Smart contract audit report

CTH

CHAINLION

NO. OC002208140001

AUGUST 14, 2022

CHAINLION

CATALOGUE

1. PROJECT SUMMARY 1

2. AUDIT SUMMARY 1

3. VULNERABILITY SUMMARY 1

4. EXECUTIVE SUMMARY 3

5. DIRECTORY STRUCTURE 4

6. FILE HASHES 4

7. VULNERABILITY DISTRIBUTION 4

8. AUDIT CONTENT 6

8.1. CODING SPECIFICATION 6

8.1.1. Compiler Version【security】 6

8.1.2. Return value verification【security】 7

8.1.3. Constructor writing【security】 8

8.1.4. Key event trigger【security】 9

8.1.5. Address non-zero check【securityk】 9

8.1.6. Code redundancy check【security】 9

8.2. CODING DESIGN 10

8.2.1. Shaping overflow detection【security】 10

8.2.2. Reentry detection【security】 14

8.2.3. Rearrangement attack detection【security】 14

8.2.4. Replay Attack Detection【security】 15

8.2.5. False recharge detection【security】 15

8.2.6. Access control detection【security】 16

8.2.7. Denial of service detection【security】 17

8.2.8. Conditional competition detection【security】 17

8.2.9. Consistency detection【security】 18

8.2.10. Variable coverage detection【security】 19

8.2.11. Random number detection【security】 19

8.2.12. Numerical operation detection【security】 20

8.2.13. Call injection detection【security】 20

8.3. BUSINESS LOGIC 21

8.3.1. Constructor initialization logic【security】 21

8.3.2. Contract basic information query function【security】 23

8.3.3. Transfer logic design【security】 25

8.3.4. Logic design of token destruction【security】 26

8.3.5. Authorization related logic【security】 28

8.3.6. Transferfrom transfer logic design【security】 30

8.3.7. _ Burnfrom business logic design【security】 32

8.3.8. Contract authority concentration【security】 33

9. Contract source code 35

10. APPENDIX:ANALYSIS TOOLS 50

10.1. SOLGRAPH 50

10.2. SOL2UML 50

10.3. REMIX-IDE 50

10.4. ETHERSPLAY 50

10.5. MYTHRIL 50

10.6. ECHIDNA 51

11. DISCLAIMERS 51

1. PROJECT SUMMARY

Entry type	Specific description
Entry name	CTH
Project type	DEFI
Application platform	TRON
DawnToken	TJmki9vmWXP6QxHJ9JVCd5iyAs8xwS5HL4

2. AUDIT SUMMARY

Entry type	Specific description
Project cycle	AUGUST/11/2022-AUGUST/14/2022
Audit method	Black box test、White box test、Grey box test
Auditors	Two

3. VULNERABILITY SUMMARY

Audit results are as follows:

Entry type	Specific description
Serious vulnerability	0
High risk vulnerability	0
Moderate risk	0

Low risk vulnerability	0

Security vulnerability rating description:

1) **Serious vulnerability:** Security vulnerabilities that can directly cause token contracts or user capital losses, For example: shaping overflow vulnerability、Fake recharge vulnerability、Reentry attacks, vulnerabilities, etc.

2) **High risk vulnerability:** Security vulnerabilities that can directly cause the contract to fail to work normally, such as reconstructed smart contract caused by constructor design error, denial of service vulnerability caused by unreasonable design of require / assert detection conditions, etc.

3) **Moderate risk:** Security problems caused by unreasonable business logic design, such as accuracy problems caused by unreasonable numerical operation sequence design, variable ambiguous naming, variable coverage, call injection, conditional competition, etc.

4) **Low risk vulnerability:** Security vulnerabilities that can only be triggered by users with special permissions, such as contract backdoor vulnerability, duplicate name pool addition vulnerability, non-standard contract coding, contract detection bypass, lack of necessary events for key state variable change, and security vulnerabilities that are harmful in theory but have harsh utilization

conditions.

4. EXECUTIVE SUMMARY

This report is prepared for **CTH** smart contract , The purpose is to find the security vulnerabilities and non-standard coding problems in the smart contract through the security audit of the source code of the smart contract. This audit mainly involves the following test methods:

White box test

Conduct security audit on the source code of smart contract and check the security issues such as coding specification, DASP top 10 and business logic design

Grey box test

Deploy smart contracts locally and conduct fuzzy testing to check function robustness, function call permission and business logic security

Black box test

Conduct security test attacks on smart contracts from the perspective of attackers, combined with black-and-white and testing techniques, to check whether there are exploitable vulnerabilities.

This audit report is subject to the latest contract code provided by the current project party, does not include the newly added business logic function module after the contract upgrade, does not include new attack methods in the future, and does not include web front-end security and server-side security.

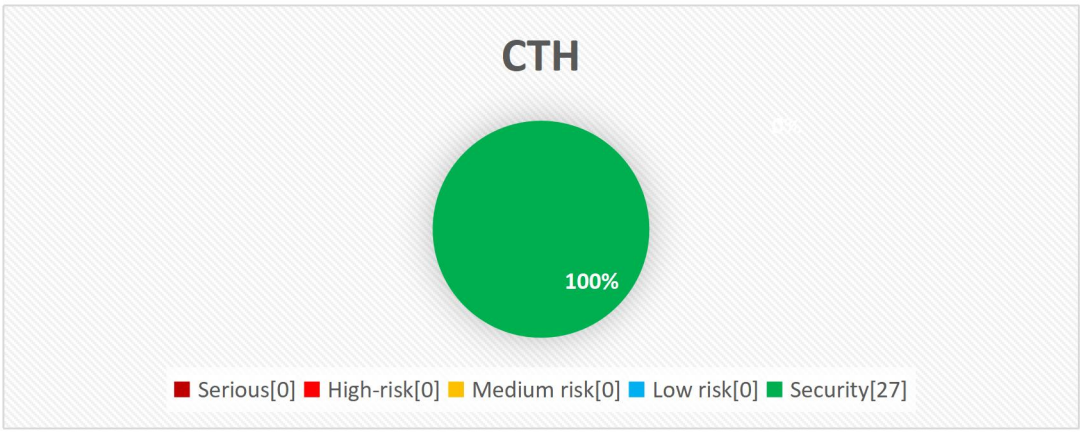
5. Directory structure

```
ERC20.sol
ERC20Detailed.sol
IERC20.sol
SafeMath.sol
Token.sol
```

