Millionaire

Intelligent contract audit report

Safety status

Safety

★ ★ ★ ★ ★

Main surveyor: know Chuangyu block chain security research team

Version description

Revised content	Time	Reviser	Version number
Write a document	20220622	Know that Chuangyu block chain security research team	V1.0

Document information

Document name	Document version	Report number	Level secrecy
Millionaire Audit report	V1.0	b060cfb0228a4667a4a9aeed43c71d42	Project team Disclosure

Statement

Chuangyu only issues this report on the facts that occurred or existed before the publication of this report, and bears the responsibility for it. Responsible Chuangyu is unable to judge the security status of its intelligent contract
and is not responsible for the fact that it occurs or exists in the future.
The security audit analysis and other contents in this report are only based on the documents and information provided by the information provider to Chuangyu as of the time of publication of this report.
Chuangyu hypothesis: there is no deletion, tampering, deletion or concealment of the provided information.
If the information provided is missing, tampered with, deleted, concealed or reflected that is not in line with the actual situation, Chuangyu shall not be liable for the losses and adverse effects caused by it.

Catalogue

1. Summary

The effective testing time of this report is from Jun 19, 2022 to Jun 22, 2021. Bundle, during this period, the security and standardization of token codesfor M i l l i o n an i r e intelligent contracts. Conduct an audit and use it as a statistical basis for the report.

The scope of this intelligent contract security audit does not include external contract calls, and does not include possible occurrence in the future.
The new attack method does not include the upgraded or tampered code of the contract (with the development of the project, the intelligent contract may add new pool, new functional modules, new external contract calls, etc.), and does not include front-end security and server security.

In this test, it is known that Chuangyu engineer made a comprehensive analysis of the common loopholes in the intelligent contract (see Chapter 3) and passed the comprehensive evaluation.

The results of this intelligent contract security audit：through

As the testing process is carried out in a non-production environment, and all the codes are up-to-date, the testing process communicates with relevant interfaces and carries out relevant testing operations when the operational risk is controllable, in order to avoid the production operation risk and code safety risk in the testing process

Report information of this audit：

Report number：**b060cfb0228a4667a4a9aeed43c71d42**

Target information of this audit：

Entry	Description
Project name	Millionaire(Mai)
Contract address	0x0DA2e5110F182B49A3C8F420C0F19893E5E10A21
Code type	Token code, BEP-20 intelligent contract code
Code language	Solidity

Contract documents and hash:

Contract document	MD5
MaiToken.sol	0xd4afffb57c041b0591d3658bba93b4b30505c55334a95a9053c7bc988f945b70

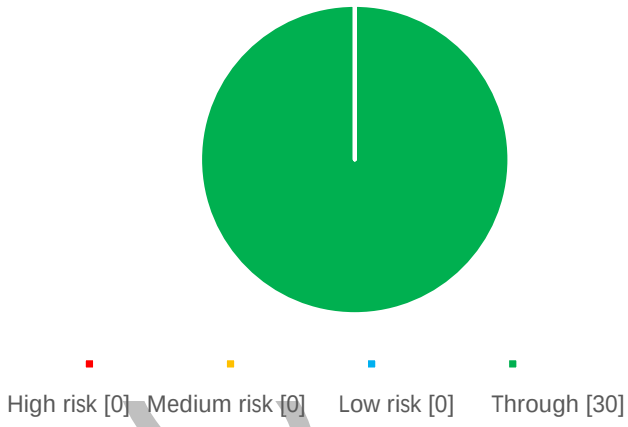
2. Code vulnerability analysis

2.1 Vulnerability level distribution

The risk of this vulnerability is calculated by level:

安全风险等级个数统计表			
High risk	Medium risk	Low risk	Through
0	0	0	30

Risk grade distribution map



2.2 Summary of audit results

Audit project	Audit content	Status	Description
Business security	Token function	through	After testing, there is no security problem
	Dividend account setting function	through	After testing, there is no security problem
	Transfer function	through	After testing, there is no security problem
Coding security	Compiler version security	through	After testing, there is no security problem
	Redundant code	through	After testing, there is no security problem
	Use of secure arithmetic libraries	through	After testing, there is no security problem
	An unrecommended coding method	through	After testing, there is no security problem
	`require/assert` **Of R**easonable use	through	After testing, there is no security problem
	`fallback` Function security	through	After testing, there is no security problem
	`tx.orgin` Authentication	through	After testing, there is no security problem
	`owner` Authority control	through	After testing, there is no security problem
	`gas` Consumption detection	through	After testing, there is no security problem
	`call` Injection attack	through	After testing, there is no security problem
	Low-level function security	through	After testing, there is no security problem
	Additional issuance token loopholes	through	After testing, there is no security problem
	Access control defect detection	through	After testing, there is no security problem
	Numerical overflow detection	through	After testing, there is no security problem
	Arithmetic accuracy error	through	After testing, there is no security problem
	Misuse Random number detection	through	After testing, there is no security problem
	Use of unsafe interfaces	through	After testing, there is no security problem
	Variable coverage	through	After testing, there is no security problem
	Uninitialized storage pointer	through	After testing, there is no security problem
	Return value invokes verification	through	After testing, there is no security problem
	Transaction order dependence	through	After testing, there is no security problem

