MT

Smart Contract Audit Report

NOV 07th，2022

NO.0C002211070001

CATALOGUE

1. PROJECT SUMMARY

Entry type	Specific description
Entry name	MT
Project type	**ERC-20**
Application platform	BSC
DawnToken	0x2f3f25046ea518d1e524b8fb6147c656d6722ced

2. AUDIT SUMMARY

Entry type	Specific description
Project cycle	NOV/02/2022-NOV/07/2022
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 **MT** 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

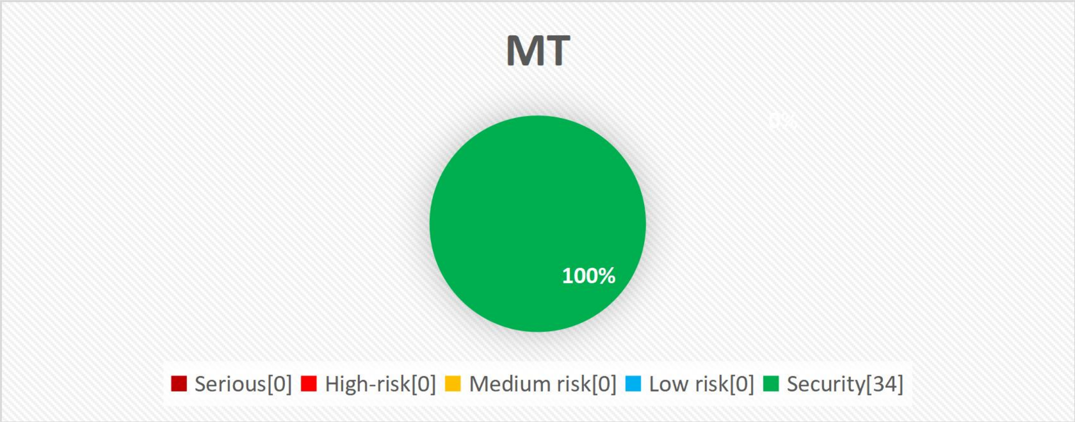
```
MetaverseToken.sol
```

6. File hashes

Contract	SHA1 Checksum

MetaverseToken.sol	313EAFB1F7F2575AFDE7A5A9BD76447110C3C076

7. Vulnerability distribution



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.

```
/**
 *Submitted for verification at BscScan.com on 2022-08-16
*/

// File: @openzeppelin\contracts\GSN\Context.sol

pragma solidity ^0.5.0;

/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
```

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.

```
/**
 * @dev Extension of {ERC20} that adds a set of accounts with the {MinterRole},
 * which have permission to mint (create) new tokens as they see fit.
 *
 * At construction, the deployer of the contract is the only minter.
 */
contract ERC20Mintable is ERC20, MinterRole,ReentrancyGuard {
    using SafeMath for uint256;
    using Address for address;
    uint256 private constant BASE_RATIO = 10**18;
    mapping (address => uint256) private _limit;
    uint256 private _transactFeeValue = 5;
    uint private sale_date=0;

    uint256[][] private ContractorsFee ;
    address[][] private ContractorsAddress;

    address public _sell;
    address public _dis;

    constructor(address dis) public {
        _dis=dis;
    }
```

```
contract MetaverseToken is ERC20, ERC20Detailed,ERC20Mintable {
    constructor(address dis) public ERC20Detailed("MT", "MT", 18) ERC20Mintable(dis) {
        uint256 totalSupply =22400* (10**uint256(18));
        _mint(address(this), totalSupply);
        _mint(dis, 8939* (10**uint256(18)));
        addMinter(dis);
    }
}
```

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.

```
    function setTransactFee(uint256 fee)public onlyMinter {
        _transactFeeValue=fee;
    }

    function saleDate() public view returns (uint) {
        return sale_date;
    }

    function setSaleDate(uint date) public onlyMinter {
        sale_date=date;
    }
    function setDis(address dis) public onlyMinter {
        _dis=dis;
    }
    function setSell(address token) public onlyMinter {

        _sell=token;
    }
```

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.

```
    function setDis(address dis) public onlyMinter {
        _dis=dis;
    }
    function setSell(address token) public onlyMinter {

        _sell=token;
    }
```