6. File hashes

Contract	SHA1 Checksum
ERC20.sol	4067F8D642EAEEAFEC33EA53E8317F8CB4176330
ERC20Detailed.sol	1DBE74B054C234713E80D071F57FB93C821D9709
IERC20.sol	0A5214ECF95B707ACD6AE9C517C693EAD0B6DCAA
SafeMath.sol	03BF3B948D0A61410E7C9C205ABBECF7F3EB05AE
Token.sol	5EB5F0EF8E10060016CC200AAAC9EC5B7820A685

7. Vulnerability distribution


CTH
100%
Serious[0]
High-risk[0]
Medium risk[0]
Low risk[0]
Security[27]

8. Audit content

8.1. Coding specification

Smart contract supports contract development in programming languages such as solid, Vyper, C + +, Python and rust. Each programming language has its own coding specification. In the development process, the coding specification of the development language should be strictly followed to avoid security problems such as business function design defects.

8.1.1. Compiler Version 【security】

Audit description： The compiler version should be specified in the smart contract code. At the same time, it is recommended to use the latest compiler version. The old version of the compiler may cause various known security problems. At present, the latest version is v 0.8 x. And this version has been protected against shaping overflow.

Audit results： According to the audit, the compiler version used in the smart contract code is 0.5.0, and there is no low version security risk.

```
pragma solidity ^0.5.0;

import "./IERC20.sol";

/**
 * @dev Optional functions from the ERC20 standard.
 */
contract ERC20Detailed is IERC20 {
    string private _name;
    string private _symbol;
    uint8 private _decimals;

    /**
     * @dev Sets the values for `name`, `symbol`, and `decimals`. All three of
     * these values are immutable: they can only be set once during
     * construction.
     */
    constructor (string memory name, string memory symbol, uint8 decimals) public {
        _name = name;
        _symbol = symbol;
        _decimals = decimals;
    }
```

Safety advice：NONE.

8.1.2. Return value verification 【security】

Audit description： Smart contract requires contract developers to strictly follow EIP / tip and other standards and specifications during contract development. For transfer, transferfrom and approve functions, Boolean values should be returned to feed back the final execution results. In the smart contract, the relevant business logic code often calls the transfer or transferfrom function to transfer. In this case, the return value involved in the transfer operation should be strictly checked to determine whether the transfer is successful or not, so as to avoid security vulnerabilities such as false recharge caused by the lack of return value verification.

Audit results： According to the audit, there is no embedded function calling the

official standards transfer and transferfrom in the smart contract, so there is no such security problem.

Safety advice：NONE.

8.1.3. Constructor writing【security】

Audit description： In solid v0 The smart contract written by solidity before version 4.22 requires that the constructor must be consistent with the contract name. When the constructor name is inconsistent with the contract name, the constructor will become an ordinary public function. Any user can call the constructor to initialize the contract. After version V 0.4.22, The constructor name can be replaced by constructor, so as to avoid the coding problems caused by constructor writing.

Audit results： After audit, the constructor in the smart contract is written correctly, and there is no such security problem.

```
8   contract ERC20Detailed is IERC20 {
9       string private _name;
10      string private _symbol;
11      uint8 private _decimals;
12
13      /**
14       * @dev Sets the values for `name`, `symbol`, and `decimals`. All three of
15       * these values are immutable: they can only be set once during
16       * construction.
17       */
18      constructor (string memory name, string memory symbol, uint8 decimals) public {
19          _name = name;
20          _symbol = symbol;
21          _decimals = decimals;
22      }
23
```

Safety advice：NONE.

8.1.4. Key event trigger【security】

Audit description： Most of the key global variable initialization or update operations similar to setXXX exist in the smart contract. It is recommended to trigger the corresponding event through emit when operating on similar key events.

Audit results： Through audit, there is no such security problem.

Safety advice：NONE.

8.1.5. Address non-zero check【security】

Audit description： The smart contract initializes the key information of the contract through the constructor. When it comes to address initialization, the address should be non-zero checked to avoid irreparable economic losses.

Audit results： Through audit, there is no such security problem.

Safety advice：NONE.

8.1.6. Code redundancy check【security】

Audit description： The deployment and execution of smart contracts need to consume certain gas costs. The business logic design should be optimized as much as possible, while avoiding unnecessary redundant code to improve efficiency and save costs.

Audit results： After audit, there is no such security problem.

Safety advice：NONE.

8.2. Coding design

DASP top 10 summarizes the common security vulnerabilities of smart contracts. Smart contract developers can study smart contract security vulnerabilities before developing contracts to avoid security vulnerabilities during contract development. Contract auditors can quickly audit and check the existing security vulnerabilities of smart contracts according to DASP top 10.

8.2.1. Shaping overflow detection 【security】

Audit description： Solid can handle 256 digits at most. When the number is unsigned, the maximum value will overflow by 1 to get 0, and 0 minus 1 will overflow to get the maximum value. The problem of shaping overflow often appears in the relevant logic code design function modules such as transaction transfer, reward calculation and expense calculation. The security problems caused by shaping overflow are also very serious, such as excessive coinage, high sales and low income, excessive distribution, etc. the problem of shaping overflow can be solved by using solid V 0.8 X version or by using the safemath library officially provided by openzenppelin.

