KNOWNSEC

Degree of secrets ★ | Project team open

Smart contract audit report

Security status

Security

★ ★ ★ ★ ★

Chief test Officer:

Imprint

content	time	version
Write document ntation	20200915	V1.0

文档信息

name	version	number	Confidentiality level
Smart contract audit report	V1.0	【MINEABLE-ZNHY-20200915】	Project team public

statement

Chuangyu only issues this report for facts that have occurred or existed before the issuance of this report, and assumes corresponding responsibilities for this. For facts that will happen or exist in the future, Chuangyu cannot judge the security status of its smart contracts, nor is it liable for them. The security audit analysis and other content made in this report are only based on the documents and information provided by the information provider to Chuangyu as of the issuance of this report. Chuangyu hypothesis: There is no missing, tampered, deleted or concealed information in the provided information. If the provided information is missing, tampered with, deleted, concealed or reflected in the actual situation, Chuangyu shall not be liable for any losses and adverse effects caused thereby

table of Contents

1. Summary

The effective test time of this report is from September 15, 2020 to September 15, 2020. During this period, the security and standardization of the Mineable smart contract code will be audited and used as the statistical basis for the report.

In this test, I know that Chuangyu engineers conducted a comprehensive analysis of the common vulnerabilities of smart contracts (see Chapter 3) and found that there was a problem of issuing additional tokens. This problem needs to be determined according to the requirements of the exchange, so the comprehensive evaluation is passed .

The results of this smart contract security audit:

Since this testing process is carried out in a non-production environment, all codes are up-to-date backups, and the testing process is communicated with the relevant interface person, and relevant testing operations are carried out under the controllable operational risk to avoid production during the testing process Operational risk, code security risk.

Target information of this test:

Module name	
Token name	Mineable
Code type	Token code
Code language	solidity

Contract document and hash:

Contract documents	MD5
Address.sol	0a13afb48ce5f7188b64f139ec5163a2
Context.sol	e888688d9ce6d1f75b85f66f1201a356
ERC20.sol	b61fca0f3bb24304dcc0dd537c76045f

ERC20Detailed.sol	43bd7d5eec7af28b7a041b5db6e75adc
IERC20.sol	fbdf6eeb1060faefa616502a611d2f34

IMineableToken.sol	7473aee65eb9c7f7244c12c791163bed
MineableToken.sol	4c33ecc4c3b1b334653043fdcc413f9f
Ownable.sol	f83a73708c1c6152846acdfef4759c27
SafeMath.sol	a2f1cb6f9cb39b289d3b78faad8d6c54

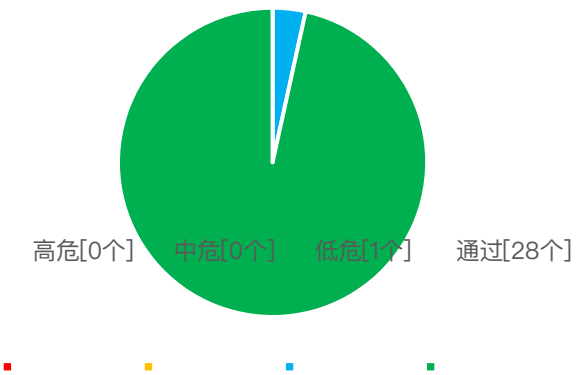
2. Code vulnerability analysis

2.1. Vulnerability level distribution

The risk of this vulnerability is counted by level:

Vulnerability risk level statistics table			
High risk	In dan ger	Low risk	Pas sed
0	0	1	28

Risk level distribution map



2.2. Summary of audit results

Audit results			
Test items	Test content	status	description
Smart contract	Reentry attack detection	Passed	After testing, there is no such safety problem.
	Replay attack detection	Passed	After testing, there is no such safety problem.
	Rearrangement attack detection	Passed	After testing, there is no such safety problem.
	Numerical overflow detection	Passed	After testing, there is no such safety problem.
	Arithmetic accuracy error	Passed	After testing, there is no such safety problem.
	Access control defect detection	Passed	After testing, there is no such safety problem.
	tx.progin authentication	Passed	After testing, there is no such safety problem.
	call injection attack	Passed	After testing, there is no such safety problem.
	Return value call verification	Passed	After testing, there is no such safety problem.
	Uninitialized storage pointer	Passed	After testing, there is no such safety problem.
	Wrong use of random number detection	Passed	After testing, there is no such safety problem.
	Transaction order dependency detection	Passed	After testing, there is no such safety problem.
	Denial of service attack detection	Passed	After testing, there is no such safety problem.
	Logical design defect detection	Passed	After testing, there is no such safety problem.
	Fake recharge vulnerability detection	Passed	After testing, there is no such safety problem.