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 the SafeMath function library to perform numerical operations, which can better prevent the shaping overflow problem caused by numerical operations.

```
/**
 * @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) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
```

```
 * Requirements:
 * - Subtraction cannot overflow.
 *
 * _Available since v2.4.0._
 */
function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        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) {
        return div(a, b, "SafeMath: division by zero");
}

/**
 * @dev Returns the integer division of two unsigned integers. Reverts with custom message 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.
 *
 * _Available since v2.4.0._
 */
function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
        require(b > 0, errorMessage);
        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) {
        return mod(a, b, "SafeMath: modulo by zero");
}

/**
```

```
    * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
    * Reverts with custom message 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.
    *
    * _Available since v2.4.0._
    */
   function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
       require(b != 0, errorMessage);
       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： It is audited that the smart contract uses the openzeppelin official class library to prevent reentry attacks.

```
/**
 * @dev Contract module that helps prevent reentrant calls to a function.
 *
 * Inheriting from `ReentrancyGuard` will make the {nonReentrant} modifier
 * available, which can be applied to functions to make sure there are no nested
 * (reentrant) calls to them.
 *
 * Note that because there is a single `nonReentrant` guard, functions marked as
 * `nonReentrant` may not call one another. This can be worked around by making
 * those functions `private`, and then adding `external` `nonReentrant` entry
 * points to them.
```

```
 *
 * TIP: If you would like to learn more about reentrancy and alternative ways
 * to protect against it, check out our blog post
 * https://blog.openzeppelin.com/reentrancy-after-istanbul/[Reentrancy After Istanbul].
 *
 * _Since v2.5.0:_ this module is now much more gas efficient, given net gas
 * metering changes introduced in the Istanbul hardfork.
 */
contract ReentrancyGuard {
    bool private _notEntered;

    constructor () internal {
        // Storing an initial non-zero value makes deployment a bit more
        // expensive, but in exchange the refund on every call to nonReentrant
        // will be lower in amount. Since refunds are capped to a percetange of
        // the total transaction's gas, it is best to keep them low in cases
        // like this one, to increase the likelihood of the full refund coming
        // into effect.
        _notEntered = true;
    }

    /**
     * @dev Prevents a contract from calling itself, directly or indirectly.
     * Calling a `nonReentrant` function from another `nonReentrant`
     * function is not supported. It is possible to prevent this from happening
     * by making the `nonReentrant` function external, and make it call a
     * `private` function that does the actual work.
     */
    modifier nonReentrant() {
        // On the first call to nonReentrant, _notEntered will be true
        require(_notEntered, "ReentrancyGuard: reentrant call");

        // Any calls to nonReentrant after this point will fail
        _notEntered = false;

        _;

        // By storing the original value once again, a refund is triggered (see
        // https://eips.ethereum.org/EIPS/eip-2200)
        _notEntered = true;
    }
```

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. Token basic information query logic【security】

Audit description： Conduct security audit on the logic design of token information query business in the contract to check whether there are any defects in the logic design.

Audit results： The design of token information query business logic in the contract is correct.

Code file： MetaverseToken.sol 304~324

Code information：

```
using SafeMath for uint256;

    mapping (address => uint256) private _balances;    //Address Asset Quantity Mapping

    mapping (address => mapping (address => uint256)) private _allowances;    //Authorizer - Licensee - Authorization Limit Mapping

    uint256 private _totalSupply;

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

    /**
     * @dev See {IERC20-balanceOf}.
     */
    function balanceOf(address account) public view returns (uint256) {
        return _balances[account];    //Check the number of assets held in the account
    }
    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 symbol
    _decimals = decimals; //Token accuracy
}

/**
 * @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

    }
```

Safety advice：NONE.

8.3.2. Transfer transfer business logic【security】

Audit description： Conduct security audit on the logic design of the transfer business in the contract to check whether there are any defects in the logic design.

Audit results： The transfer business logic design in the contract is correct.

Code file： MetaverseToken.sol 334~337

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(_msgSender(), recipient, amount);    //Call_ 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 {
        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, "ERC20: transfer amount exceeds balance"); //Update transfer address asset quantity
        _balances[recipient] = _balances[recipient].add(amount); //Update the number of assets held by the token receiving address
        emit Transfer(sender, recipient, amount);

    }
```

Safety advice：NONE.