	Timestamp dependent attack	through	After testing, there is no security problem
	Denial of service attack	through	After testing, there is no security problem
	False recharge loophole	through	After testing, there is no security problem
	Reentrant attack detection	through	After testing, there is no security problem
	Replay attack detection	through	After testing, there is no security problem
	Rearrangement attack detection	through	After testing, there is no security problem

3. Service security detection

3.1. Token function【through】

Audit analysis：Tokens are inherited from IERC20 contracts and comply with TRC20 tokens contract standards. After audit, the function of token is complete and the logical design is reasonable.

```
contract MaiToken is Context, IERC20, Ownable {
    using SafeMath for uint256;// Millionaire Use a secure arithmetic library
    using Address for address;

    //////////////////////////////// constant ////////////////////////////////
    address constant BURN_ADDRESS = 0x000000000000000000000000000000000000dEaD;
    address constant DIVIDENDS_ADDRESS = 0x23128A25E9285cd62ac79a5659C65fbeb86EE31e;
    address constant LP_ADDRESS = 0x9E55bda94dFc8fe18898b715602846BC419915A4;

    //////////////////////////////// storage ////////////////////////////////
    string private _name = "Mai Token";// Millionaire Token name
    string private _symbol = "Mai";// Millionaire Token symbol
    uint8 private _decimals = 18;// Millionaire Token precision

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

    mapping (address => bool) private _isExcludedFromFee;

    mapping (address => bool) private _isExcluded;
    address[] private _excluded;

    uint256 private constant MAX = ~uint256(0);
```

```
uint256 private _tTotal = 100_000_000 * 10 ** _decimals;// Millionaire Total circulation
uint256 private _rTotal = (MAX - (MAX % _tTotal));
uint256 private _tFeeTotal;

uint256 public _taxFee = 1;
uint256 private _previousTaxFee = _taxFee;


  uint256 public _dividendsFee = 1;
  uint256 public _burnFee = 2;
  uint256 public _specifyFee = 1;

  uint256 public _liquidityFee = _dividendsFee + _burnFee + _specifyFee;
  uint256 private _previousLiquidityFee = _liquidityFee;



  constructor () {// Millionaire Constructor function
      _rOwned[_msgSender()] = _rTotal;

      //exclude owner and this contract from fee
      _isExcludedFromFee[owner()] = true;
      _isExcludedFromFee[address(this)] = true;

      emit Transfer(address(0), _msgSender(), _tTotal);
  }
```

Safety recommendation：None.

3.2. Dividend account setting function【through】

Audit analysis：The contract is passed._isExcluded[account] Judge whether it is a dividend account, Owner By calling the excludeFromReward And includeInReward Realize the function of setting the user not to participate in the dividend and setting the user to participate in the dividend.

After audit, the authority control is correct and the logical design is reasonable.

```
function excludeFromReward(address account) public onlyOwner() {// Millionaire Set users not to participate in dividends
        require(!_isExcluded[account], "Account is already excluded");
        if(_rOwned[account] > 0) {
            _tOwned[account] = tokenFromReflection(_rOwned[account]);
        }
        _isExcluded[account] = true;
        _excluded.push(account);
    }

    function includeInReward(address account) external onlyOwner() {// Millionaire Set up user participation dividend

        require(_isExcluded[account], "Account is already excluded");
        for (uint256 i = 0; i < _excluded.length; i++) {
            if (_excluded[i] == account) {
                _excluded[i] = _excluded[_excluded.length - 1];
                _tOwned[account] = 0;
                _isExcluded[account] = false;
                _excluded.pop();
                break;
            }
        }
    }
```

Safety recommendation：None.

3.3. Transfer function【through】

Audit analysis：Check account number and amount of transfer before transfer, judge whether to charge service fee

Make token transfers. The transfer is completed after determining whether the sender and receiver are dividend accounts and using different transfer functions

according to the actual situation and deducting transfer fees and liquidity fees.