	Additional token issuance vulnerability detection	Low risk (Passed)	After testing, the code has the function of issuing additional tokens, but because it depends on the requirements of the exchange, the comprehensive assessment is passed.
	Frozen account bypass detection	Passed	After testing, there is no such safety problem.
	Compiler version security	Passed	After testing, there is no such safety problem.
	Not recommended encoding	Passed	After testing, there is no such safety problem.
	Redundant code	Passed	After testing, there is no such safety problem.
	Use of safe arithmetic library	Passed	After testing, there is no such safety problem.

	The use of require/assert use	Passed	After testing, there is no such safety problem.
	gas consumption	Passed	After testing, there is no such safety problem.
	Use of fallback function	Passed	After testing, there is no such safety problem.
	owner permission control	Passed	After testing, there is no such safety problem.
	Low-level function safety	Passed	After testing, there is no such safety problem.
	Variable coverage	Passed	After testing, there is no such safety problem.
	Timestamp dependent attack	Passed	After testing, there is no such safety problem.
	Unsafe interface use	Passed	After testing, there is no such safety problem.

3. Analysis of code audit results

3.1. Reentry attack detection【Passed】

Re-entry vulnerability is the most famous Ethereum smart contract vulnerability, which once led to the fork of Ethereum (TheDAO hack).

The call.value() function in Solidity consumes all the gas it receives when it is used to send Ether. When the call.value() function to send Ether occurs before the actual reduction of the sender's account balance, There is a risk of reentry attacks.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

3.2. Replay attack detection【Passed】

If the contract involves the need for entrusted management, attention should be paid to the non-reusability of verification to avoid replay attacks

In the asset management system, there are often cases of entrusted management. The principal assigns assets to the trustee for management, and the principal pays a certain fee to the trustee. This business scenario is also common in smart contracts.

Detection result: After detection, the smart contract does not use the call function, and this vulnerability does not exist.

Safety advice: None.

3.3. Rearrangement attack detection【Passed】

A rearrangement attack refers to a miner or other party trying to "compete" with smart contract participants by inserting their own information into a list or mapping (mapping), so that the attacker has the opportunity to store their own information in the contract in.

Test result: After testing, there are no related vulnerabilities in the smart contract code.

Safety advice: None.

3.4. Numerical overflow detection【Passed】

The arithmetic problems in smart contracts refer to integer overflow and integer underflow.

Solidity can handle up to 256-bit numbers (2^256-1). If the maximum number increases by 1, it will overflow to 0. Similarly, when the number is an unsigned type, 0 minus 1 will underflow to get the maximum digital value.

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

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

3.5. Arithmetic accuracy error [passed]

As a programming language, Solidity has data structure design similar to ordinary programming languages, such as variables, constants, functions, arrays, functions, structures, etc. There is also a big difference between

Solidity and ordinary programming languages–Solidity does not float. Point type, and all the numerical calculation results of Solidity will only be integers, there will be no decimals, and it is not allowed to define decimal type data. Numerical calculations in the contract are indispensable, and the design of numerical calculations may cause relative errors. For example, the same–level calculations: 5/2*10=20, and 5*10/2=25, resulting in errors, which are larger in data The error will be larger and more obvious.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

3.6. Access control detection【Passed】

。

Different functions in the contract should set reasonable permissions

Check whether each function in the contract correctly uses keywords such as public and private for visibility modification, check whether the contract is correctly defined and use modifier to restrict access to key functions to avoid problems caused by unauthorized access.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: none

3.7. tx.origin authentication [passed]

tx.origin is a global variable of Solidity that traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in a smart contract makes the contract vulnerable to attacks like phishing.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

3.8. call injection attack [passed]

When the call function is called, strict permission control should be done, or the function called by call should be written dead.

Detection result: After detection, the smart contract does not use the call function, and this vulnerability does not exist.

Safety advice: None.

3.9. Return value call verification [Passed]

This problem mostly occurs in smart contracts related to currency transfer, so it is also called silent failed delivery or unchecked delivery.

There are transfer(), send(), call.value() and other currency transfer methods in Solidity, which can all be used to send Ether to an address. The difference is: When the transfer fails, it will be thrown and the state will be rolled back; Only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when send fails; only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when call.value fails to be sent; all available gas will be passed for calling (can be By passing in the gas_value parameter to limit), it cannot effectively prevent reentry attacks.

If the return value of the above send and call.value coin transfer functions is not checked in the code, the contract will continue to execute the following code, which may cause unexpected results due to the failure of Ether sending.

Test result: After testing, the security problem does

not exist in the smart contract code.

Safety advice: None.

3.10. Uninitialized storage pointer [Passed]

In solidity, a special data structure is allowed to be a struct structure, and local variables in a function are stored in storage or memory by default.

The existence of storage (memory) and memory (memory) are two different concepts. Solidity allows pointers to point to an uninitialized reference, while uninitialized local storage will cause variables to point to other storage variables, leading to variable coverage, or even more serious As a consequence, you should avoid initializing struct variables in functions during development.

Test result: After testing, the smart contract code does not use structure, and there is no such problem.

Safety advice: None.

3.11. Wrong use of random numbers [Passed]

Smart contracts may need to use random numbers. Although the functions and variables provided by Solidity can access values that are obviously unpredictable, such as block.number and block.timestamp, they are usually more public than they appear or are affected by miners, that is These random numbers are predictable to a certain extent, so malicious users can usually copy it and rely on its unpredictability to attack the function.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

3.12. Transaction order depends on【Passed】

Since miners always obtain gas fees through codes that represent externally owned addresses (EOA), users can specify higher fees for faster transactions. Since the Ethereum blockchain is public, everyone can see the content of other people's pending transactions. This means that if a user submits a valuable solution, a malicious user can steal the solution and copy its transaction at a higher fee to preempt the original solution.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: none

3.13. Denial of Service Attack【Passed】

In the world of Ethereum, denial of service is fatal, and a smart contract that has suffered this type of attack may never be able to return to its normal working state. There may be many reasons for the denial of service of a smart contract, including malicious behavior as a transaction receiver, artificially increasing the gas required for computing functions to cause gas exhaustion, abuse of access control to access private components of the smart contract, use of confusion and negligence, etc. Wait.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

3.14. Logical design defect【Passed】

Check the security issues related to business design in the smart contract code.Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

3.15. Fake recharge loophole【Passed】

The transfer function of the token contract uses if to check the balance of the transfer initiator (msg.sender)
Judgment method, when balances[msg.sender] <value, enter the else logic part and return false, and finally no exception is thrown. We believe that only if/else is a mild judgment method in sensitive function scenarios such as transfer. Not rigorous coding method.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

3.16. Vulnerabilities in additional token issuance [low

risk]

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

Detection result: After detection, the security problem exists in the smart contract code.

```
//ERC20.sol
function _mint(address account, uint256 amount) internal {//knownsec// 增发代币
        require(account != address(0), "ERC20: mint to the zero address");//knownsec// 校验地址不为 0
        totalSupply = _totalSupply.add(amount);
        _balances[account] = _balances[account].add(amount);
        emit Transfer(address(0), account, amount);
}
//MineableToken.sol
function mint(address to, uint256 amount) public {
        require(minters[msg.sender] != address(0), "not minter");//knownsec// 校验是否为矿工
        mint(to, amount);
}
```

Security Recommendations: This issue is not a security issue, but some exchanges will restrict the use of additional issuance functions. The specific situation depends on the requirements of the exchange.

3.17. Freeze account bypass【Passed】

In the source check token contract, when the token is transferred, whether there is an operation of whether the unverified token account, the originating account, and the target account are frozen.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

3.18. Compiler version security [Passed]

Check whether a safe compiler version is used in the contract code implementation

Test result: After testing, the smart contract code has a compiler version 0.5.8 or higher, and there is no such security issue.

Safety advice: None.

3.19. Unrecommended encoding method [Passed]

Check whether there is an encoding method that is not officially recommended or abandoned in the contract code implementation

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: none。

3.20. Redundant code【Passed】

Check whether the contract code implementation contains redundant code

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

3.21. The use of safe arithmetic library【Passed】

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

Test result: After testing, the SafeMath safe arithmetic library has been used in the smart contract code, and there is no such security problem.

Safety advice: None.

3.22. Use of require/assert [Passed]

Check the rationality of the use of require and assert statements in the contract code implementation

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

3.23. as consumption【Passed】

Check whether the consumption of gas exceeds the maximum block limit

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

3.24. fallback function use [passed]

Check whether the fallback function is used correctly in the contract code implementation

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: none

3.25. owner permission control [passed]

。Check whether the owner in the contract code implementation has excessive authority. For example, arbitrarily modify other

Account balance, etc.

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: none

3.26. Low-level function security [passed]

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

The execution context of the call function is in the called contract; the execution context of the delegatecall function is in the contract that currently calls the function

Test result: After testing, the security problem does not exist in the smart contract code. Safety advice: None.

3.27. Variable coverage【Passed】

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

Test result: After testing, the security problem does not exist in the smart contract code. Safety advice: none

3.28. Timestamp dependency attack【Passed】

The timestamp of the data block usually uses the local time of the miner, and this time can fluctuate in the range of about 900 seconds. When other nodes accept a new block, only need to verify whether the timestamp is later than

The previous block is within 900 seconds of the local time. A miner can profit from it by setting the timestamp of the block to satisfy the conditions that are beneficial to him as much as possible

Check whether there is a key function that depends on the timestamp in the contract code implementation. Test results: After testing, the smart contract code does not have this security problem.

Safety advice: none

3.29.Unsafe interface usage [Passed]

Check whether unsafe interfaces are used in the contract code implementation

Test result: After testing, the security problem does not exist in the smart contract code.

Safety advice: None.

4. appendix A: Contract code

Source of the test code:

Address.sol

```
pragma solidity ^0.5.9;

/**
 * @dev Collection of functions related to the address type
 */
library Address {//knownsec// OpenZeppelin Address 库
    /**
     * @dev Returns true if `account` is a contract.
     *
     * This test is non-exhaustive, and there may be false-negatives: during the
     * execution of a contract's constructor, its address will be reported as
     * not containing 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.
     */
    function isContract(address account) internal view returns (bool) {//knownsec// 判断地址是否为合约地址,OpenZeppelin 实现方法
        // This method relies in extcodesize, which returns 0 for contracts in
        // construction, since the code is only stored at the end of the
        // constructor execution.

        // 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 != 0x0 && codehash != accountHash);
    }

    /**
     * @dev Converts an `address` into `address payable`. Note that this is
     * simply a type cast: the actual underlying value is not changed.
     *
     * _Available since v2.4.0._
     */
    function toPayable(address account) internal pure returns (address payable) {
        return address(uint160(account));
    }

    /**
     * @dev Replacement for Solidity's `transfer`: sends `amount` wei to
     * `recipient`, forwarding all available gas and reverting on errors.
     *
     * https://eips.ethereum.org/EIPS/eip-1884[EIP1884] increases the gas cost
     * of certain opcodes, possibly making contracts go over the 2300 gas limit
     * imposed by `transfer`, making them unable to receive funds via
     * `transfer`. {sendValue} removes this limitation.
     *
     * https://diligence.consensys.net/posts/2019/09/stop-using-soliditys-transfer-now/[Learn more].
     *
     * IMPORTANT: because control is transferred to `recipient`, care must be
     * taken to not create reentrancy vulnerabilities. Consider using
     * {ReentrancyGuard} or the
     * https://solidity.readthedocs.io/en/v0.5.11/security-considerations.html#use-the-checks-effects-interactions-pattern[checks-effects-interactions pattern].
     *
     * _Available since v2.4.0._
     */
    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");

        // solhint-disable-next-line avoid-call-value
        (bool success, ) = recipient.call.value(amount)("");
        require(success, "Address: unable to send value, recipient may have reverted");
    }
}
```

Context.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.5.9;
```

```
/*
 * @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.
 */
contract Context {//knownsec// 上下文属性
    // Empty internal constructor, to prevent people from mistakenly deploying
    // an instance of this contract, which should be used via inheritance.
    constructor () internal { }
    // solhint-disable-previous-line no-empty-blocks

    function _msgSender() internal view returns (address payable)
        { return msg.sender;
    }

    function _msgData() internal view returns (bytes memory) {
        this; // silence state mutability warning without generating bytecode - see
https://github.com/ethereum/solidity/issues/2691
        return msg.data;
    }
}
```

ERC20.sol

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

pragma solidity ^0.5.9;

import "./Context.sol";
import "./IERC20.sol";
import "./SafeMath.sol";
import "./Address.sol";
import "./Ownable.sol";

/**
 * @dev Implementation of the {IERC20} interface.
 *
 * This implementation is agnostic to the way tokens are created. This means
 * that a supply mechanism has to be added in a derived contract using {_mint}.
 * For a generic mechanism see {ERC20Mintable}.
 *
 * TIP: For a detailed writeup see our guide
 * https://forum.zeppelin.solutions/t/how-to-implement-erc20-supply-mechanisms/226[How
 * to implement supply mechanisms].
 *
 * We have followed general OpenZeppelin guidelines: functions revert instead
 * of returning `false` on failure. This behavior is nonetheless conventional
 * and does not conflict with the expectations of ERC20 applications.
 *
 * Additionally, an {Approval} event is emitted on calls to {transferFrom}.
 * This allows applications to reconstruct the allowance for all accounts just
 * by listening to said events. Other implementations of the EIP may not emit
 * these events, as it isn't required by the specification.
 *
 * Finally, the non-standard {decreaseAllowance} and {increaseAllowance}
 * functions have been added to mitigate the well-known issues around setting
 * allowances. See {IERC20-approve}.
 */
contract ERC20 is Context, IERC20 {//knownsec// ERC20 代币标准实现,继承自 Context、IERC20
    using SafeMath for uint256;

    mapping (address => uint256) private _balances;

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

    uint256 private _totalSupply;

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

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

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

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

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

    /**
     * @dev See {IERC20-transferFrom}.
     *
     * Emits an {Approval} event indicating the updated allowance. This is not
     * required by the EIP. See the note at the beginning of {ERC20};
     *
     * Requirements:
     * - `sender` and `recipient` cannot be the zero address.
     * - `sender` must have a balance of at least `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);
        _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "ERC20: transfer
amount exceeds allowance"));
        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));
        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"));
        return true;
```

```
}

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

    _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount exceeds balance");
    _balances[recipient] = _balances[recipient].add(amount);
    emit Transfer(sender, recipient, amount);
}

/** @dev Creates `amount` tokens and assigns them to `account`, increasing
 * the total supply.
 *
 * Emits a {Transfer} event with `from` set to the zero address.
 *
 * Requirements
 *
 * - `to` cannot be the zero address.
 */
function _mint(address account, uint256 amount) internal {//knownsec// 增发代币
    require(account != address(0), "ERC20: mint to the zero address");//knownsec// 校验地址不为 0
    _totalSupply = _totalSupply.add(amount);
    _balances[account] = _balances[account].add(amount);
    emit Transfer(address(0), account, amount);
}

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

    _balances[account] = _balances[account].sub(amount, "ERC20: burn amount exceeds balance");
    _totalSupply = _totalSupply.sub(amount);
    emit Transfer(account, address(0), amount);
}

/**
 * @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");
    require(spender != address(0), "ERC20: approve to the zero address");

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

/**
 * @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 {
        _burn(account, amount);
        _approve(account, _msgSender(), _allowances[account][_msgSender()].sub(amount, "ERC20: burn
amount exceeds allowance"));
    }
}
```

ERC20Detailed.sol

```
pragma solidity ^0.5.9;

import "./IERC20.sol";

/**
 * @dev Optional functions from the ERC20 standard.
 */
contract ERC20Detailed is IERC20 {//knownsec// ERC20 补充细节信息,继承自 IERC20
    string public name;
    string public symbol;
    uint8 public decimals;

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

IERC20.sol

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

pragma solidity ^0.5.9;

/**
 * @dev Interface of the ERC20 standard as defined in the EIP.
 */
interface IERC20 {//knownsec// ERC20 代币标准接口
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

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

    /**
     * @dev Moves `amount` tokens from the caller's account to `recipient`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, uint256 amount) external returns (bool);

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

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

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

    /**
     * @dev Moves `amount` tokens from `sender` to `recipient` using the
     * allowance mechanism. `amount` is then deducted from the caller's
     * allowance.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

    /**
     * @dev 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. This is the value {ERC20} uses, unless {_setupDecimals} is
     * called.
     *
     * 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() external view returns (uint8);


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

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

IMineableToken.sol

```
pragma solidity ^0.5.9;

interface IMineableToken {//knownsec// MineableToken 接口
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

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

    /**
     * @dev Moves `amount` tokens from the caller's account to `recipient`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, uint256 amount) external returns (bool);

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

    /**
     * @dev Sets `amount` as the allowance of `spender` over the caller's tokens.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * IMPORTANT: Beware that changing an allowance with this method brings the risk
     * that someone may use both the old and the new allowance by unfortunate
     * transaction ordering. One possible solution to mitigate this race
```

```
     * condition is to first reduce the spender's allowance to 0 and set the
     * desired value afterwards:
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     *
     * Emits an {Approval} event.
     */
    function approve(address spender, uint256 amount) external returns (bool);

    /**
     * @dev Moves `amount` tokens from `sender` to `recipient` using the
     * allowance mechanism. `amount` is then deducted from the caller's
     * allowance.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

    /**
     * @dev 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. This is the value {ERC20} uses, unless {_setupDecimals} is
     * called.
     *
     * 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() external view returns (uint8);


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

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

    function transferOwnership(address newOwner) external;

    function mint(address to, uint256 amount) external ;
}
```

MineableToken.sol

```
pragma solidity ^0.5.9;

import "./ERC20.sol";
import "./ERC20Detailed.sol";
import "./IMineableToken.sol";

contract MineableToken is IMineableToken, ERC20, ERC20Detailed, Ownable {//knownsec// MineableToken 代 币合约,继承自 IMineableToken、ERC20、ERC20Detailed、Ownable

    constructor(address _owner, string memory _name, string memory _symbol, uint8 _decimals)
    public ERC20Detailed(_name, _symbol, _decimals) Ownable(_owner){}

    mapping(address=>address) public minters;//knownsec// 矿 工

    function setMinter(address _minter) public onlyOwner {//knownsec// 设置矿工,仅 owner 调用
        minters[_minter] = _minter;
    }

    function removeMinter(address _minter) public onlyOwner {//knownsec// 移除矿工,仅 owner 调用
        delete minters[_minter];
    }

    function mint(address to, uint256 amount) public {
        require(minters[msg.sender] != address(0), "not minter");//knownsec// 校验是否为矿工
        _mint(to, amount);
    }

    function signature() external pure returns (string memory) {//knownsec// 签 名
        return "provided by Seal-SC / www.sealsc.com";
    }
```

```
}
```

Ownable.sol

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

pragma solidity ^0.5.9;

import "./Context.sol";

/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
contract Ownable is Context {//knownsec// 所有权合约,继承自 Context
    address private _owner;

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

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor (address owner_) internal {
        _owner = owner_;
        emit OwnershipTransferred(address(0), _owner);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view returns (address)
        { return _owner;
    }

    /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(isOwner(), "Ownable: caller is not the owner");
        _;
    }

    /**
     * @dev Returns true if the caller is the current owner.
     */
    function isOwner() public view returns (bool)
        { return _msgSender() == _owner;
    }

    /**
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public onlyOwner {//knownsec// 放弃所有权
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public onlyOwner {//knownsec// 转移所有权,仅 owner 调用
        _transferOwnership(newOwner);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     */
    function _transferOwnership(address newOwner) internal {
        require(newOwner != address(0), "Ownable: new owner is the zero address");//knownsec// 校验地址不为 0
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}
```

SafeMath.sol

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

pragma solidity ^0.5.9;

/**
 * @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 {//knownsec// 安全算数库
    /**
     * @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.
     */
    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.
     */
    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256)
        { 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.
     */
    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256)
        { require(b != 0, errorMessage);
        return a % b;
    }
}
```

appendix B：Vulnerability risk rating standard

Smart contract vulnerability rating standards	
Vulnerability rating	Vulnerability rating description
High-risk vulnerabilities	Vulnerabilities that can directly cause the loss of token contracts or user funds, such as: value overflow loopholes that can cause the value of tokens to zero, fake recharge loopholes that can cause exchanges to lose tokens, and can cause contract accounts to lose ETH or tokens. Access loopholes, etc.; Vulnerabilities that can cause loss of ownership of token contracts, such as: access control defects of key functions, call injection leading to bypassing of access control of key functions, etc.; Vulnerabilities that can cause the token contract to not work properly, such as: caused by sending ETH to a malicious address Denial of service vulnerabilities, denial of service vulnerabilities caused by gas exhaustion.
Medium-Dangerous Vulnerability	High-risk vulnerabilities that require a specific address to trigger, such as a value that can be triggered by the owner of a token contract Overflow vulnerabilities, etc.; access control defects of non-critical functions, logical design defects that cannot cause direct capital losses, etc.
Low-risk vulnerabilities	Vulnerabilities that are difficult to trigger, and vulnerabilities with limited damage after triggering, such as a large amount of ETH or tokens. Numerical overflow vulnerabilities that can be triggered, vulnerabilities that attackers cannot directly profit after triggering numerical overflow, and transaction sequence triggered by specifying high gas depends on risk, etc.

appendixC：Introduction to vulnerability testing tools

6.1 Manticore

Manticore is a symbolic execution tool for analyzing binary files and smart contracts. Manticore includes a symbolic Ethereum Virtual Machine (EVM), an EVM disassembler/assembler, and a convenient interface for automatic compilation and analysis of Solidity. It also integrates Ethersplay, the Bit of Traits of Bits visual disassembler for EVM bytecode, for visual analysis. Like binary files, Manticore provides a simple command line interface and a Python API for analyzing EVM bytecode

6.2. Oyente

Oyente is a smart contract analysis tool. Oyente can be used to detect common bugs in smart contracts, such as reentrancy, transaction sequencing dependencies, and so on. What's more convenient is that Oyente's design is modular, so this allows advanced users to implement and insert their own detection logic to check custom attributes in their contracts.

6.3. securify.sh

Securify can verify common security issues of Ethereum smart contracts, such as disorder of transactions and lack of input verification. It analyzes all possible execution paths of the program while fully automated. In addition, Securify also has a specific language for specifying vulnerabilities, which makes Securify can keep an eye on current security and other reliability issues.

6.4. Echidna

Echidna is a Haskell library designed for fuzzing EVM code

6.5. MAIAN

MAIAN is an automated tool for finding vulnerabilities in Ethereum smart

contracts. Maian processes the bytecode of the contract and tries to establish a series of transactions to find and confirm errors.

6.6. ethersplay

ethersplay is an EVM disassembler, which contains relevant analysis tools.

6.7. ida-evm

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

6.8. Remix-ide

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

6.9 Know the special toolkit for Chuangyu penetration testers

Know Chuangyu Penetration Tester's special toolkit is developed, collected and used by Know Chuangyu penetration test engineers. It contains batch automatic test tools dedicated to testers, self-developed tools, scripts or utilization tools, etc.

Beijing Knownsec Information Tech. CO., LTD.

Telephone number	+86(10)400 060 9587
E-mail	sec@knownsec.com
Official website	www.knowsec.com
Address	2509, block T2-B, Wangjing SOHO, Chaoyang District, Beijing