Audit results： According to the audit, the smart contract uses safemath library, which can better prevent the shaping overflow problem caused by numerical operation.

```
pragma solidity ^0.5.0;

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
```

```
}

/**
 * @dev Returns the subtraction of two unsigned integers, reverting on
 * overflow (when the result is negative).
 *
 * Counterpart to Solidity's `-` operator.
 *
 * Requirements:
 * - Subtraction cannot overflow.
 */
function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b <= a, "SafeMath: subtraction overflow");
    uint256 c = a - b;

    return c;
}

/**
 * @dev Returns the multiplication of two unsigned integers, reverting on
 * overflow.
 *
 * Counterpart to Solidity's `*` operator.
 *
 * Requirements:
 * - Multiplication cannot overflow.
 */
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
    // benefit is lost if 'b' is also tested.
    // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
    if (a == 0) {
        return 0;
    }

    uint256 c = a * b;
```

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

        return c;
    }

    /**
     * @dev Returns the integer division of two unsigned integers. Reverts on
     * division by zero. The result is rounded towards zero.
     *
     * Counterpart to Solidity's `/` operator. Note: this function uses a
     * `revert` opcode (which leaves remaining gas untouched) while Solidity
     * uses an invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
        require(b > 0, "SafeMath: division by zero");
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
```

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

Safety advice：NONE.

8.2.2. Reentry detection【security】

Audit description： The in solidity provides call Value(), send(), transfer() and other functions are used for transfer operation. When call When value() sends ether, it will send all gas for transfer operation by default. If the transfer function can be called recursively again through call transfer, it can cause reentry attack.

Audit results： After audit, there is no such security problem.

Safety advice：NONE.

8.2.3. Rearrangement attack detection【security】

Audit description： Rearrangement attack means that miners or other parties try to compete with smart contract participants by inserting their information into the list or mapping, so that attackers have the opportunity to store their information in the contract.

Audit results： After audit, there is no such security problem.

Safety advice：NONE.

8.2.4. Replay Attack Detection 【security】

Audit description： When the contract involves the business logic of delegated management, attention should be paid to the non reusability of verification to avoid replay attacks. In common asset management systems, there are often delegated management businesses. The principal gives the assets to the trustee for management, and the principal pays a certain fee to the trustee. In similar delegated management scenarios, it is necessary to ensure that the verification information will become invalid once used.

Audit results： After audit, there is no such security problem.

Safety advice： NONE.

8.2.5. False recharge detection 【security】

Audit description：When a smart contract uses the transfer function for transfer, it should use require / assert to strictly check the transfer conditions. It is not recommended to use if Use mild judgment methods such as else to check, otherwise it will misjudge the success of the transaction, resulting in the security problem of false recharge.

Audit results： After audit, there is no such security problem.

Safety advice： NONE.

8.2.6. Access control detection 【security】

Audit description： Solid provides four function access domain Keywords: public, private, external and internal to limit the scope of function. In the smart contract, the scope of function should be reasonably designed to avoid the security risk of improper access control. The main differences of the above four keywords are as follows:

1．public： The marked function or variable can be called or obtained by any account, which can be a function in the contract, an external user or inherit the function in the contract

2．external： The marked functions can only be accessed from the outside and cannot be called directly by the functions in the contract, but this can be used Func() calls this function as an external call

3．private： Marked functions or variables can only be used in this contract (Note: the limitation here is only at the code level. Ethereum is a public chain, and anyone can directly obtain the contract status information from the chain)

4．internal: It is generally used in contract inheritance. The parent contract is marked as an internal state variable or function, which can be directly accessed and called by the child contract (it cannot be directly obtained and called externally)

Audit results： After audit, there is no such security problem.

Safety advice：NONE.

8.2.7. Denial of service detection 【security】

Audit description：Denial of service attack is a DoS attack on Ethereum contract, which makes ether or gas consume a lot. In more serious cases, it can make the contract code logic unable to operate normally. The common causes of DoS attack are: unreasonable design of require check condition, uncontrollable number of for cycles, defects in business logic design, etc.

Audit results： After audit, there is no such security problem.

Safety advice：NONE.

8.2.8. Conditional competition detection 【security】

Audit description： The Ethereum node gathers transactions and forms them into blocks. Once the miners solve the consensus problem, these transactions are considered effective. The miners who solve the block will also choose which transactions from the mine pool will be included in the block. This is usually determined by gasprice transactions. Attackers can observe whether there are

transactions in the transaction pool that may contain problem solutions, After that, the attacker can obtain data from this transaction, create a higher-level transaction gasprice, and include its transaction in a block before the original, so as to seize the original solution.

Audit results： After audit, there is no such security problem.

Safety advice：NONE.

8.2.9. Consistency detection 【security】

Audit description： The update logic in smart contract (such as token quantity update, authorized transfer quota update, etc.) is often accompanied by the check logic of the operation object (such as anti overflow check, authorized transfer quota check, etc.), and when the update object is inconsistent with the check object, the check operation may be invalid, Thus, the conditional check logic is ignored and unexpected logic is executed. For example, the authorized transfer function function function transfer from (address _from, address _to, uint256 _value) returns (bool success) is used to authorize others to transfer on behalf of others. During transfer, the permission [_from] [MSG. Sender] authorized transfer limit will be checked, After passing the check, the authorized transfer limit will be updated at the same time of

transfer. When the update object in the update logic is inconsistent with the check object in the check logic, the authorized transfer limit of the authorized transfer user will not change, resulting in that the authorized transfer user can transfer all the assets of the authorized account.

Audit results： After audit, there is no such security problem.

Safety advice：NONE.

8.2.10. Variable coverage detection 【security】

Audit description： Smart contracts allow inheritance relationships, in which the child contract inherits all the methods and variables of the parent contract. If a global variable with the same name as the parent contract is defined in the child contract, it may lead to variable coverage and corresponding asset losses.

Audit results： After audit, there is no such security problem.

Safety advice：NONE.

8.2.11. Random number detection 【security】

Audit description： Random numbers are often used in smart contracts. When designing the random number generation function, the generation and selection of

random seeds should avoid the data information that can be queried on the blockchain, such as block Number and block Timestamp et al. These data are vulnerable to the influence of miners, resulting in the predictability of random numbers to a certain extent.

Audit results： After audit, there is no such security problem.

Safety advice：NONE.

8.2.12. Numerical operation detection 【security】

Audit description： Solidity supports addition, subtraction, multiplication, division and other conventional numerical operations, but solidty does not support floating-point types. When multiplication and division operations exist at the same time, the numerical operation order should be adjusted reasonably to reduce the error as much as possible.

Audit results： After audit, there is no such security problem.

Safety advice：NONE.

8.2.13. Call injection detection 【security】

Audit description： In the solid language, you can call a contract or a method of a

local contract through the call method. There are roughly two ways to call: < address > Call (method selector, arg1, arg2,...) or < address > Call (bytes). When using call call, we can pass method selectors and parameters by passing parameters, or directly pass in a byte array. Based on this function, it is recommended that strict permission check or hard code the function called by call when using call function call.

Audit results： After audit, there is no such security problem.

Safety advice：NONE.

8.3. Business logic

Business logic design is the core of smart contract. When using programming language to develop contract business logic functions, developers should fully consider all aspects of the corresponding business, such as parameter legitimacy check, business permission design, business execution conditions, interaction design between businesses, etc.

8.3.1. Constructor initialization logic 【security】

Audit description： Conduct security audit on the constructor initialization and business logic design in the contract, and check whether the initialization value is consistent with the requirements document.

Audit results： The constructor initialization business logic design in the contract

is correct, and no relevant security risks are found.

Code file： Token.sol L19~21　ERC20Detailed.sol L18~22

Code information：

```
contract Token is ERC20, ERC20Detailed {

    /**
     * @dev Constructor that gives msg.sender all of existing tokens.
     */
    constructor () public ERC20Detailed("Payment chain", "CTH", 6) {
        _mint(msg.sender, 210000000 * (10 ** uint256(decimals()))); //Initialize the issuance of additional tokens
    }
}
    /** @dev Creates `amount` tokens and assigns them to `account`, increasing
     * the total supply.
     *
     * Emits a {Transfer} event with `from` set to the zero address.
     *
     * Requirements
     *
     * - `to` cannot be the zero address.
     */
    function _mint(address account, uint256 amount) internal {
        require(account != address(0), "ERC20: mint to the zero address"); //Address non-zero check

        _totalSupply = _totalSupply.add(amount);    //Additional tokens
        _balances[account] = _balances[account].add(amount); //Update asset quantity
        emit Transfer(address(0), account, amount);
    }

    /**
     * @dev Returns the number of decimals used to get its user representation.
     * For example, if `decimals` equals `2`, a balance of `505` tokens should
     * be displayed to a user as `5,05` (`505 / 10 ** 2`).
     *
```

```
 * Tokens usually opt for a value of 18, imitating the relationship between
 * Ether and Wei.
 *
 * NOTE: This information is only used for _display_ purposes: it in
 * no way affects any of the arithmetic of the contract, including
 * {IERC20-balanceOf} and {IERC20-transfer}.
 */
function decimals() public view returns (uint8) {
      return _decimals; //accuracy
}
string private _name;
string private _symbol;
uint8 private _decimals;

/**
 * @dev Sets the values for `name`, `symbol`, and `decimals`. All three of
 * these values are immutable: they can only be set once during
 * construction.
 */
constructor (string memory name, string memory symbol, uint8 decimals) public {
      _name = name;    //Token name
      _symbol = symbol;    //Token
      _decimals = decimals;    //Token accuracy
}
```

Safety advice：NONE.

8.3.2. Contract basic information query function 【security】

Audit description： Conduct code audit on the functions related to the query of contract basic information in the contract, and check whether the relevant business logic design is reasonable.

Audit results： The design of relevant functions for querying the basic contract

information in the contract is correct, and no relevant security risks are found.

Code file： ERC20Detailed.sol L27~54

Code information：

```
/**
 * @dev Returns the name of the token.
 */
function name() public view returns (string memory) {
    return _name;    //Token name
}

/**
 * @dev Returns the symbol of the token, usually a shorter version of the
 * name.
 */
function symbol() public view returns (string memory) {
    return _symbol;    //Token symbol
}

/**
 * @dev Returns the number of decimals used to get its user representation.
 * For example, if `decimals` equals `2`, a balance of `505` tokens should
 * be displayed to a user as `5,05` (`505 / 10 ** 2`).
 *
 * Tokens usually opt for a value of 18, imitating the relationship between
 * Ether and Wei.
 *
 * NOTE: This information is only used for _display_ purposes: it in
 * no way affects any of the arithmetic of the contract, including
 * {IERC20-balanceOf} and {IERC20-transfer}.
 */
function decimals() public view returns (uint8) {
    return _decimals;    //Token accuracy
}
/**
 * @dev See {IERC20-totalSupply}.
```

```
 */
function totalSupply() public view returns (uint256) {
        return _totalSupply;    //Total token amount query
}

/**
 * @dev See {IERC20-balanceOf}.
 */
function balanceOf(address account) public view returns (uint256) {
        return _balances[account];      //Number of specified address assets
}
```

Safety advice： NONE.

8.3.3. Transfer logic design 【security】

Audit results： Conduct code audit on the transfer function transfer in the contract to check whether the relevant business logic design is reasonable.

Audit results： The transfer function in the contract is designed correctly, and no relevant security risks are found.

Code file： ECR20.sol L61~64

Code information：

```
/**
 * @dev See {IERC20-transfer}.
 *
 * Requirements:
 *
 * - `recipient` cannot be the zero address.
 * - the caller must have a balance of at least `amount`.
 */
function transfer(address recipient, uint256 amount) public returns (bool) {
        _transfer(msg.sender, recipient, amount); //Invoke_ Transfer transfer
        return true;
```

```
    }
    /**
     * @dev Moves tokens `amount` from `sender` to `recipient`.
     *
     * This is internal function is equivalent to {transfer}, and can be used to
     * e.g. implement automatic token fees, slashing mechanisms, etc.
     *
     * Emits a {Transfer} event.
     *
     * Requirements:
     *
     * - `sender` cannot be the zero address.
     * - `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     */
    function _transfer(address sender, address recipient, uint256 amount) internal {     //Function internal call
        require(sender != address(0), "ERC20: transfer from the zero address");    //Address non-zero check
        require(recipient != address(0), "ERC20: transfer to the zero address");    //Address non-zero check

        _balances[sender] = _balances[sender].sub(amount);    //Update the number of sender assets
        _balances[recipient] = _balances[recipient].add(amount);    //Update the number of recipient assets
        emit Transfer(sender, recipient, amount);
    }
```

Safety advice：NONE.

8.3.4. Logic design of token destruction business 【security】

Audit description：Conduct code audit on the token destruction function brun in the contract to check whether the relevant business logic design is reasonable.

Audit results： The token destruction function burn in the contract is designed

correctly, and no relevant security risks are found.

Code file： ECR20.sol L65~68

Code information：

```
    function burn(uint256 amount) public returns (bool) {
        _burn(msg.sender, amount);    //Call the burn function to destroy tokens, and you can only destroy your own tokens
            return true;
        }

        /**
         * @dev Destroys `amount` tokens from `account`, reducing the
         * total supply.
         *
         * Emits a {Transfer} event with `to` set to the zero address.
         *
         * Requirements
         *
         * - `account` cannot be the zero address.
         * - `account` must have at least `amount` tokens.
         */
        function _burn(address account, uint256 value) internal {     //Can only be called inside the contract
            require(account != address(0), "ERC20: burn from the zero address");    //Address non-zero check

            _totalSupply = _totalSupply.sub(value);    //Update total tokens
            _balances[account] = _balances[account].sub(value);    //Update the number of assets held by the account address
            emit Transfer(account, address(0), value);
        }
```

Safety advice： NONE.

8.3.5. Authorization related logic 【security】

Audit description： Conduct code audit on the logic related to authorized transfer in the contract and check whether the design of relevant business logic is reasonable.

Audit results： The logic design related to authorized transfer in the contract is correct, and no relevant security risk is found.

Code file： ECR20.sol L74~89

Code information：

```
/**
 * @dev See {IERC20-allowance}.
 */
function allowance(address owner, address spender) public view returns (uint256) {
    return _allowances[owner][spender];    //Query authorization limit
}
/**
 * @dev See {IERC20-approve}.
 *
 * Requirements:
 *
 * - `spender` cannot be the zero address.
 */
function approve(address spender, uint256 value) public returns (bool) {
    _approve(msg.sender, spender, value);    //Update authorization limit
    return true;
}
/**
 * @dev Sets `amount` as the allowance of `spender` over the `owner`s tokens.
 *
```

```
     * This is internal function is equivalent to `approve`, and can be used to
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     */
    function _approve(address owner, address spender, uint256 value) internal {    //Contract internal call
        require(owner != address(0), "ERC20: approve from the zero address");    //Address non-zero check
        require(spender != address(0), "ERC20: approve to the zero address");    //Address non-zero check

        _allowances[owner][spender] = value;    //Although there is a risk of conditional competition in updating the authorization limit, the utilization conditions are extremely harsh and can be directly ignored
        emit Approval(owner, spender, value);
    }

    /**
     * @dev Atomically increases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {IERC20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
```

```
            _approve(msg.sender,      spender,      _allowances[msg.sender][spender].add(addedValue));
//Increase authorization limit
            return true;
        }

        /**
         * @dev Atomically decreases the allowance granted to `spender` by the caller.
         *
         * This is an alternative to {approve} that can be used as a mitigation for
         * problems described in {IERC20-approve}.
         *
         * Emits an {Approval} event indicating the updated allowance.
         *
         * Requirements:
         *
         * - `spender` cannot be the zero address.
         * - `spender` must have allowance for the caller of at least
         * `subtractedValue`.
         */
        function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
            _approve(msg.sender,    spender,    _allowances[msg.sender][spender].sub(subtractedValue));
//Reduce authorization limit
            return true;
            }
```

Safety advice：NONE.

8.3.6. Transferfrom transfer logic design【security】

Audit description： Conduct code audit on the logic related to authorized transfer in the contract and check whether the design of relevant business logic is reasonable.

Audit results： The logic design related to authorized transfer in the contract is

correct, and no relevant security risk is found.

Code file： ECR20.sol L102~106

Code information：

```
    /**
     * @dev See {IERC20-transferFrom}.
     *
     * Emits an {Approval} event indicating the updated allowance. This is not
     * required by the EIP. See the note at the beginning of {ERC20};
     *
     * Requirements:
     * - `sender` and `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `value`.
     * - the caller must have allowance for `sender`'s tokens of at least
     * `amount`.
     */
    function transferFrom(address sender, address recipient, uint256 amount) public returns (bool) {
        _transfer(sender, recipient, amount);    //Transfer operation
        _approve(sender,  msg.sender,  _allowances[sender][msg.sender].sub(amount));      //Update
authorization limit
        return true;
    }
    /**
     * @dev Moves tokens `amount` from `sender` to `recipient`.
     *
     * This is internal function is equivalent to {transfer}, and can be used to
     * e.g. implement automatic token fees, slashing mechanisms, etc.
     *
     * Emits a {Transfer} event.
     *
     * Requirements:
     *
     * - `sender` cannot be the zero address.
     * - `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     */
```

```
    function _transfer(address sender, address recipient, uint256 amount) internal {     //Function internal call
        require(sender != address(0), "ERC20: transfer from the zero address");    //Address non-zero check
        require(recipient != address(0), "ERC20: transfer to the zero address");    //Address non-zero check

        _balances[sender] = _balances[sender].sub(amount);    //Update the number of sender assets
        _balances[recipient] = _balances[recipient].add(amount);    //Update the number of recipient assets
        emit Transfer(sender, recipient, amount);

    }
```

Safety advice：NONE.

8.3.7. _ Burnfrom business logic design【security】

Audit description： Conduct code audit on the logic related to authorized transfer in the contract and check whether the design of relevant business logic is reasonable.

Audit results： The logic design related to authorized transfer in the contract is correct, and no relevant security risk is found.

Code file： ECR20.sol L233~236

Code information：

```
 /**
     * @dev Destoys `amount` tokens from `account`.`amount` is then deducted
     * from the caller's allowance.
     *
     * See {_burn} and {_approve}.
     */
    function _burnFrom(address account, uint256 amount) internal {    //Only the internal call of the
```

```
contract is allowed, but the whole contract is not called, and there is redundancy. However, it is hereby reserved to comply with erc20 standard
        _burn(account, amount);    //Token destruction
        _approve(account, msg.sender, _allowances[account][msg.sender].sub(amount));    //Update authorization limit
    }
```

Safety advice：NONE.

8.3.8. Contract authority concentration detection【security】

Audit description： Detect the concentration of authority in the contract and check whether the relevant business logic is reasonable.

Audit results： There is no relevant owner high permission Operation in the contract.

Code file：

Code information：

```
/** @dev Creates `amount` tokens and assigns them to `account`, increasing
     * the total supply.
     *
     * Emits a {Transfer} event with `from` set to the zero address.
     *
     * Requirements
     *
     * - `to` cannot be the zero address.
     */
    function _mint(address account, uint256 amount) internal { //Only contract internal calls
        require(account != address(0), "ERC20: mint to the zero address"); //Address non-zero check

        _totalSupply = _totalSupply.add(amount);    //Additional tokens
        _balances[account] = _balances[account].add(amount); //Update asset quantity
```

```
        emit Transfer(address(0), account, amount);
    }



    /**
     * @dev Destroys `amount` tokens from `account`, reducing the
     * total supply.
     *
     * Emits a {Transfer} event with `to` set to the zero address.
     *
     * Requirements
     *
     * - `account` cannot be the zero address.
     * - `account` must have at least `amount` tokens.
     */
    function _burn(address account, uint256 value) internal {//Only contract internal calls
        require(account != address(0), "ERC20: burn from the zero address");    //Address non-zero check

        _totalSupply = _totalSupply.sub(value);    //Update total tokens
        _balances[account] = _balances[account].sub(value);    //Update the number of assets held by the account address
        emit Transfer(account, address(0), value);
    }
    /**
     * @dev Destoys `amount` tokens from `account`.`amount` is then deducted
     * from the caller's allowance.
     *
     * See {_burn} and {_approve}.
     */
    function _burnFrom(address account, uint256 amount) internal {    //Only the internal call of the contract is allowed, but the whole contract is not called, and there is redundancy. However, it is hereby reserved to comply with erc20 standard
        _burn(account, amount);    //Token destruction
        _approve(account, msg.sender, _allowances[account][msg.sender].sub(amount));    //Update
```

```
authorization limit
    }
```

Safety advice：：NONE.

9. Contract source code

ERC20.sol

```
pragma solidity ^0.5.0;

import "./IERC20.sol";
import "./SafeMath.sol";

/**
 * @dev Implementation of the {IERC20} interface.
 *
 * This implementation is agnostic to the way tokens are created. This means
 * that a supply mechanism has to be added in a derived contract using {_mint}.
 * For a generic mechanism see {ERC20Mintable}.
 *
 * TIP: For a detailed writeup see our guide
 * https://forum.zeppelin.solutions/t/how-to-implement-erc20-supply-mechanisms/226[How
 * to implement supply mechanisms].
 *
 * We have followed general OpenZeppelin guidelines: functions revert instead
 * of returning `false` on failure. This behavior is nonetheless conventional
 * and does not conflict with the expectations of ERC20 applications.
 *
 * Additionally, an {Approval} event is emitted on calls to {transferFrom}.
 * This allows applications to reconstruct the allowance for all accounts just
 * by listening to said events. Other implementations of the EIP may not emit
 * these events, as it isn't required by the specification.
 *
 * Finally, the non-standard {decreaseAllowance} and {increaseAllowance}
 * functions have been added to mitigate the well-known issues around setting
 * allowances. See {IERC20-approve}.
```

```
 */
contract ERC20 is IERC20 {
    using SafeMath for uint256;

    mapping (address => uint256) private _balances;

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

    uint256 private _totalSupply;

    /**
     * @dev See {IERC20-totalSupply}.
     */
    function totalSupply() public view returns (uint256) {
        return _totalSupply;
    }

    /**
     * @dev See {IERC20-balanceOf}.
     */
    function balanceOf(address account) public view returns (uint256) {
        return _balances[account];
    }

    /**
     * @dev See {IERC20-transfer}.
     *
     * Requirements:
     *
     * - `recipient` cannot be the zero address.
     * - the caller must have a balance of at least `amount`.
     */
    function transfer(address recipient, uint256 amount) public returns (bool) {
        _transfer(msg.sender, recipient, amount);
        return true;
    }
```

```
function burn(uint256 amount) public returns (bool) {
_burn(msg.sender, amount);
        return true;
}

/**
 * @dev See {IERC20-allowance}.
 */

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

/**
 * @dev See {IERC20-approve}.
 *
 * Requirements:
 *
 * - `spender` cannot be the zero address.
 */
function approve(address spender, uint256 value) public returns (bool) {
        _approve(msg.sender, spender, value);
        return true;
}

/**
 * @dev See {IERC20-transferFrom}.
 *
 * Emits an {Approval} event indicating the updated allowance. This is not
 * required by the EIP. See the note at the beginning of {ERC20};
 *
 * Requirements:
 * - `sender` and `recipient` cannot be the zero address.
 * - `sender` must have a balance of at least `value`.
 * - the caller must have allowance for `sender`'s tokens of at least
 * `amount`.
```

```
 */
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));
        return true;
}




/**
 * @dev Atomically increases the allowance granted to `spender` by the caller.
 *
 * This is an alternative to {approve} that can be used as a mitigation for
 * problems described in {IERC20-approve}.
 *
 * Emits an {Approval} event indicating the updated allowance.
 *
 * Requirements:
 *
 * - `spender` cannot be the zero address.
 */
function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
        _approve(msg.sender, spender, _allowances[msg.sender][spender].add(addedValue));
        return true;
}

/**
 * @dev Atomically decreases the allowance granted to `spender` by the caller.
 *
 * This is an alternative to {approve} that can be used as a mitigation for
 * problems described in {IERC20-approve}.
 *
 * Emits an {Approval} event indicating the updated allowance.
 *
 * Requirements:
```

```
     *
     * - `spender` cannot be the zero address.
     * - `spender` must have allowance for the caller of at least
     * `subtractedValue`.
     */
    function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
        _approve(msg.sender, spender, _allowances[msg.sender][spender].sub(subtractedValue));
        return true;
    }

    /**
     * @dev Moves tokens `amount` from `sender` to `recipient`.
     *
     * This is internal function is equivalent to {transfer}, and can be used to
     * e.g. implement automatic token fees, slashing mechanisms, etc.
     *
     * Emits a {Transfer} event.
     *
     * Requirements:
     *
     * - `sender` cannot be the zero address.
     * - `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `amount`.
     */
    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
        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);
    }

    /** @dev Creates `amount` tokens and assigns them to `account`, increasing
     * the total supply.
     *
```

```
     * Emits a {Transfer} event with `from` set to the zero address.
     *
     * Requirements
     *
     * - `to` cannot be the zero address.
     */
    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);
    }

     /**
     * @dev Destroys `amount` tokens from `account`, reducing the
     * total supply.
     *
     * Emits a {Transfer} event with `to` set to the zero address.
     *
     * Requirements
     *
     * - `account` cannot be the zero address.
     * - `account` must have at least `amount` tokens.
     */
    function _burn(address account, uint256 value) internal {
        require(account != address(0), "ERC20: burn from the zero address");

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

    /**
     * @dev Sets `amount` as the allowance of `spender` over the `owner`s tokens.
     *
```

```
     * This is internal function is equivalent to `approve`, and can be used to
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     */
    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);
    }

    /**
     * @dev Destoys `amount` tokens from `account`.`amount` is then deducted
     * from the caller's allowance.
     *
     * See {_burn} and {_approve}.
     */
    function _burnFrom(address account, uint256 amount) internal {
        _burn(account, amount);
        _approve(account, msg.sender, _allowances[account][msg.sender].sub(amount));
    }
}
```

ERC20Detailed.sol

```
pragma solidity ^0.5.0;

import "./IERC20.sol";

/**
```

```
 * @dev Optional functions from the ERC20 standard.
 */
contract ERC20Detailed is IERC20 {
    string private _name;
    string private _symbol;
    uint8 private _decimals;

    /**
     * @dev Sets the values for `name`, `symbol`, and `decimals`. All three of
     * these values are immutable: they can only be set once during
     * construction.
     */
    constructor (string memory name, string memory symbol, uint8 decimals) public {
        _name = name;
        _symbol = symbol;
        _decimals = decimals;
    }

    /**
     * @dev Returns the name of the token.
     */
    function name() public view returns (string memory) {
        return _name;
    }

    /**
     * @dev Returns the symbol of the token, usually a shorter version of the
     * name.
     */
    function symbol() public view returns (string memory) {
        return _symbol;
    }

    /**
     * @dev Returns the number of decimals used to get its user representation.
     * For example, if `decimals` equals `2`, a balance of `505` tokens should
```

```
     * be displayed to a user as `5,05` (`505 / 10 ** 2`).
     *
     * Tokens usually opt for a value of 18, imitating the relationship between
     * Ether and Wei.
     *
     * NOTE: This information is only used for _display_ purposes: it in
     * no way affects any of the arithmetic of the contract, including
     * {IERC20-balanceOf} and {IERC20-transfer}.
     */
    function decimals() public view returns (uint8) {
        return _decimals;
    }
}
```

IERC20.sol

```
pragma solidity ^0.5.0;

/**
 * @dev Interface of the ERC20 standard as defined in the EIP. Does not include
 * the optional functions; to access them see {ERC20Detailed}.
 */
interface IERC20 {
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens owned by `account`.
     */
    function balanceOf(address account) external view returns (uint256);

    /**
     * @dev Moves `amount` tokens from the caller's account to `recipient`.
     *
```

```
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, uint256 amount) external returns (bool);

    /**
     * @dev Returns the remaining number of tokens that `spender` will be
     * allowed to spend on behalf of `owner` through {transferFrom}. This is
     * zero by default.
     *
     * This value changes when {approve} or {transferFrom} are called.
     */
    function allowance(address owner, address spender) external view returns (uint256);

    /**
     * @dev Sets `amount` as the allowance of `spender` over the caller's tokens.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * IMPORTANT: Beware that changing an allowance with this method brings the risk
     * that someone may use both the old and the new allowance by unfortunate
     * transaction ordering. One possible solution to mitigate this race
     * condition is to first reduce the spender's allowance to 0 and set the
     * desired value afterwards:
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     *
     * Emits an {Approval} event.
     */
    function approve(address spender, uint256 amount) external returns (bool);

    /**
     * @dev Moves `amount` tokens from `sender` to `recipient` using the
     * allowance mechanism. `amount` is then deducted from the caller's
     * allowance.
     *
```

```
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

    /**
     * @dev Emitted when `value` tokens are moved from one account (`from`) to
     * another (`to`).
     *
     * Note that `value` may be zero.
     */
    event Transfer(address indexed from, address indexed to, uint256 value);

    /**
     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256 value);
}
```

SafeMath.sol

```
pragma solidity ^0.5.0;

/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * `SafeMath` restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
```

```
 * class of bugs, so it's recommended to use it always.
 */
library SafeMath {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b <= a, "SafeMath: subtraction overflow");
        uint256 c = a - b;

        return c;
    }

    /**
```

```
 * @dev Returns the multiplication of two unsigned integers, reverting on
 * overflow.
 *
 * Counterpart to Solidity's `*` operator.
 *
 * Requirements:
 * - Multiplication cannot overflow.
 */
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
    // benefit is lost if 'b' is also tested.
    // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
    if (a == 0) {
        return 0;
    }

    uint256 c = a * b;
    require(c / a == b, "SafeMath: multiplication overflow");

    return c;
}

/**
 * @dev Returns the integer division of two unsigned integers. Reverts on
 * division by zero. The result is rounded towards zero.
 *
 * Counterpart to Solidity's `/` operator. Note: this function uses a
 * `revert` opcode (which leaves remaining gas untouched) while Solidity
 * uses an invalid opcode to revert (consuming all remaining gas).
 *
 * Requirements:
 * - The divisor cannot be zero.
 */
function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // Solidity only automatically asserts when dividing by 0
    require(b > 0, "SafeMath: division by zero");
```

```
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
     * Reverts when dividing by zero.
     *
     * Counterpart to Solidity's `%` operator. This function uses a `revert`
     * opcode (which leaves remaining gas untouched) while Solidity uses an
     * invalid opcode to revert (consuming all remaining gas).
     *
     * Requirements:
     * - The divisor cannot be zero.
     */
    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b != 0, "SafeMath: modulo by zero");
        return a % b;
    }
}
```

Token.sol

```
// 0.5.1-c8a2
// Enable optimization
pragma solidity ^0.5.0;

import "./ERC20.sol";
import "./ERC20Detailed.sol";
/**
 * @title SimpleToken
 * @dev Very simple ERC20 Token example, where all tokens are pre-assigned to the creator.
 * Note they can later distribute these tokens as they wish using `transfer` and other
 * `ERC20` functions.
 */
contract Token is ERC20, ERC20Detailed {
```

```
    /**
     * @dev Constructor that gives msg.sender all of existing tokens.
     */
    constructor () public ERC20Detailed("Payment chain", "CTH", 6) {
        _mint(msg.sender, 210000000 * (10 ** uint256(decimals())));
    }
}
```

10. Appendix:Analysis tools

10.1.Solgraph

Solgraph is used to generate a graph of the call relationship between smart contract functions, which is convenient for quickly understanding the call relationship between smart contract functions.

Project address：https://github.com/raineorshine/solgraph

10.2.Sol2uml

Sol2uml is used to generate the calling relationship between smart contract functions in the form of UML diagram.

Project address：https://github.com/naddison36/sol2uml

10.3.Remix-ide

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

Project address：http://remix.ethereum.org

10.4.Ethersplay

Etherplay is a plug-in for binary ninja. It can be used to analyze EVM bytecode and graphically present the function call process.

Project address：https://github.com/crytic/ethersplay

10.5.Mythril

Mythril is a security audit tool for EVM bytecode, and supports online contract

audit.

Project address：https://github.com/ConsenSys/mythril

10.6.Echidna

Echidna is a security audit tool for EVM bytecode. It uses fuzzy testing technology and supports integrated use with truss.

Project address：https://github.com/crytic/echidna

11. DISCLAIMERS

Chainlion only issues this report on the facts that have occurred or existed before the issuance of this report, and assumes corresponding responsibilities. For the facts occurring or existing after the issuance, chainlion cannot judge the security status of its smart contract, and is not responsible for it. The security audit analysis and other contents in this report are only based on the documents and materials provided by the information provider to chainlion as of the issuance of this report. Chainlion assumes that the information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed or reflected inconsistent with the actual situation, chainlion shall not be liable for the losses and adverse effects caused thereby. Chainlion only conducted

the agreed safety audit on the safety of the project and issued this report. Chainlion is not responsible for the background and other conditions of the project.