After audit, the authority control is correct and the function design is reasonable.

```
function _transfer(
        address from,
        address to,
        uint256 amount
    ) private {// Millionaire Transfer
        require(from != address(0), "TRC20: transfer from the zero address");
        require(to != address(0), "TRC20: transfer to the zero address");
        require(amount > 0, "Transfer amount must be greater than zero");

        //indicates if fee should be deducted from transfer
        bool takeFee = true;

        //if any account belongs to _isExcludedFromFee account then remove the fee
        if(_isExcludedFromFee[from] || _isExcludedFromFee[to]){// Millionaire Judge
        whether it is a whitelist, and if it is a whitelist, there is no
        service charge.
            takeFee = false;
        }

        //transfer amount, it will take tax, burn, liquidity fee
        _tokenTransfer(from,to,amount,takeFee);
    }

    //this method is responsible for taking all fee, if takeFee is true
    function _tokenTransfer(address sender, address recipient, uint256 amount,bool takeFee)
private {// Millionaire Actual transfer
        if(!takeFee)
            removeAllFee();
        // Millionaire Judge whether it is a dividend account and select the
        transfer channel
            if (_isExcluded[sender] && !_isExcluded[recipient]) {
            _transferFromExcluded(sender, recipient, amount);
```

```
        } else if (!_isExcluded[sender] && _isExcluded[recipient]) {
            _transferToExcluded(sender, recipient, amount);
        } else if (!_isExcluded[sender] && !_isExcluded[recipient]) {
            _transferStandard(sender, recipient, amount);
        } else if (_isExcluded[sender] && _isExcluded[recipient]) {
            _transferBothExcluded(sender, recipient, amount);
        } else {
            _transferStandard(sender, recipient, amount);
        }

        if(!takeFee)
            restoreAllFee();
    }

    function _transferStandard(address sender, address recipient, uint256 tAmount) private {
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransferAmount,
uint256 tFee, uint256 tLiquidity) = _getValues(tAmount);
        _rOwned[sender] = _rOwned[sender].sub(rAmount);
        _rOwned[recipient] = _rOwned[recipient].add(rTransferAmount);
        if (0 < tLiquidity) {
            _takeLiquidity(tLiquidity * _specifyFee / _liquidityFee);// Millionaire Mobile fee
            _takeDividends(tLiquidity * _dividendsFee / _liquidityFee);// Millionaire The
            holder of the coin receives dividends

            _takeBurn(tLiquidity * _burnFee / _liquidityFee);
        }
        _reflectFee(rFee, tFee);
        emit Transfer(sender, recipient, tTransferAmount);
    }

    function _transferToExcluded(address sender, address recipient, uint256 tAmount) private {
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransferAmount,
uint256 tFee, uint256 tLiquidity) = _getValues(tAmount);
        _rOwned[sender] = _rOwned[sender].sub(rAmount);
```

```
        _tOwned[recipient] = _tOwned[recipient].add(tTransferAmount);
        _rOwned[recipient] = _rOwned[recipient].add(rTransferAmount);
        if (0 < tLiquidity) {
            _takeLiquidity(tLiquidity * _specifyFee / _liquidityFee);
            _takeDividends(tLiquidity * _dividendsFee / _liquidityFee);
            _takeBurn(tLiquidity * _burnFee / _liquidityFee);
        }
        _reflectFee(rFee, tFee);
        emit Transfer(sender, recipient, tTransferAmount);
    }

    function _transferFromExcluded(address sender, address recipient, uint256 tAmount) private
{
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransferAmount,
uint256 tFee, uint256 tLiquidity) = _getValues(tAmount);
        _tOwned[sender] = _tOwned[sender].sub(tAmount);
        _rOwned[sender] = _rOwned[sender].sub(rAmount);
        _rOwned[recipient] = _rOwned[recipient].add(rTransferAmount);
        if (0 < tLiquidity) {
            _takeLiquidity(tLiquidity * _specifyFee / _liquidityFee);
            _takeDividends(tLiquidity * _dividendsFee / _liquidityFee);
            _takeBurn(tLiquidity * _burnFee / _liquidityFee);
        }
        _reflectFee(rFee, tFee);
        emit Transfer(sender, recipient, tTransferAmount);
    }

    function _transferBothExcluded(address sender, address recipient, uint256 tAmount) private {
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransferAmount,
uint256 tFee, uint256 tLiquidity) = _getValues(tAmount);
        _tOwned[sender] = _tOwned[sender].sub(tAmount);
        _rOwned[sender] = _rOwned[sender].sub(rAmount);
        _tOwned[recipient] = _tOwned[recipient].add(tTransferAmount);
```

```
        _rOwned[recipient] = _rOwned[recipient].add(rTransferAmount);
        if (0 < tLiquidity) {
            _takeLiquidity(tLiquidity * _specifyFee / _liquidityFee);
            _takeDividends(tLiquidity * _dividendsFee / _liquidityFee);
            _takeBurn(tLiquidity * _burnFee / _liquidityFee);
        }
        _reflectFee(rFee, tFee);
        emit Transfer(sender, recipient, tTransferAmount);
    }
```

Safety recommendation：None.

4. Code basic vulnerability detection

4.1. Compiler version security【through】

Check to see if a secure compiler version is used in the contract code implementation

Test result：After testing, the compiler version ^ 0.8.0 is established in the

intelligent contract code, and there is no security problem.

Safety recommendation：None.

4.2. Redundant code【through】

Check if redundant code is included in the contract code implementation

Test result：It has been tested that the security problem

does not exist in the intelligent contract code.

Safety recommendation：None.

4.3. The use of secure arithmetic Library【through】

Check whether the SafeMath secure arithmetic library is used in the contract code implementation

Test result：After testing, the SafeMath security arithmetic library has been used in the intelligent contract code, and the security problem does not exist.

Safety recommendation：None.

4.4. An unrecommended coding method【through】

Check if there are any coding methods that are not officially recommended or discarded in the implementation of the contract code.

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.5. require/assert Rational use of【through】

Check the reasonableness of the use of require and assert

statements in the contract code implementation

Test result：It has been tested that the security problem

does not exist in the intelligent contract code.

Safety recommendation：None.

4.6. fallback Function security【through】

Check whether the fallback function is used correctly in the

contract code implementation

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.7. tx.origin Authentication【through】

Tx.origin is a global variable of Solidity that traverses the call stack and returns the address of the account that originally sent the call (or transaction).

Using this variable for authentication in a smart contract makes the contract vulnerable to phishing attacks.

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.8. owner Authority control【through】

Check to see if the owner in the contract code implementation has excessive permissions. For example, arbitrarily modify other account balances and so on.

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.9. gas Consumption detection【through】

Check whether the gas consumption exceeds the block maximum limit

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.10. call Injection attack【through】

call When a function is called, it should be strictly controlled or written to death call The function called.

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.11. Low-level function security【through】

Check whether there are security vulnerabilities in the use of low-level functions (call/delegatecall) in the contract code implementation

call The execution context of the function is in the contract being called; And delegatecall On the execution of the function

The following is in the contract that currently calls the function

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.12. Loophole in the issuance of additional tokens【through】

Check whether there is a function in the token contract that may increase the total amount of tokens after initializing the total amount of tokens.

Test result：After testing, the intelligent contract code does not contain the function of issuing additional tokens.

Safety recommendation：None.

4.13. Access control defect detection【through】

Reasonable permissions should be set for different functions in the contract.

Check whether the functions in the contract correctly use public, private and other keywords for visibility modification, check whether the contract is correctly defined and use modifier to restrict access to key functions to avoid problems caused by ultra vires.

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.14. Numerical overflow detection【through】

The arithmetic problem in intelligent contract refers to integer overflow and integer underflow.

Solidity can handle up to 256 digits (2 ^ 256-1), and increasing the maximum number by 1 will overflow to 0.
Similarly, when the number is unsigned, 0 minus 1 overflows to the maximum numeric value.

Integer overflows and underflows are not a new type of vulnerability, but they are particularly dangerous in smart contracts.
Overflow conditions can lead to incorrect results, especially if the possibility is not expected, which may affect the reliability and security of the program.

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.15. Arithmetic accuracy error【through】

As a programming language, Solidity has similar data structure design to ordinary programming languages, such as variables, constants, functions, arrays, functions, structures, and so on. There is also a big difference between Solidity and ordinary programming languages-Solidity does not have floating point, and all numerical operations of Solidity will only be integers, there will be no decimal number, and it is not allowed to define decimal data.

The numerical operation in the contract is essential, and the design of the numerical operation may cause relative errors, such as the sibling operation: 5, 2, 10, 20, and 5, 10, 10, and 25, resulting in errors, and the errors will be larger and more obvious when the data is larger.

Test result: It has been tested that the security problem does not exist in the intelligent contract code.
Safety recommendation: None.

4.16. Incorrect use of random numbers【through】

Random numbers may be used in smart contracts, and although the functions and variables provided by Solidity can access obviously unpredictable values, such as block.number and block.timestamp, they are usually either more public than they seem, or influenced by miners, that is, these random numbers are somewhat predictable, so malicious users can usually copy it and rely on its unpredictability to attack the feature.

Test result: It has been tested that the security problem does not exist in the intelligent contract code.
Safety recommendation: None.

4.17. Use of unsafe interfaces【through】

Check whether unsafe interfaces are used in the contract code implementation

Test result: It has been tested that the security problem does not exist in the intelligent contract code.
Safety recommendation: None.

4.18. Variable coverage【through】

Check whether there are security problems caused by variable coverage in the contract code implementation

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.19. Uninitialized storage pointer【through】

In solidity, a special data structure is allowed to be a struct structure, while local variables within a function are stored using storage or memory by default. The existence of storage (memory) and memory (memory) are two different concepts. Solidity allows pointers to point to an uninitialized reference, while uninitialized local stroage will cause variables to point to other stored variables, leading to variable overrides, and even other more serious consequences. Initialization of struct variables in functions should be avoided in development.

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.20. Return value invokes verification【through】

This problem often occurs in intelligent contracts related to money transfer, so it is also known as silent failure transmission or unchecked transmission.

There are transfer (), send (), call.value () and other currency conversion methods in Solidity, which can all be used to send BNB to a certain address. The difference lies in: when transfer fails to send, it will throw and rollback the status; it will only pass 2300gas for call to prevent reentrant attacks; when send fails to send, it will return false; only for 2300gas to prevent reentrant attacks; when call. value fails to send, it will return false.

Passing all available gas calls (which can be restricted by passing in gas_value parameters) does not effectively prevent reentrant attacks.

If the return values of the above send and call.value money transfer functions are not checked in the code, the contract will continue to execute the following code, which may result in unexpected results due to the failure of TRX delivery.

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.21. Transaction order dependence【through】

Because miners always get gas fees through codes that represent externally owned addresses (EOA), users can specify higher fees to trade faster. Because the block chain is public, everyone can see the content of other people'soutst anding transactions. This means that if a user submits a valuable solution, a ma licious user can steal the solution and copy its transaction at a higher cost to pr eempt the original solution.

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.22. Timestamp dependent attack【through】

Generally speaking, the timestamp of the data block uses the miner's local time, and this time can fluctuate over a range of about 900 seconds. When other nodes accept a new block, you only need to verify that the timestamp is later than the previous block and the error with the local time is less than 900 seconds.

A miner can profit from it by setting a timestamp of the block to meet as many conditions as possible in his favor.

Check whether there are key functions that depend on timestamps in the contract code implementation

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.23. Denial of service attack【through】

In the world of Tron, denial of service is fatal, and smart contracts subjected to this type of attack may last forever.

It is far from being able to return to normal working condition. There may be many reasons for the denial of service of the smart contract, including malicious behavior as the recipient of the transaction, gas exhaustion caused by artificially increasing the gas needed for computing functions, abuse of access control to access the private components of the smart contract, confusion and negligence, and so on.

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.24. False recharge loophole【through】

In the transfer function of the token contract, the if judgment method is used to check the balance of the transfer initiator (msg.sender). When balances [msg. sender] < value, enter the else logic part and returnfalse, and finally no exception is thrown. We think that only the mild judgment method of if/else is a lax coding method in sensitive function scenarios such as transfer.

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.25. Reentrant attack detection【through】

The call.value () function in Solidity consumes all the gas it receives when it is used to send TRX, and there is a risk of reentrant attacks when calling the call.value () function to send TRX occurs before the balance of the sender's account is actually reduced.

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.26. Replay attack detection【through】

If the requirements of entrusted management are involved in the contract, we should pay attention to the non-reusability of verification and avoid replay attacks. In the asset management system, there is often entrusted management, and the principal manages the assets to the trustee, and the principal pays a certain fee to the trustee.

This business scenario is also common in smart contracts.

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

4.27. Rearrangement attack detection【through】

A rearrangement attack is an attempt by miners or other parties to "compete" with smart contract participants by inserting their own information into a list (list) or mapping (mapping), giving the attacker the opportunity to store his information in the contract.

Test result：It has been tested that the security problem does not exist in the intelligent contract code.

Safety recommendation：None.

5. Appendix A: Contract code

The source of this test code is:

MaiToken.sol

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;
pragma experimental ABIEncoderV2;

import "@openzeppelin/contracts/utils/Context.sol";
import "@openzeppelin/contracts/utils/Address.sol";
import "@openzeppelin/contracts/utils/math/SafeMath.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";


contract MaiToken is Context, IERC20, Ownable {
    using SafeMath for uint256;// Millionaire Use a secure arithmetic library
    using Address for address;

    ////////////////////////////////// constant //////////////////////////////////
    address constant BURN_ADDRESS = 0x000000000000000000000000000000000000dEaD;
    address constant DIVIDENDS_ADDRESS = 0x23128A25E9285cd62ac79a5659C65fbeb86EE31e;
    address constant LP_ADDRESS = 0x9E55bda94dFc8fe18898b715602846BC4199154A4;

    ////////////////////////////////// storage //////////////////////////////////
    string private _name = "Mai Token";// Millionaire Token name
    string private _symbol = "Mai";// Millionaire Token symbol
    uint8 private _decimals = 18;// Millionaire Token precision

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

    mapping (address => bool) private _isExcludedFromFee;

    mapping (address => bool) private _isExcluded;
    address[] private _excluded;

    uint256 private constant MAX = ~uint256(0);

    uint256 private _tTotal = 100_000_000 * 10 ** _decimals;// Millionaire Total circulation
    uint256 private _rTotal = (MAX - (MAX % _tTotal));
    uint256 private _tFeeTotal;

    uint256 public _taxFee = 1;
    uint256 private _previousTaxFee = _taxFee;

    uint256 public _dividendsFee = 1;
    uint256 public _burnFee = 2;
    uint256 public _specifyFee = 1;

    uint256 public _liquidityFee = _dividendsFee + _burnFee + _specifyFee;
    uint256 private _previousLiquidityFee = _liquidityFee;


    constructor () {// Millionaire Constructor function
        _rOwned[_msgSender()] = _rTotal;

        //exclude owner and this contract from fee
        _isExcludedFromFee[owner()] = true;
        _isExcludedFromFee[address(this)] = true;

        emit Transfer(address(0), _msgSender(), _tTotal);
    }

    function name() public view returns (string memory) {
        return _name;
    }

    function symbol() public view returns (string memory) {
        return _symbol;
    }

    function decimals() public view returns (uint8) {
        return _decimals;
    }

    function totalSupply() public view override returns (uint256) {
        return _tTotal;
    }
```

```
    function balanceOf(address account) public view override returns (uint256) {
        if (_isExcluded[account]) return _tOwned[account];
        return tokenFromReflection(_rOwned[account]);
    }

    function transfer(address recipient, uint256 amount) public override returns (bool) {
        _transfer(_msgSender(), recipient, amount);
        return true;
    }

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

    function approve(address spender, uint256 amount) public override returns (bool) {
        _approve(_msgSender(), spender, amount);
        return true;
    }

    function transferFrom(address sender, address recipient, uint256 amount) public override returns (bool) {
        _transfer(sender, recipient, amount);
        _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "ERC20: transfer
amount exceeds allowance"));
        return true;
    }

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

    function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, "ERC20:
decreased allowance below zero"));
        return true;
    }

    function isExcludedFromReward(address account) public view returns (bool) {
        return _isExcluded[account];
    }

    function totalFees() public view returns (uint256) {
        return _tFeeTotal;
    }

    function reflectionFromToken(uint256 tAmount, bool deductTransferFee) public view returns(uint256) {
        require(tAmount <= _tTotal, "Amount must be less than supply");
        if (!deductTransferFee) {
            (uint256 rAmount,,,,,) = _getValues(tAmount);
            return rAmount;
        } else {
            (,uint256 rTransferAmount,,,,) = _getValues(tAmount);
            return rTransferAmount;
        }
    }

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

    function excludeFromReward(address account) public onlyOwner() {// Millionaire Set users not to part
        require(!_isExcluded[account], "Account is already excluded");                    icipate in dividends
        if(_rOwned[account] > 0) {
            _tOwned[account] = tokenFromReflection(_rOwned[account]);
        }
        _isExcluded[account] = true;
        _excluded.push(account);
    }

    function includeInReward(address account) external onlyOwner() {// Millionaire Set up user participat
        require(_isExcluded[account], "Account is already excluded");                     ion dividend
        for (uint256 i = 0; i < _excluded.length; i++) {
            if (_excluded[i] == account) {
                _excluded[i] = _excluded[_excluded.length - 1];
                _tOwned[account] = 0;
                _isExcluded[account] = false;
                _excluded.pop();
                break;
            }
        }
    }

    function excludeFromFee(address account) public onlyOwner {
        _isExcludedFromFee[account] = true;
```

```
    }

    function includeInFee(address account) public onlyOwner {
        _isExcludedFromFee[account] = false;
    }

    function _reflectFee(uint256 rFee, uint256 tFee) private {
        _rTotal = _rTotal.sub(rFee);
        _tFeeTotal = _tFeeTotal.add(tFee);
    }

    function _getValues(uint256 tAmount) private view returns (uint256, uint256, uint256, uint256, uint256, uint256) {
        (uint256 tTransferAmount, uint256 tFee, uint256 tLiquidity) = _getTValues(tAmount);
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee) = _getRValues(tAmount, tFee, tLiquidity, _getRate());
        return (rAmount, rTransferAmount, rFee, tTransferAmount, tFee, tLiquidity);
    }

    function _getTValues(uint256 tAmount) private view returns (uint256, uint256, uint256) {
        uint256 tFee = tAmount.mul(_taxFee).div(100);
        uint256 tLiquidity = tAmount.mul(_liquidityFee).div(100);
        uint256 tTransferAmount = tAmount.sub(tFee).sub(tLiquidity);
        return (tTransferAmount, tFee, tLiquidity);
    }

    function _getRValues(uint256 tAmount, uint256 tFee, uint256 tLiquidity, uint256 currentRate) private pure returns (uint256, uint256, uint256) {
        uint256 rAmount = tAmount.mul(currentRate);
        uint256 rFee = tFee.mul(currentRate);
        uint256 rLiquidity = tLiquidity.mul(currentRate);
        uint256 rTransferAmount = rAmount.sub(rFee).sub(rLiquidity);
        return (rAmount, rTransferAmount, rFee);
    }

    function _getRate() private view returns(uint256) {
        (uint256 rSupply, uint256 tSupply) = _getCurrentSupply();
        return rSupply.div(tSupply);
    }

    function _getCurrentSupply() private view returns(uint256, uint256) {
        uint256 rSupply = _rTotal;
        uint256 tSupply = _tTotal;
        for (uint256 i = 0; i < _excluded.length; i++) {
            if (_rOwned[_excluded[i]] > rSupply || _tOwned[_excluded[i]] > tSupply) return (_rTotal, _tTotal);
            rSupply = rSupply.sub(_rOwned[_excluded[i]]);
            tSupply = tSupply.sub(_tOwned[_excluded[i]]);
        }
        if (rSupply < _rTotal.div(_tTotal)) return (_rTotal, _tTotal);
        return (rSupply, tSupply);
    }

    function _takeDividends(uint256 tDividends) private {
        uint256 currentRate = _getRate();
        uint256 rDividends = tDividends.mul(currentRate);
        _rOwned[DIVIDENDS_ADDRESS] = _rOwned[DIVIDENDS_ADDRESS].add(rDividends);
        if(_isExcluded[DIVIDENDS_ADDRESS])
            _tOwned[DIVIDENDS_ADDRESS] = _tOwned[DIVIDENDS_ADDRESS].add(tDividends);
    }

    function _takeBurn(uint256 tBurn) private {
        uint256 currentRate = _getRate();
        uint256 rBurn = tBurn.mul(currentRate);
        _rOwned[BURN_ADDRESS] = _rOwned[BURN_ADDRESS].add(rBurn);
        if(_isExcluded[BURN_ADDRESS])
            _tOwned[BURN_ADDRESS] = _tOwned[BURN_ADDRESS].add(tBurn);
    }

    function _takeLiquidity(uint256 tLiquidity) private {
        uint256 currentRate = _getRate();
        uint256 rLiquidity = tLiquidity.mul(currentRate);
        _rOwned[LP_ADDRESS] = _rOwned[LP_ADDRESS].add(rLiquidity);
        if(_isExcluded[LP_ADDRESS])
            _tOwned[LP_ADDRESS] = _tOwned[LP_ADDRESS].add(tLiquidity);
    }

    function removeAllFee() private {// Millionaire Removal cost
        if(_taxFee == 0 && _liquidityFee == 0) return;

        _previousTaxFee = _taxFee;
        _previousLiquidityFee = _liquidityFee;

        _taxFee = 0;
        _liquidityFee = 0;
    }

    function restoreAllFee() private {// Millionaire Reset cost
```

```
        _taxFee = _previousTaxFee;
        _liquidityFee = _previousLiquidityFee;
    }

    function isExcludedFromFee(address account) public view returns(bool) {
        return _isExcludedFromFee[account];
    }

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

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

    function _transfer(
        address from,
        address to,
        uint256 amount
    ) private {// Millionaire Transfer
        require(from != address(0), "TRC20: transfer from the zero address");
        require(to != address(0), "TRC20: transfer to the zero address");
        require(amount > 0, "Transfer amount must be greater than zero");

        //indicates if fee should be deducted from transfer
        bool takeFee = true;

        //if any account belongs to _isExcludedFromFee account then remove the fee
        if(_isExcludedFromFee[from] || _isExcludedFromFee[to]){// Millionaire Judge whether it is a
        whitelist, and if it is a whitelist, there is no service charge.
            takeFee = false;
        }

        //transfer amount, it will take tax, burn, liquidity fee
        _tokenTransfer(from,to,amount,takeFee);
    }

    //this method is responsible for taking all fee, if takeFee is true
    function _tokenTransfer(address sender, address recipient, uint256 amount,bool takeFee) private {// Millionaire
Actual transfer
        if(!takeFee)
            removeAllFee();
        // Millionaire Judge whether it is a dividend account and select the transfer channel
        if (_isExcluded[sender] && !_isExcluded[recipient]) {
            _transferFromExcluded(sender, recipient, amount);
        } else if (!_isExcluded[sender] && _isExcluded[recipient]) {
            _transferToExcluded(sender, recipient, amount);
        } else if (!_isExcluded[sender] && !_isExcluded[recipient]) {
            _transferStandard(sender, recipient, amount);
        } else if (_isExcluded[sender] && _isExcluded[recipient]) {
            _transferBothExcluded(sender, recipient, amount);
        } else {
            _transferStandard(sender, recipient, amount);
        }

        if(!takeFee)
            restoreAllFee();
    }

    function _transferStandard(address sender, address recipient, uint256 tAmount) private {
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransferAmount, uint256 tFee,
uint256 tLiquidity) = _getValues(tAmount);
        _rOwned[sender] = _rOwned[sender].sub(rAmount);
        _rOwned[recipient] = _rOwned[recipient].add(rTransferAmount);
        if (0 < tLiquidity) {
            _takeLiquidity(tLiquidity * _specifyFee / _liquidityFee);// Millionaire Turnover fee
            _takeDividends(tLiquidity * _dividendsFee / _liquidityFee);// Millionaire Coin holders dividends
            _takeBurn(tLiquidity * _burnFee / _liquidityFee);
        }
        _reflectFee(rFee, tFee);
        emit Transfer(sender, recipient, tTransferAmount);
    }

    function _transferToExcluded(address sender, address recipient, uint256 tAmount) private {
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransferAmount, uint256 tFee,
uint256 tLiquidity) = _getValues(tAmount);
        _rOwned[sender] = _rOwned[sender].sub(rAmount);
        _tOwned[recipient] = _tOwned[recipient].add(tTransferAmount);
        _rOwned[recipient] = _rOwned[recipient].add(rTransferAmount);
        if (0 < tLiquidity) {
            _takeLiquidity(tLiquidity * _specifyFee / _liquidityFee);
            _takeDividends(tLiquidity * _dividendsFee / _liquidityFee);
            _takeBurn(tLiquidity * _burnFee / _liquidityFee);
        }
        _reflectFee(rFee, tFee);
        emit Transfer(sender, recipient, tTransferAmount);
```

```
    }

    function _transferFromExcluded(address sender, address recipient, uint256 tAmount) private {
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransferAmount, uint256 tFee,
uint256 tLiquidity) = _getValues(tAmount);
        _tOwned[sender] = _tOwned[sender].sub(tAmount);
        _rOwned[sender] = _rOwned[sender].sub(rAmount);
        _rOwned[recipient] = _rOwned[recipient].add(rTransferAmount);
        if (0 < tLiquidity) {
            _takeLiquidity(tLiquidity * _specifyFee / _liquidityFee);
            _takeDividends(tLiquidity * _dividendsFee / _liquidityFee);
            _takeBurn(tLiquidity * _burnFee / _liquidityFee);
        }
        _reflectFee(rFee, tFee);
        emit Transfer(sender, recipient, tTransferAmount);
    }

    function _transferBothExcluded(address sender, address recipient, uint256 tAmount) private {
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256 tTransferAmount, uint256 tFee,
uint256 tLiquidity) = _getValues(tAmount);
        _tOwned[sender] = _tOwned[sender].sub(tAmount);
        _rOwned[sender] = _rOwned[sender].sub(rAmount);
        _tOwned[recipient] = _tOwned[recipient].add(tTransferAmount);
        _rOwned[recipient] = _rOwned[recipient].add(rTransferAmount);
        if (0 < tLiquidity) {
            _takeLiquidity(tLiquidity * _specifyFee / _liquidityFee);
            _takeDividends(tLiquidity * _dividendsFee / _liquidityFee);
            _takeBurn(tLiquidity * _burnFee / _liquidityFee);
        }
        _reflectFee(rFee, tFee);
        emit Transfer(sender, recipient, tTransferAmount);
    }

}
```

6. Appendix B: Safety risk rating standard

Intelligent contract vulnerability rating standard	
Rating	Vulnerability rating description
High risk loophole	Loopholes that can directly cause losses of token contracts or users' funds, such as numerical overflow loopholes that can cause tokens' value to zero, fake recharge loopholes that can cause exchange losses, loopholes that can cause contract account losses or token reentry loopholes, etc. Loopholes that can cause the loss of ownership of token contracts, such as access control defects of key functions, access control bypass of key functions caused by call injection, etc. Loopholes that can cause token contracts not to work properly, such as a denial of service vulnerability caused by sending TRX to a malicious address, and a denial of service vulnerability caused by gas exhaustion.
Medium risk loophole	High-risk vulnerabilities that need a specific address to trigger, such as numerical overflow vulnerabilities that can only be triggered by the owner of a token contract, access control defects of non-critical functions, logical design defects that can not cause direct capital losses, and so on.
Low risk loophole	Vulnerabilities that are difficult to trigger, vulnerabilities with limited harm after triggering, such as numerical overflow vulnerabilities that require a large number of TRX or tokens to trigger, vulnerabilities that attackers cannot directly profit after triggering numerical overflows, transaction sequence dependency risks triggered by specifying high gas, and so on.

7. Appendix C: Brief introduction of Intelligent contract Security Audit tool

7.1. Manticore

Manticore is a symbol execution tool for analyzing binaries and smart contracts. Manticore includes a symbolic Ethernet Square Virtual Machine (EVM), an EVM disassembler / assembler, and a convenient interface for automatically compiling and analyzing Solidity.

It also integrates Ethersplay, a Bit of Traits of Bits visual disassembler for EVM bytecode, for visual analysis.
Like binaries, Manticore provides a simple command-line interface and a Python API for parsing EVM bytecode.

7.2. Oyente

Oyente is an intelligent contract analysis tool. Oyente can be used to detect common bug in smart contracts, such as reentrancy, transaction ordering dependency, and so on.

More conveniently, the design of Oyente is modular, so this allows advanced users to implement and insert their own detection logic to check custom attributes in their contracts.

7.3. securify.sh

Securify you can verify common security issues in ethersquare smart contracts, such as transaction disorder and lack of input validation, which analyzes all possible execution paths of the program while fully automated. In addition.

Securify has a specific language for specifying vulnerabilities, which allows Securify to pay attention to current security and other reliability issues at any time.

7.4. Echidna

Echidna is a Haskell library designed for fuzzy testing of EVM code.

7.5. MAIAN

MAIAN is an automated tool for finding vulnerabilities in smart contracts.

Maian processes the bytecode of contracts and attempts to establish a series of transactions to identify and confirm errors.

7.6. ethersplay

Ethersplay is an EVM disassembler that contains related analysis tools.

7.7. ida-evm

Ida-evm is an IDA processor module for virtual machines (EVM).

7.8. Remix-ide

Remix is a browser-based compiler and IDE that allows users to build contracts and debug transactions using the Solidity language.

7.9. Know the special tool kit for Chuangyu block chain security auditors

Know that the special tool kit for penetration testers is developed, collected and used by Chuangyu penetration testing engineers, including batch automatic testing tools for testers, self-developed tools, scripts or utilization tools, etc.

Know Chuangyu

Know Chuangyu Information Technology Co Ltd

Email： sec@kownsec.com

website： www.knownsec.com