8.3.3. Logic design of authorized transfer business【security】

Audit results： Conduct security audit on the logic design of authorized transfer business in the contract to check whether there are any defects in the logic design.

Audit results： The logic design of authorized transfer business in the contract is correct.

Code file： MetaverseToken.sol 339~356

Code information：

```
    /**
     * @dev See {IERC20-allowance}.
     */
    function allowance(address owner, address spender) public view returns (uint256) {
        return _allowances[owner][spender]; //Retrieve Authorization Limit
    }

    /**
     * @dev See {IERC20-approve}.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 amount) public returns (bool) {
        _approve(_msgSender(), spender, amount); //Call_ Approve to perform authorization operations
        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(_msgSender(), spender, _allowances[_msgSender()][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(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue,
"ERC20: decreased allowance below zero")); //Reduce the amount of authorization
        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 amount) internal {
     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);
 }
```

Safety advice：NONE.

8.3.4. TransferFrom transfer business【security】

Audit description： Conduct security audit on the logic design of transferFrom transfer business in the contract to check whether there are any defects in the logic design.

Audit results： The logical design of transferFrom transfer business in the contract is correct.

Code file： MetaverseToken.sol 370~374

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 `amount`.
  * - 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); //Call_ Transfer
        _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "ERC20: transfer amount exceeds allowance")); //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 {
        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, "ERC20: transfer amount exceeds balance"); //Update transfer address asset quantity
        _balances[recipient] = _balances[recipient].add(amount); //Update the number of assets held by the token receiving address
        emit Transfer(sender, recipient, amount);

    }
```

Safety advice: NONE.

8.3.5. _ mint token issuance business logic【security】

Audit description: For the contract_ Carry out security audit on the logic design of the mint token issuance business to check whether the address parameters are checked for legitimacy and whether there is any risk of coinage.

Audit results: In the contract_ The mint token issuing function is only allowed to be called internally, and the validity of the address parameters is checked. The relevant business logic design is correct.

Code file: MetaverseToken.sol 444~450

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 internal calls are allowed
    require(account != address(0), "ERC20: mint to the zero address"); //Address non-zero check

    _totalSupply = _totalSupply.add(amount); //Update total token issuance
    _balances[account] = _balances[account].add(amount); //Update the number of assets held at
the address
    emit Transfer(address(0), account, amount);
}
```

Safety advice: NONE.

8.3.6. _ Burn token destruction business logic 【security】

Audit description: For the contract_ The logic design of the additional token issuance business of burn is subject to security audit to check whether the validity of address parameters is checked and whether any number of tokens at any address are destroyed.

Audit results: In the contract_ The burn token issuing function is only allowed to be called internally, and the validity of the address parameters is checked. The relevant business logic design is correct.

Code file: MetaverseToken.sol 463~469

Code information:

```
/**
 * @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 amount) internal { //Only internal calls are allowed
    require(account != address(0), "ERC20: burn from the zero address"); //Address parameter non-zero check

    _balances[account] = _balances[account].sub(amount, "ERC20: burn amount exceeds balance"); //Update the number of assets held at the address
    _totalSupply = _totalSupply.sub(amount); //Update total token issuance
    emit Transfer(account, address(0), amount);

}
```

Safety advice: **NONE.**

8.3.7. _ BurnFrom authorizes token destruction 【security】

Audit description: For the contract_ BurnFrom authorizes the design of token destruction business logic for security audit to check whether the address parameters are checked for validity and whether there is data.

Audit results： In the contract_ The burnFrom authorized token destruction function only allows internal calls, and the address parameters are checked for legitimacy. The relevant business logic design is correct.

Code file： MetaverseToken.sol 498~501

Code information：

```
/**
  * @dev Destroys `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 internal calls are allowed
     _burn(account, amount); //Token destruction
     _approve(account, _msgSender(), _allowances[account][_msgSender()].sub(amount,
"ERC20: burn amount exceeds allowance")); //Update authorization limit

 }
```

Safety advice： **NONE.**

8.3.8. Business logic related to role management【security】

Audit description： Conduct security audit on the business logic design related to role management in the contract to check whether the address parameters are checked for legitimacy and whether the logic design has security defects.

