spender, _allowances[msg.sender][spender].add(addedValue));
        return true;
    }

    function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool)
{
        _approve(msg.sender, spender, _allowances[msg.sender][spender].sub(subtractedValue));
        return true;
    }

    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
        //SlowMist// This kind of check is very good, avoiding user mistake leading to the
loss of token during transfer
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _balances[sender] = _balances[sender].sub(amount);
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }

    function _mint(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: mint to the zero address");

        _totalSupply = _totalSupply.add(amount);
        _balances[account] = _balances[account].add(amount);
        emit Transfer(address(0), account, amount);
    }
```

```
    function _burn(address owner, uint256 value) internal {
        require(owner != address(0), "ERC20: burn from the zero address");

        _totalSupply = _totalSupply.sub(value);
        _balances[owner] = _balances[owner].sub(value);
        emit Transfer(owner, address(0), value);
    }

    function _approve(address owner, address spender, uint256 value) internal {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = value;
        emit Approval(owner, spender, value);
    }

    function _burnFrom(address owner, uint256 amount) internal {
        _burn(owner, amount);
        _approve(owner, msg.sender, _allowances[owner][msg.sender].sub(amount));
    }
}

contract ENTERBUTTON is ERC20 {
    string public constant name = "ENTERBUTTON";
    string public constant symbol = "ENTC";
    uint8 public constant decimals = 18;
    uint256 public constant initialSupply = 10000000000 * (10 ** uint256(decimals));

    constructor() public {
        super._mint(msg.sender, initialSupply);
        owner = msg.sender;
    }

    address public owner;

    event OwnershipRenounced(address indexed previousOwner);
    event OwnershipTransferred(
    address indexed previousOwner,
    address indexed newOwner
    );

    modifier onlyOwner() {
        require(msg.sender == owner, "Not owner");
```

```
        _;
    }

    function transferOwnership(address _newOwner) public onlyOwner {
        _transferOwnership(_newOwner);
    }

    function _transferOwnership(address _newOwner) internal {
        //SlowMist// This check is quite good in avoiding losing control of the contract caused by user mistakes
        require(_newOwner != address(0), "Already Owner");
        emit OwnershipTransferred(owner, _newOwner);
        owner = _newOwner;
    }

    function transfer(
        address _to,
        uint256 _value
    )
      public

      returns (bool)
    {
        releaseLock(msg.sender);
        return super.transfer(_to, _value);
    }

    function transferFrom(
        address _from,
        address _to,
        uint256 _value
    )
      public

      returns (bool)
    {
        releaseLock(_from);
        return super.transferFrom(_from, _to, _value);
    }

    event Burn(address indexed burner, uint256 value);

    function burn(uint256 _value) public onlyOwner {
        require(_value <= super.balanceOf(owner), "Balance is too small.");
```

```
    _burn(owner, _value);
    emit Burn(owner, _value);
}

struct LockInfo {
    uint256 releaseTime;
    uint256 balance;
}
mapping(address => LockInfo[]) internal lockInfo;

event Lock(address indexed holder, uint256 value, uint256 releaseTime);
event Unlock(address indexed holder, uint256 value);

function balanceOf(address _holder) public view returns (uint256 balance) {
    uint256 lockedBalance = 0;
    for(uint256 i = 0; i < lockInfo[_holder].length ; i++ ) {
        lockedBalance = lockedBalance.add(lockInfo[_holder][i].balance);
    }
    return super.balanceOf(_holder).add(lockedBalance);
}

function releaseLock(address _holder) internal {

    for(uint256 i = 0; i < lockInfo[_holder].length ; i++ ) {
        if (lockInfo[_holder][i].releaseTime <= now) {
            _balances[_holder] = _balances[_holder].add(lockInfo[_holder][i].balance);
            emit Unlock(_holder, lockInfo[_holder][i].balance);
            lockInfo[_holder][i].balance = 0;

            if (i != lockInfo[_holder].length - 1) {
                lockInfo[_holder][i] = lockInfo[_holder][lockInfo[_holder].length - 1];
                i--;
            }
            lockInfo[_holder].length--;

        }
    }
}
function lockCount(address _holder) public view returns (uint256) {
    return lockInfo[_holder].length;
}
function lockState(address _holder, uint256 _idx) public view returns (uint256, uint256) {
    return (lockInfo[_holder][_idx].releaseTime, lockInfo[_holder][_idx].balance);
```

```
    }
    //SlowMist// The owner role can lock users' tokens at a specified period of time, but the lock release time is before the lock expires a count of period
    function lock(address _holder, uint256 _amount, uint256 _releaseTime) public onlyOwner {
        require(super.balanceOf(_holder) >= _amount, "Balance is too small.");
        require(block.timestamp <= _releaseTime, "TokenTimelock: release time is before current time");

        _balances[_holder] = _balances[_holder].sub(_amount);
        lockInfo[_holder].push(
            LockInfo(_releaseTime, _amount)
        );
        emit Lock(_holder, _amount, _releaseTime);
    }
    //SlowMist// The owner role can unlock the time lock to release the locked tokens before the lock expires
    function unlock(address _holder, uint256 i) public onlyOwner {
        require(i < lockInfo[_holder].length, "No lock information.");

        _balances[_holder] = _balances[_holder].add(lockInfo[_holder][i].balance);
        emit Unlock(_holder, lockInfo[_holder][i].balance);
        lockInfo[_holder][i].balance = 0;

        if (i != lockInfo[_holder].length - 1) {
            lockInfo[_holder][i] = lockInfo[_holder][lockInfo[_holder].length - 1];
        }
        lockInfo[_holder].length--;
    }

    function transferWithLock(address _to, uint256 _value, uint256 _releaseTime) public onlyOwner returns (bool) {
        require(_to != address(0), "wrong address");
        require(_value <= super.balanceOf(owner), "Not enough balance");
        require(block.timestamp <= _releaseTime, "TokenTimelock: release time is before current time");

        _balances[owner] = _balances[owner].sub(_value);
        lockInfo[_to].push(
            LockInfo(_releaseTime, _value)
        );
        emit Transfer(owner, _to, _value);
        emit Lock(_to, _value, _releaseTime);

        return true;
```

```
    }
}
```

Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist