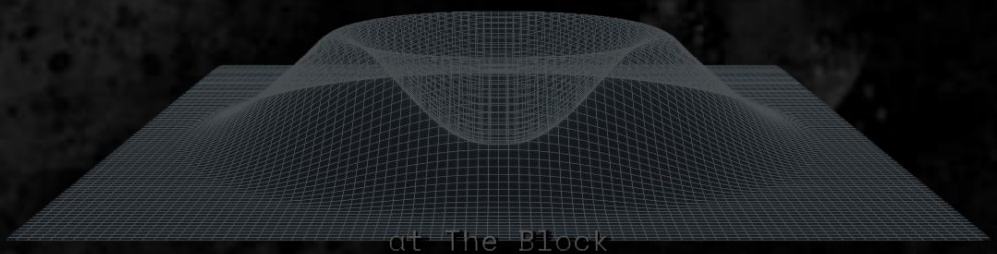
CHAINLION

BND

Smart Contract Audit Report

OCT 09th，2022

NO.0C002210090001



CATALOGUE

1. PROJECT SUMMARY

Entry type	Specific description
Entry name	BND
Project type	BEP-20
Application platform	BSC
DawnToken	0xe06104eA6E76Bb68cf5fDDE1e71B90511AeFA07E

2. AUDIT SUMMARY

Entry type	Specific description
Project cycle	OCT/07/2022-OCT/09/2022
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 **BND** 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.

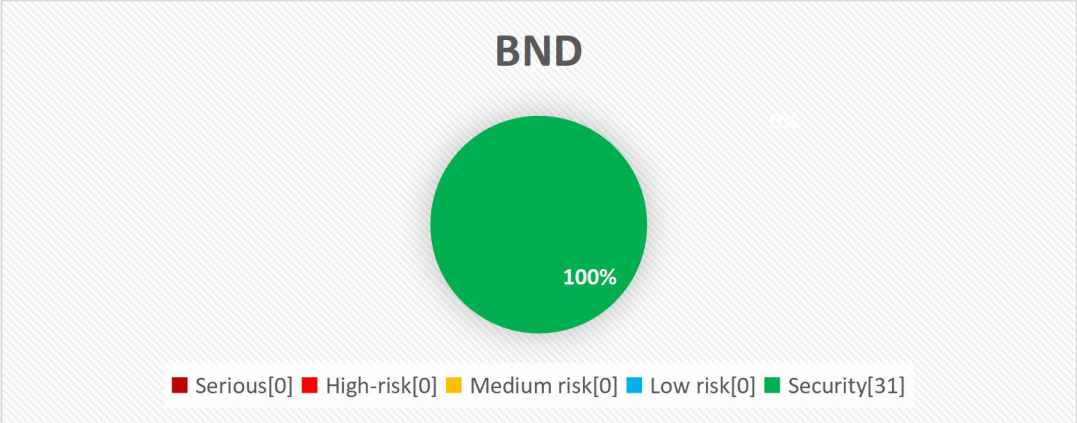
5. Directory structure

```
ERC20.sol
Ido.sol
```

6. File hashes

Contract	SHA1 Checksum
ERC20.sol	E037766A205C1047E9A176BD3D579B25AF68564A
Ido.sol	75A97BA4F35F3C902DCFAC797CF0A7A1C2243649

7. Vulnerability distribution


BND
100%
Serious[0]
High-risk[0]
Medium risk[0]
Low risk[0]
Security[31]

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 0.8.0, so there is no such security problem.

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface IERC20 {
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);
    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns (uint256);
    function transfer(address to, uint256 amount) external returns (bool);
    function allowance(address owner, address spender) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
    function transferFrom(address from, address to, uint256 amount) external returns (bool);
}
```

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.

```
32  contract ERC20 is Context, IERC20, IERC20Metadata {
33      mapping(address => uint256) private _balances;
34
35      mapping(address => mapping(address => uint256)) private _allowances;
36
37      uint256 private _totalSupply;
38
39      string private _name;
40      string private _symbol;
41
42
43      constructor(string memory name_, string memory symbol_,uint256 amount_) {
44          _name = name_;
45          _symbol = symbol_;
46          _mint(msg.sender, amount_);
47      }
48
```

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： After 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： After 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 is applicable to the compiler of version 0.8.0, and the safemath library is used for numerical operation, which better prevents the problem of shaping overflow.

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface IERC20 {
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);
    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns (uint256);
    function transfer(address to, uint256 amount) external returns (bool);
    function allowance(address owner, address spender) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
    function transferFrom(address from, address to, uint256 amount) external returns (bool);
}
```

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： ERC20.sol L32~47

Code information：

```
contract ERC20 is Context, IERC20, IERC20Metadata {
    mapping(address => uint256) private _balances;    //Asset quantity retrieval corresponding to address

    mapping(address => mapping(address => uint256)) private _allowances; //Authorization limit retrieval

    uint256 private _totalSupply;    //Total number of tokens issued

    string private _name;    //Token name
    string private _symbol;    //Token symbol


    constructor(string memory name_, string memory symbol_,uint256 amount_) {
        _name = name_;    //Initialize token name
        _symbol = symbol_; //Initialize token symbol
        _mint(msg.sender, amount_);    //Initialize the number of tokens issued
    }

contract Ido is Context, Ownable, Pausable, ReentrancyGuard {
    using SafeERC20 for IERC20;

    IERC20 public usdt;          //Usdt contract address
    IERC20 public idoToken;    //Ido contract address
    uint256 public maxLimit;    //Maximum transaction quantity limit
    uint256    public    basePoint; //basic point
    uint256 public    CAP; //upper limit
    uint256 public    price; //Price
    address public payeeAddress; //Receiver Address
    uint256 private _totalSold = 0; //Total sales
    mapping(address => uint256) public buyRound; //Buy Round

    event activeLog(uint256 time);
    event BuyLog(address buyer,    uint256 _amount);

    constructor(IERC20 usdtAddress_,IERC20 idoToken_, uint256 cap_,uint256 maxLimit_, uint256 price_, address    payeeAddress_, uint256 basePoint_)    {
```

```
        usdt = usdtAddress_;      //Usdt contract address
        idoToken = idoToken_;     //Ido contract address
        CAP = cap_;                 //upper limit
        maxLimit = maxLimit_;     //Maximum transaction quantity limit
        price = price_;             //Price
        basePoint = basePoint_; //basic point
        payeeAddress = payeeAddress_;     //Receiver Address

        }
```

Safety advice： NONE.

8.3.2. Contract Basic Information Query Function【security】

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

Audit results： No relevant safety issues.

Code file： ERC20.sol L49~67

Code information：

```
    function name() public view virtual override returns (string memory) {
        return _name;    //Token name retrieval
    }

    function symbol() public view virtual override returns (string memory) {
        return _symbol;    //Token symbol retrieval
    }

    function decimals() public view virtual override returns (uint8) {
        return 18;    //Token precision retrieval
    }

    function totalSupply() public view virtual override returns (uint256) {
        return _totalSupply;    //Search of total token issuance
    }

    function balanceOf(address account) public view virtual override returns (uint256) {
        return _balances[account];    //Retrieve the asset quantity corresponding to the address
```

```
}
```

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 design of relevant business logic is reasonable.

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

Code file： ECR20.sol L69~73

Code information：

```
    function transfer(address to, uint256 amount) public virtual override returns (bool) {
        address owner = _msgSender();    //Get Function Caller
        _transfer(owner, to, amount);    //Call_ Transfer completes transfer
        return true;
    }
    function _transfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {
        require(from != address(0), "ERC20: transfer from the zero address"); //Address non-zero check
        require(to != address(0), "ERC20: transfer to the zero address");//Address non-zero check

        _beforeTokenTransfer(from, to, amount);

        uint256 fromBalance = _balances[from]; //Obtain the number of assets from the address account before the transfer
        require(fromBalance >= amount, "ERC20: transfer amount exceeds balance"); //Check whether the quantity of assets held by the from address is greater than the quantity to be transferred
    unchecked {
        _balances[from] = fromBalance - amount; //Update the number of assets held by the from address
        _balances[to] += amount;    //Update the number of assets held by to address
    }
```

```
        emit Transfer(from, to, amount);

        _afterTokenTransfer(from, to, amount);
    }
```

Safety advice： NONE.

8.3.4. Approve authorization related logic【security】

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

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

Code file： ECR20.sol L76~84

Code information：

```
    function allowance(address owner, address spender) public view virtual override returns (uint256)
{
        return _allowances[owner][spender];    //Retrieve Authorization Limit
    }

    function approve(address spender, uint256 amount) public virtual override returns (bool) {
        address owner = _msgSender(); //Retrieve function callers
        _approve(owner, spender, amount);    //Perform authorization operation
        return true;
    }
    function _approve(
        address owner,
        address spender,
        uint256 amount
    ) internal virtual { //Only internal calls are allowed
        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] = amount;    //Update authorization limit
        emit Approval(owner, spender, amount);
    }
    function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
        address owner = _msgSender();    //Retrieve function callers
        _approve(owner, spender, allowance(owner, spender) + addedValue);        //Update
authorization limit (increase)
        return true;
    }

    function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool)
{
        address owner = _msgSender();      //Retrieve function callers
        uint256 currentAllowance = allowance(owner, spender);     //Get the current authorization
limit
        require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");
//Check whether the current authorization limit is greater than the limit to be reduced
    unchecked {
        _approve(owner, spender, currentAllowance - subtractedValue);//Update authorization limit
(decrease)
    }

        return true;

    }
```

Safety advice： NONE.

8.3.5. TransferFrom transfer logic design【security】

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

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

Code file： ECR20.sol L86~95

Code information：

```
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        address spender = _msgSender(); //Retrieve function callers
        _spendAllowance(from, spender, amount);    //Update authorization limit
        _transfer(from, to, amount);    //Call_ Transfer
        return true;
    }
    function _spendAllowance(
        address owner,
        address spender,
        uint256 amount
    ) internal virtual {
        uint256 currentAllowance = allowance(owner, spender);       //Retrieve the current authorization limit
        if (currentAllowance != type(uint256).max) { //Check whether the current quota is not equal to type (uint256). max
            require(currentAllowance >= amount, "ERC20: insufficient allowance"); //Check whether the current quota is greater than the quota to be transferred. If it is greater than the quota, return
        unchecked {
            _approve(owner, spender, currentAllowance - amount); //Update authorized transfer limit
        }
        }
    }
    function _transfer(
        address from,
        address to,
        uint256 amount
    ) internal virtual {
        require(from != address(0), "ERC20: transfer from the zero address"); //Address non-zero check
        require(to != address(0), "ERC20: transfer to the zero address");//Address non-zero check

        _beforeTokenTransfer(from, to, amount);

        uint256 fromBalance = _balances[from]; //Obtain the number of assets from the address account before the transfer
        require(fromBalance >= amount, "ERC20: transfer amount exceeds balance"); //Check
```

```
whether the quantity of assets held by the from address is greater than the quantity to be transferred
    unchecked {
        _balances[from] = fromBalance - amount; //Update the number of assets held by the from address
        _balances[to] += amount;    //Update the number of assets held by to address
    }

        emit Transfer(from, to, amount);

        _afterTokenTransfer(from, to, amount);

        }
```

Safety advice： NONE.

8.3.6. Logic design of token issuing business【security】

Audit description： Additional issuing function for tokens in the contract_ Mint conducts code audit to check whether the design of relevant business logic is reasonable.

Audit results： Token destruction function in the contract_ The mint design is correct, and no relevant safety risks are found.

Code file： ECR20.sol L136~144

Code information：

```
function _mint(address account, uint256 amount) internal virtual { //Can only be called internally
        require(account != address(0), "ERC20: mint to the zero address"); //Address non-zero check

        _beforeTokenTransfer(address(0), account, amount);

        _totalSupply += amount; //Update total assets
    unchecked {
        _balances[account] += amount;    //Update the number of tokens held by the address
    }
        emit Transfer(address(0), account, amount);

        _afterTokenTransfer(address(0), account, amount);
```

```
}
```

Safety advice： NONE.

8.3.7. 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 design of relevant business logic 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 L150~160

Code information：

```
function _burn(address account, uint256 amount) internal virtual {    //Can only be called internally
        require(account != address(0), "ERC20: burn from the zero address"); //Address non-zero check

        _beforeTokenTransfer(account, address(0), amount);

        uint256 accountBalance = _balances[account]; //Get the number of tokens held by the current address
        require(accountBalance >= amount, "ERC20: burn amount exceeds balance"); //Check whether the current token quantity is greater than the quantity to be destroyed
    unchecked {
        _balances[account] = accountBalance - amount; //Update address account asset quantity
        _totalSupply -= amount; //Update total tokens
    }

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

        _afterTokenTransfer(account, address(0), amount);

    }
```

Safety advice： NONE.

8.3.8. Logic design of contract start stop business【security】

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

Audit results： The logic design related to startup and shutdown in the contract is correct, and no relevant safety risk is found.

Code file： ido.sol L458~479

Code information：

```
function paused() public view virtual returns (bool) {
        return _paused; //Status retrieval
    }

    function _requireNotPaused() internal view virtual { //Only internal calls are allowed
        require(!paused(), "Pausable: paused"); //Non paused status check
    }

    function _requirePaused() internal view virtual {//Only internal calls are allowed
        require(paused(), "Pausable: not paused"); //Pause status check
    }

    function _pause() internal virtual whenNotPaused { //Called internally and not paused
        _paused = true;    //Change the status to Pause
        emit Paused(_msgSender());
    }

    function _unpause() internal virtual whenPaused {//Called internally and paused
        _paused = false; //Change the status to "Not Paused"
        emit Unpaused(_msgSender());

    }
```

Safety advice：： NONE.

8.3.9. Contract Owner Update Logic【security】

Audit description： Audit the owner update logic in the contract to check whether the zero address is verified and whether the relevant business logic is reasonable.

Audit results： The owner update related operations in the contract are correct.

Code file： ido.sol 481~517

Code information：

```
abstract contract Ownable is Context {
    address private _owner;

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

    constructor() {
        _transferOwnership(_msgSender());
    }

    modifier onlyOwner() {
        _checkOwner();
        _;
    }

    function owner() public view virtual returns (address) {
        return _owner;
    }

    function _checkOwner() internal view virtual {
        require(owner() == _msgSender(), "Ownable: caller is not the owner");    //Check whether the caller is the contract owner
    }

    function renounceOwnership() public virtual onlyOwner { //Only the owner of the contract can call
        _transferOwnership(address(0)); //更新 owner 地址
    }

    function transferOwnership(address newOwner) public virtual onlyOwner { //Only the owner of the contract can call
```

```
        require(newOwner != address(0), "Ownable: new owner is the zero address"); //Address
non-zero check
        _transferOwnership(newOwner); //Update the owner of the contract
    }

    function _transferOwnership(address newOwner) internal virtual {
        address oldOwner = _owner; //Staging the old owner address
        _owner = newOwner; //Update owner address
        emit OwnershipTransferred(oldOwner, newOwner);
    }
}
```

Safety advice：：NONE.

8.3.10. Check the current remaining quantity logic【security】

Audit description： Audit the business logic of checking the current remaining quantity in the contract to check whether there are design defects such as shaping overflow and permission design.

Audit results： Check the current remaining quantity in the contract. The business logic design is correct.

Code file： ido.sol 548~551

Code information：

```
 function checkCurrentRoundRemaining() public view returns (uint256) { //Query current round IDO

Number of remaining tokens

        return CAP - _totalSold; //Return remaining quantity
    }
    function checkCurrentMyRemaining (address account_)public view returns (uint256) {//View the
remaining purchase quota of the current user.
        uint256    remainingTotal = CAP - _totalSold; // Remaining quantity
        if (CAP - _totalSold <= 0){ //If it is 0 or less than 0, return 0 directly
            return 0;
        }
        if (remainingTotal < maxLimit - buyRound[account_] ) { //If the remaining quantity is less
```

```
than the difference between the maximum quantity and the purchased quantity of the address, the remaining quantity is returned
                return remainingTotal ;
            } else {
                return maxLimit - buyRound[account_] ;      //Otherwise, return the remaining purchasable difference
            }

    }
```

Safety advice： NONE.

8.3.11. Logic design of IDO purchase business【security】

Audit description： Audit the IDO purchase business logic within the contract to check whether there are design defects such as shaping overflow, lack of necessary non-zero address check, and permission design.

Audit results： The logic design of IDO purchase business in the contract is correct.

Code file： ido.sol 553~568

Code information：

```
 function buy(uint256 amount_) public whenNotPaused nonReentrant { //Non paused state, preventing reentry
            address myAddress = _msgSender(); //Retrieve function callers
            require(myAddress == tx.origin, "contract can not buy"); //Check whether the caller is a contract address. If yes, purchase is prohibited
            require(address(idoToken) != address(0), "idoToken not set"); //Check whether the idoToken address is zero (that is, uninitialized)

            require(_totalSold + amount_ <= CAP, "can not exceed max cap"); //Check whether the maximum number of rounds is exceeded
            require( amount_+    buyRound[myAddress] <= maxLimit, "maxLimit");    //Single address purchase upper limit check
            require(amount_ * price / basePoint <= usdt.balanceOf(myAddress), "usdt value sent is not enough"); //Insufficient usdt values available for sending

            _totalSold += amount_; //Update sales volume
```

```
        buyRound[myAddress] += amount_; //Update the purchased quantity of the address
        uint256 usdtAmount = (amount_ * price / basePoint); //Calculate the USDT quantity to be paid
        usdt.safeTransferFrom(myAddress, payeeAddress, usdtAmount); //Transfer USDT to payeeAddress address
        idoToken.safeTransfer(myAddress, amount_); //Transfer ido to myAddress
        emit BuyLog(myAddress, amount_); //Add purchase record
    }
```

Safety advice： ： NONE.

8.3.12. Logic design of IDO withdrawal business【security】

Audit description： Audit the IDO withdrawal business logic within the contract to check whether there are design defects such as shaping overflow, lack of necessary non-zero address check, and permission design.

Audit results： The logic design of IDO withdrawal business in the contract is correct.

Code file： ido.sol 591~595

Code information：

```
 function withdrawToken(address to, uint256 amount) public onlyOwner { //Only contract owner calls are allowed
        require(amount <= idoToken.balanceOf(address(this)), "idoToken value sent is not enough"); //Check whether the current withdrawal quantity is less than the ido quantity in the current contract
        idoToken.safeTransfer(to, amount); //Withdraw the remaining unsold tokens to the designated account
    }
```

Safety advice：NONE.

9. Appendix:Analysis tools

9.1.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

9.2.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

9.3.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

9.4.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

9.5.Mythril

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

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

9.6.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

10. 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.