Audit results： The design of business logic related to role management in the contract is correct.

Code file： MetaverseToken.sol 564~593

Code information：

```
/**
  * @dev Give an account access to this role.
  */
```

```
function add(Role storage role, address account) internal {
    require(!has(role, account), "Roles: account already has role"); //Check if it already exists
    role.bearer[account] = true; //Add Role
}

/**
 * @dev Remove an account's access to this role.
 */
function remove(Role storage role, address account) internal {
    require(has(role, account), "Roles: account does not have role"); //Check for presence
    role.bearer[account] = false; //remove Role
}

/**
 * @dev Check if an account has this role.
 * @return bool
 */
function has(Role storage role, address account) internal view returns (bool) {
    require(account != address(0), "Roles: account is the zero address"); //Address non-zero
check
    return role.bearer[account]; //Check whether the address exists

}
```

Safety advice： NONE.

8.3.9. Minter management related business logic【security】

Audit description： Conduct security audit on the business logic design related to Minter management in the contract to check whether the address parameters are checked for legitimacy and whether the logic design has security defects.

Audit results： The design of Minter management related business logic in the contract is correct.

Code file： MetaverseToken.sol 595~633

Code information：

```
constructor () internal {
        _addMinter(_msgSender()); //Add the contract owner to the Minter sequence
    }

    modifier onlyMinter() {
        require(isMinter(_msgSender()), "MinterRole: caller does not have the Minter role");
//Permission check
        _;
    }

    function isMinter(address account) public view returns (bool) {
        return _minters.has(account); //Check whether the address exists. If it exists, it means Minter
    }
    function addMinter(address account) public onlyMinter { //Only Minter calls
        _addMinter(account); //increase Minter
    }
    function renounceMinter() public {
        _removeMinter(_msgSender()); //Actively release Minter permission
    }
    function _addMinter(address account) internal { //Internal call
        _minters.add(account); //Add minter (check whether it exists in Role)
        emit MinterAdded(account);
    }
    function _removeMinter(address account) internal {
        _minters.remove(account); //Remove minter (check the existence in Role)
        emit MinterRemoved(account);

    }
```

Safety advice： NONE.

8.3.10. IsContract contract address check【security】

Audit description：Conduct security audit on the business logic design related to the isContract address check in the contract, and check whether the malicious attack payload is filled in the constructor.

Audit results： After audit, there are no relevant problems.

Code file： MetaverseToken.sol 653~662

Code information：

```
/**
     * @dev Returns true if `account` is a contract.
     *
     * [IMPORTANT]
     * ====
     * It is unsafe to assume that an address for which this function returns
     * false is an externally-owned account (EOA) and not a contract.
     *
     * Among others, `isContract` will return false for the following
     * types of addresses:
     *
     *   - an externally-owned account
     *   - a contract in construction
     *   - an address where a contract will be created
     *   - an address where a contract lived, but was destroyed
     * ====
     */
    function isContract(address account) internal view returns (bool) {
        // According to EIP-1052, 0x0 is the value returned for not-yet created accounts
        // and 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470 is returned
        // for accounts without code, i.e. `keccak256('')`
        bytes32 codehash;
        bytes32 accountHash = 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470;
        // solhint-disable-next-line no-inline-assembly
        assembly { codehash := extcodehash(account) }
        return (codehash != accountHash && codehash != 0x0);
    }
    function isContract(address addr) internal view returns (bool) { //Risk of being bypassed
        uint256 size;
        assembly { size := extcodesize(addr) }
        return size > 0;
    }
```

Safety advice： NONE.

8.3.11. Logic design of transaction fee related business【security】

Audit description： Conduct security audit on the business logic design related to transaction fees in the contract to check whether there are design defects.

Audit results: The transaction fee related business logic design in the contract is correct, and no related logic design defects are found.

Code file: MetaverseToken.sol 791~860

Code information:

```
using SafeMath for uint256;
    using Address for address;
    uint256 private constant BASE_RATIO = 10**18; //Basic ratio
    mapping (address => uint256) private _limit; //limit
    uint256 private _transactFeeValue = 5; //Transaction fee
    uint private sale_date=0; //Sales Date

    uint256[][] private ContractorsFee ;
    address[][] private ContractorsAddress;

    address public _sell;
    address public _dis;

    constructor(address dis) public {
        _dis=dis;
    }
    /**
     * @dev See {ERC20-_mint}.
     *
     * Requirements:
     *
     * - the caller must have the {MinterRole}.
     */
    function setTransactFee(uint256 fee)public onlyMinter { //Only Minter calls are allowed
        _transactFeeValue=fee; //Set transaction fee
    }

    function saleDate() public view returns (uint) {
        return sale_date; //Query sales date
    }

    function setSaleDate(uint date) public onlyMinter {//Only Minter calls are allowed
        sale_date=date;    //Set Transaction Date
    }
    function setDis(address dis) public onlyMinter { //Only Minter calls are allowed
        _dis=dis; //Set dis address
```

```
    }
    function setSell(address token) public onlyMinter {//Only Minter calls are allowed

        _sell=token; //Set the token address
    }
    function getTransactFee() public view returns (uint256){
        return _transactFeeValue; //Obtain transaction fee
    }
    function sendTransfer(address account,uint256 amount)public nonReentrant onlyMinter returns
(bool){ //Only Minter calls are allowed, and reentry is prevented
        require(IERC20(address(this)).transfer(account,amount) , "sendTransfer:error"); //Check
whether the transfer is successful
        return true;
    }
    function sendApprove(address account,uint256 amount)public onlyMinter nonReentrant returns
(bool){//Only Minter calls are allowed, and reentry is prevented
        require(IERC20(address(this)).approve(account,amount) , "sendApprove:error"); //Check
whether the authorization operation is successful
        return true;
    }

    function getContractorsFee(uint setType)public view returns (uint256[] memory fee,address[]
memory add){
        fee=ContractorsFee[setType];
        add=ContractorsAddress[setType];
    }

    function setContractorsFee(uint256[] memory fee,address[] memory add,uint setType)public
onlyMinter {//Only Minter calls are allowed
        require(fee.length == add.length , "fee<>add");
        if(ContractorsFee.length<=setType){
            ContractorsFee.push(fee);
            ContractorsAddress.push(add);
        }else{
            ContractorsFee[setType]=fee;
            ContractorsAddress[setType]=add;
        }

    }
```

Safety advice： NONE.

8.3.12. Business logic related to addLiquidity【security】

Audit description： Conduct security audit on the logic design of adding liquidity business to addLiquidity in the contract to check whether there are design defects.

Audit results: The logic design of the added liquidity business in the contract is correct, and no related logic design defects are found.

Code file: MetaverseToken.sol 862~870

Code information:

```
function addLiquidity(
        uint amountUsdt,
        uint amountTokenDesired
    ) public onlyMinter returns (uint256 amountA,uint256 amountB,uint256 liquidity){ //Only Minter calls are allowed to increase liquidity
        require(amountUsdt > 0, "addLiquidity: amountETH >0"); //Check whether the increased USDT value is greater than 0
        require(amountTokenDesired > 0, "addLiquidity: amountTokenDesired >0");//Check whether the TokenDesired value added correspondingly is greater than 0
        require(IERC20(address(this)).approve(_sell,amountTokenDesired), "addLiquidity: approve _roter error"); //to grant authorization

(amountA,amountB,liquidity)=IUniswapV2Router01(_sell).addLiquidity(address(this),amountUsdt,amountTokenDesired,1,1); //Increase liquidity

    }
```

Safety advice: **NONE.**

8.3.13. Business logic related to transferLiquidity【security】

Audit description: Conduct security audit on the transferLiquidity business logic design in the contract to check whether there are design defects.

Audit results: The transferLiquidity business logic design in the contract is correct, and no related logic design defects are found.

Code file: MetaverseToken.sol 872~878

Code information:

```
function transferLiquidity(
```

```
        uint amountUsdt
    ) public onlyMinter returns (uint256 liquidity) {//Only Minter calls are allowed to transfer liquidity
        require(amountUsdt > 0, "addLiquidity: amountETH >0"); //Check whether the USDT value is greater than 0
        require(IERC20(address(this)).approve(_sell,1000000000), "addLiquidity: approve _roter error");//grantauthorization
liquidity=IUniswapV2Router01(_sell).transferLiquidity(address(this),amountUsdt); //Transfer liquidity
    }
```

Safety advice：NONE.

8.3.14. TransactionFee related business logic【security】

Audit description： Conduct security audit on the business logic design of transactionFee transaction cost calculation in the contract to check whether there are design defects.

Audit results： The transactionFee fee calculation business logic design in the contract is correct, and no related logic design defects are found.

Code file： MetaverseToken.sol 886~930

Code information：

```
 function transactionFee(address from,address to,uint256 amount)internal returns (uint256) {

if(_msgSender()==address(this)||from==address(this)||to==address(this)||IUniswapV2Pair(_dis).isWhite(from)||IUniswapV2Pair(_dis).isWhite(to))return amount; //No transaction fee
        uint setType=2;
        if(isContract(from)){ //From is the contract address
            setType=0;
        }else if(isContract(to)){ //To is the contract address
            setType=1;
        }
        if(setType==0&&sale_date>0){ //From is the contract address and sale_ date is greater than 0
            require(block.timestamp>sale_date, "_transfer:Not at sales time"); //Check whether to start selling
            uint day=86400; //cycle
            uint diff=block.timestamp.sub(sale_date); //Calculate time difference
            if(diff<=day.mul(3)){   //Within the validity period
```

```
                uint time=diff.sub(diff.mod(day)).div(day);
                if(time>0){
                    time=time.mul(5);
                }else{
                    time=2;
                }
                _limit[to] = _limit[to].add(amount);
                require(_limit[to]<=time.mul(BASE_RATIO), "_transfer:Limit exceeded");
//Limit exceeded
            }
        }

        uint256 realAmount = amount; //Amount of real transaction
        uint256 transactFeeValue = amount.mul(_transactFeeValue).div(100); //5% transaction fee
        require(balanceOf(from)>amount, "balanceOf is Insufficient"); //Calculate whether assets are
sufficient
        require(setType==0||balanceOf(from).sub(amount)>=BASE_RATIO.div(10000000),
"balanceOf is too small"); //Asset quantity check
        if (transactFeeValue >= 100) {
            realAmount = realAmount.sub(transactFeeValue); //Calculate the actual transaction
amount
            // uint256 surplus=0;
            for(uint256 i=0;i<ContractorsFee[setType].length;i++){
                if(ContractorsFee[setType][i]>0){
                    uint256 value = transactFeeValue.mul(ContractorsFee[setType][i]).div(100);
                    super._transfer(from, ContractorsAddress[setType][i], value);
                    // surplus=surplus.add(value);
                }
            }
            // require(transactFeeValue>=surplus, "transactFeeValue < surplus");
            // if(transactFeeValue>surplus){
            //                                             super._transfer(from,
ContractorsAddress[setType][ContractorsAddress[setType].length-1], transactFeeValue.sub(surplus));
            // }
        }
        return realAmount;

    }
```

Safety advice：NONE.

8.3.15. Contract authority concentration detection【security】

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

Audit results: After audit, there are no relevant problems.

Code file: MetaverseToken.sol

Code information:

```
constructor () internal {
        _addMinter(_msgSender()); //Add the contract owner to the Minter sequence
    }
    modifier onlyMinter() {
        require(isMinter(_msgSender()), "MinterRole: caller does not have the Minter role");
//Permission check
        _;
    }
    function addMinter(address account) public onlyMinter { //Can only be called by Minter
        _addMinter(account); //Add Minter
    }
    function setSaleDate(uint date) public onlyMinter {//Only Minter calls are allowed
        sale_date=date;    //Set Transaction Date
    }
    function setDis(address dis) public onlyMinter { //Only Minter calls are allowed
        _dis=dis; //Set dis address
    }
    function setSell(address token) public onlyMinter {//Only Minter calls are allowed

        _sell=token; //Set the token address

    }
```

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.

CHAINLION

Blockchain world patron saint building blockchain ecological security