KNOWNSEC

Smart Contract Security Audit Report

Audit Results

PASS

★ ★ ★ ★ ★

Version description

Revised man	Revised content	Revised time	version	Reviewer
Yifeng Luo	Document creation and editing	2020/10/24	V1.0	Haojie Xu

Document information

Document Name	Audit Date	Audit results	Privacy level	Audit enquiry telephone
UquidCoin Smart Contract Security Audit Report	2020/10/24	PASS	Open project team	+86 400-060-9587

Copyright statement

Company statement

Catalog

1. Review

The effective testing time of this report is from October 23, 2020 to October 24, 2020. During this period, the Knownsec engineers audited the safety and regulatory aspects of UquidCoin smart contract code.

In this test, engineers comprehensively analyzed common vulnerabilities of smart contracts (Chapter 3) and It was not discovered medium-risk or high-risk vulnerability,so it's evaluated as **pass**.

The result of the safety auditing: Pass

Since the test process is carried out in a non-production environment, all the codes are the latest backups. We communicates with the relevant interface personnel, and the relevant test operations are performed under the controllable operation risk to avoid the risks during the test..

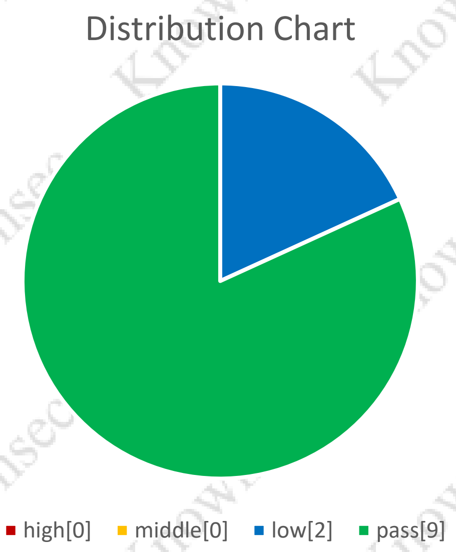
Target information for this test:

Project name	Project content
Token name	UquidCoin
Code type	Token code
Code language	solidity
Code address	https://etherscan.io/address/0x8806926Ab68EB5a7b909DcAf6FdBe5d93271D6e2#code

2. Analysis of code vulnerability

2.1. Distribution of vulnerability Levels

Vulnerability statistics			
high	Middle	low	pass
0	0	2	9



2.2. Audit result summary

Other unknown security vulnerabilities are not included in the scope of this audit.

Result			
Test project	Test content	status	description
Smart Contract Security Audit	Reentrancy	Pass	Check the call.value() function for security
	Arithmetic Issues	Pass	Check add and sub functions
	Access Control	Pass	Check the operation access control
	Unchecked Return Values For Low Level Calls	Pass	Check the currency conversion method.
	Bad Randomness	Pass	Check the unified content filter
	Transaction ordering dependence	Low risk	Check the transaction ordering dependence
	Denial of service attack detection	Low risk	Check whether the code has a resource abuse problem when using a resource
	Logic design Flaw	Pass	Examine the security issues associated with business design in intelligent contract codes.
	USDT Fake Deposit Issue	Pass	Check for the existence of USDT Fake Deposit Issue
	Adding tokens	Pass	It is detected whether there is a function in the token contract that may increase the total amounts of tokens
	Freezing accounts bypassed	Pass	It is detected whether there is an unverified token source account, an originating account, and whether the target account is frozen.

3. Result analysis

3.1. Reentrancy【Pass】

The Reentrancy attack, probably the most famous Blockchain vulnerability，led to a hard fork of Ethereum.

When the low level call() function sends tokens to the msg.sender address, it becomes vulnerable; if the address is a smart token, the payment will trigger its fallback function with what's left of the transaction gas.

Test results: No related vulnerabilities in smart contract code.

Safety advice：None.

3.2. Arithmetic Issues【Pass】

Also known as integer overflow and integer underflow. Solidity can handle up to 256 digits (2^256-1), The largest number increases by 1 will overflow to 0. Similarly, when the number is an unsigned type, 0 minus 1 will underflow to get the maximum numeric value.

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

Test results: No related vulnerabilities in smart contract code.

Safety advice：None.

3.3. Access Control【Pass】

Access Control issues are common in all programs,Also smart contracts. The famous Parity Wallet smart contract has been affected by this issue.

Test results: No related vulnerabilities in smart contract code.

Safety advice：None.

3.4. Unchecked Return Values For Low Level Calls【Pass】

Also known as or related to silent failing sends, unchecked-send. There are transfer methods such as transfer(), send(), and call.value() in Solidity and can be used to send tokens s to an address. The difference is: transfer will be thrown when failed to send, and rollback; only 2300gas will be passed for call to prevent reentry attacks; send will return false if send fails; only 2300gas will be passed for call to prevent reentry attacks; If .value fails to send, it will return false; passing all available gas calls (which can be restricted by passing in the gas_value parameter) cannot effectively prevent reentry attacks.

If the return value of the send and call.value switch functions is not been checked in the code, the contract will continue to execute the following code,and it may have caused unexpected results due to tokens sending failure.

Test results: No related vulnerabilities in smart contract code.

Safety advice：None.

3.5. Bad Randomness【Pass】

Smart Contract May Need to Use Random Numbers. While Solidity offers functions and variables that can access apparently hard-to-predict values just as block.number and block.timestamp. they are generally either more public than they seem or subject to miners' influence. Because these sources of randomness are to an extent predictable, malicious users can generally replicate it and attack the function relying on its unpredictablility.

Test results: No related vulnerabilities in smart contract code.

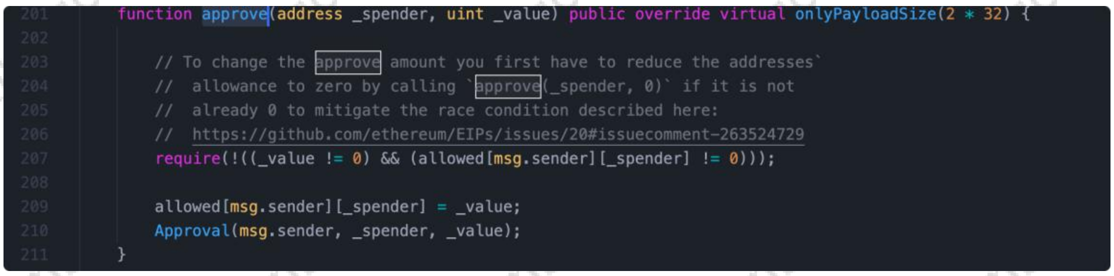
Safety advice：None.

3.6. Transaction ordering dependence【Low risk】

Since miners always get rewarded via gas fees for running code on behalf of externally owned addresses (EOA), users can specify higher fees to have their transactions mined more quickly. Since the blockchain is public, everyone can see the contents of others' pending transactions.

This means if a given user is revealing the solution to a puzzle or other valuable secret, a malicious user can steal the solution and copy their transaction with higher fees to preempt the original solution.

Test results: Having related vulnerabilities in smart contract code.



```
    function approve(address _spender, uint _value) public override virtual onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Safety advice:

1. User A allows the number of user B transfers to be N (N > 0) by calling the approve function;

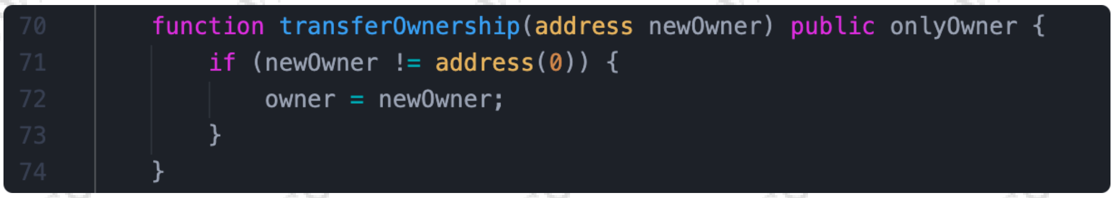
2. After a while, user A decided to change N to M (M > 0), so he called the approve function again;

3. User B quickly calls the transfer from function to transfer the number of N before the second call is processed by the miner. After user A's second call to approve is successful, user B can get the transfer amount of M again. That is, user B obtains the transfer amount of N+M by trading sequence attack.

3.7. Denial of service attack detection 【Low risk】

In the blockchain world, denial of service is deadly, and smart contracts under attack of this type may never be able to return to normal. There may be a number of reasons for a denial of service in smart contracts, including malicious behavior as a recipient of transactions, gas depletion caused by artificially increased computing gas, and abuse of access control to access the private components of the intelligent contract. Take advantage of confusion and neglect, etc.

Test results: After testing, there is an error in the smart contract code because of the user's owner access control strategy, which will cause the user to permanently lose control.



```
    function transferOwnership(address newOwner) public onlyOwner {
        if (newOwner != address(0)) {
            owner = newOwner;
        }
    }
```

Safety advice:　For the conversion of control authority, attention should be paid to the determination of user ownership to avoid permanent loss of control.

3.8. **Logical design Flaw【Pass】**

Detect the security problems related to business design in the contract code.

Test results: No related vulnerabilities in smart contract code.

Safety advice: None.

3.9. **USDT Fake Deposit Issue【Pass】**

In the transfer function of the token contract, the balance check of the transfer initiator (msg.sender) is judged by if. When balances[msg.sender] < value, it enters the else logic part and returns false, and finally no exception is thrown. We believe that only the modest judgment of if/else is an imprecise coding method in the sensitive function scene such as transfer.

Detection results: No related vulnerabilities in smart contract code.

Safety advice: None.

3.10. **Adding tokens【Pass】**

It is detected whether there is a function in the token contract that may increase the total amount of tokens after the total amount of tokens is initialized.

Detection results: No related vulnerabilities in smart contract code.

Safety advice: None.

3.11. Freezing accounts bypassed【Pass】

In the token contract, when transferring the token, it is detected whether there is an unverified token source account, an originating account, and whether the target account is frozen.

Detection results: No related vulnerabilities in smart contract code.

Safety advice: None.

4. Appendix A: Contract code

```
    /**
     *Submitted for verification at Etherscan.io on 2020-10-03
    */

    // SPDX-License-Identifier: MIT

    pragma solidity ^0.6.12;

    /**
     * @title SafeMath
     * @dev Math operations with safety checks that throw on error
     */
    library SafeMath {
        function mul(uint256 a, uint256 b) internal pure returns (uint256) {
            if (a == 0) {
                return 0;
            }
            uint256 c = a * b;
            assert(c / a == b);
            return c;
        }

        function div(uint256 a, uint256 b) internal pure returns (uint256) {
            // assert(b > 0); // Solidity automatically throws when dividing by 0
            uint256 c = a / b;
            // assert(a == b * c + a % b); // There is no case in which this doesn't hold
            return c;
        }

        function sub(uint256 a, uint256 b) internal pure returns (uint256) {
            assert(b <= a);
            return a - b;
        }

        function add(uint256 a, uint256 b) internal pure returns (uint256) {
            uint256 c = a + b;
            assert(c >= a);
            return c;
        }
    }

    /**
     * @title Ownable
     * @dev The Ownable contract has an owner address, and provides basic authorization
control
     * functions, this simplifies the implementation of "user permissions".
     */
    contract Ownable {
        address public owner;

        /**
          * @dev The Ownable constructor sets the original `owner` of the contract to the
sender
          * account.
          */
        constructor() public {
            owner = msg.sender;
        }

        /**
          * @dev Throws if called by any account other than the owner.
          */
        modifier onlyOwner() {
            require(msg.sender == owner);
            _;
        }

        /**
        * @dev Allows the current owner to transfer control of the contract to a newOwner.
        * @param newOwner The address to transfer ownership to.
        */
        function transferOwnership(address newOwner) public onlyOwner {
            if (newOwner != address(0)) {
                owner = newOwner;
            }
        }

    }

    /**
     * @title ERC20Basic
     * @dev Simpler version of ERC20 interface
     * @dev see https://github.com/ethereum/EIPs/issues/20
     */
    abstract contract ERC20Basic {
        uint public _totalSupply;
        function totalSupply() public virtual view returns (uint);
        function balanceOf(address who) public virtual view returns (uint);
        function transfer(address to, uint value) public virtual;
        event Transfer(address indexed from, address indexed to, uint value);
    }

    /**
     * @title ERC20 interface
     * @dev see https://github.com/ethereum/EIPs/issues/20
     */
    abstract contract ERC20 is ERC20Basic {
        function allowance(address owner, address spender) public virtual view returns
(uint);
        function transferFrom(address from, address to, uint value) public virtual;
        function approve(address spender, uint value) public virtual;
        event Approval(address indexed owner, address indexed spender, uint value);
    }

    /**
     * @title Basic token
     * @dev Basic version of StandardToken, with no allowances.
     */
    abstract contract BasicToken is Ownable, ERC20Basic {
        using SafeMath for uint;

        mapping(address => uint) public balances;

        // additional variables for use if transaction fees ever became necessary
        uint public basisPointsRate = 0;
        uint public maximumFee = 0;

        /**
        * @dev Fix for the ERC20 short address attack.
        */
        modifier onlyPayloadSize(uint size) {
            require(!(msg.data.length < size + 4));
            _;
        }

        /**
        * @dev transfer token for a specified address
        * @param _to The address to transfer to.
        * @param _value The amount to be transferred.
        */
        function transfer(address _to, uint _value) public override virtual
onlyPayloadSize(2 * 32) {
            uint fee = (_value.mul(basisPointsRate)).div(10000);
            if (fee > maximumFee) {
                fee = maximumFee;
            }
            uint sendAmount = _value.sub(fee);
            balances[msg.sender] = balances[msg.sender].sub(_value);
            balances[_to] = balances[_to].add(sendAmount);
            if (fee > 0) {
                balances[owner] = balances[owner].add(fee);
                emit Transfer(msg.sender, owner, fee);
            }
            emit Transfer(msg.sender, _to, sendAmount);
        }

        /**
        * @dev Gets the balance of the specified address.
        * @param _owner The address to query the the balance of.
        * @return balance uint An uint representing the amount owned by the passed address.
        */
        function balanceOf(address _owner) public override virtual view returns (uint
balance) {
            return balances[_owner];
        }

    }

    /**
     * @title Standard ERC20 token
     *
     * @dev Implementation of the basic standard token.
     * @dev https://github.com/ethereum/EIPs/issues/20
     * @dev Based oncode by FirstBlood:
https://github.com/Firstbloodio/token/blob/master/smart_contract/FirstBloodToken.sol
     */
    abstract contract StandardToken is BasicToken, ERC20 {

        mapping (address => mapping (address => uint)) public allowed;

        uint public constant MAX_UINT = 2**256 - 1;

        /**
        * @dev Transfer tokens from one address to another
        * @param _from address The address which you want to send tokens from
        * @param _to address The address which you want to transfer to
        * @param _value uint the amount of tokens to be transferred
        */
        function transferFrom(address _from, address _to, uint _value) public  override
virtual onlyPayloadSize(3 * 32) {
            uint256 _allowance = allowed[_from][msg.sender];

            // Check is not needed because sub(_allowance, _value) will already throw if this
condition is not met
            // if (_value > _allowance) throw;

            uint fee = (_value.mul(basisPointsRate)).div(10000);
            if (fee > maximumFee) {
                fee = maximumFee;
            }
            if (_allowance < MAX_UINT) {
                allowed[_from][msg.sender] = _allowance.sub(_value);
            }
            uint sendAmount = _value.sub(fee);
            balances[_from] = balances[_from].sub(_value);
            balances[_to] = balances[_to].add(sendAmount);
            if (fee > 0) {
                balances[owner] = balances[owner].add(fee);
                emit Transfer(_from, owner, fee);
            }
            emit Transfer(_from, _to, sendAmount);
        }

        /**
        * @dev Approve the passed address to spend the specified amount of tokens on behalf
of msg.sender.
        * @param _spender The address which will spend the funds.
        * @param _value The amount of tokens to be spent.
        */
        function approve(address _spender, uint _value) public override virtual
onlyPayloadSize(2 * 32) {

            // To change the approve amount you first have to reduce the addresses`
            //  allowance to zero by calling `approve(_spender, 0)` if it is not
            //  already 0 to mitigate the race condition described here:
            //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
            require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

            allowed[msg.sender][_spender] = _value;
            Approval(msg.sender, _spender, _value);
        }

        /**
        * @dev Function to check the amount of tokens than an owner allowed to a spender.
        * @param _owner address The address which owns the funds.
        * @param _spender address The address which will spend the funds.
        * @return remaining uint A uint specifying the amount of tokens still available for
the spender.
        */
        function allowance(address _owner, address _spender) public override virtual view
returns (uint remaining) {
            return allowed[_owner][_spender];
        }

    }


    /**
     * @title Pausable
     * @dev Base contract which allows children to implement an emergency stop mechanism.
     */
    contract Pausable is Ownable {
      event Pause();
      event Unpause();

      bool public paused = false;


      /**
       * @dev Modifier to make a function callable only when the contract is not paused.
       */
      modifier whenNotPaused() {
        require(!paused);
        _;
      }

      /**
       * @dev Modifier to make a function callable only when the contract is paused.
       */
      modifier whenPaused() {
        require(paused);
        _;
      }

      /**
       * @dev called by the owner to pause, triggers stopped state
       */
      function pause() onlyOwner whenNotPaused public {
        paused = true;
        Pause();
      }

      /**
       * @dev called by the owner to unpause, returns to normal state
       */
      function unpause() onlyOwner whenPaused public {
        paused = false;
        Unpause();
      }
    }

    abstract contract BlackList is Ownable, BasicToken {

        /////// Getters to allow the same blacklist to be used also by other contracts
(including upgraded Tether) ///////
        function getBlackListStatus(address _maker) external view returns (bool) {
            return isBlackListed[_maker];
        }

        function getOwner() external view returns (address) {
            return owner;
        }

        mapping (address => bool) public isBlackListed;

        function addBlackList (address _evilUser) public onlyOwner {
            isBlackListed[_evilUser] = true;
            AddedBlackList(_evilUser);
        }

        function removeBlackList (address _clearedUser) public onlyOwner {
            isBlackListed[_clearedUser] = false;
            RemovedBlackList(_clearedUser);
        }

        function destroyBlackFunds (address _blackListedUser) public onlyOwner {
            require(isBlackListed[_blackListedUser]);
            uint dirtyFunds = balanceOf(_blackListedUser);
            balances[_blackListedUser] = 0;
            _totalSupply -= dirtyFunds;
            DestroyedBlackFunds(_blackListedUser, dirtyFunds);
        }

        event DestroyedBlackFunds(address _blackListedUser, uint _balance);

        event AddedBlackList(address _user);

        event RemovedBlackList(address _user);

    }

    abstract contract UpgradedStandardToken is StandardToken{
        // those methods are called by the legacy contract
        // and they must ensure msg.sender to be the contract address
        function transferByLegacy(address from, address to, uint value) virtual public;
        function transferFromByLegacy(address sender, address from, address spender, uint
value) virtual public;
        function approveByLegacy(address from, address spender, uint value) virtual public;
    }

    contract UquidCoin is Pausable, StandardToken, BlackList {

        string public name;
        string public symbol;
        uint public decimals;
        address public upgradedAddress;
        bool public deprecated;

        //  The contract can be initialized with a number of tokens
        //  All the tokens are deposited to the owner address
        //
        // @param _balance Initial supply of the contract
        // @param _name Token Name
        // @param _symbol Token symbol
        // @param _decimals Token decimals
        constructor (uint _initialSupply, string memory _name, string memory _symbol, uint
_decimals) public {
            _totalSupply = _initialSupply;
            name = _name;
            symbol = _symbol;
            decimals = _decimals;
            balances[owner] = _initialSupply;
            deprecated = false;
        }

        // Forward ERC20 methods to upgraded contract if this one is deprecated
        function transfer(address _to, uint _value) public override whenNotPaused {
            require(!isBlackListed[msg.sender]);
            if (deprecated) {
                return
UpgradedStandardToken(upgradedAddress).transferByLegacy(msg.sender, _to, _value);
            } else {
                return super.transfer(_to, _value);
            }
        }

        // Forward ERC20 methods to upgraded contract if this one is deprecated
        function transferFrom(address _from, address _to, uint _value) public override
whenNotPaused {
            require(!isBlackListed[_from]);
            if (deprecated) {
                return
UpgradedStandardToken(upgradedAddress).transferFromByLegacy(msg.sender, _from, _to,
_value);
            } else {
                return super.transferFrom(_from, _to, _value);
            }
        }

        // Forward ERC20 methods to upgraded contract if this one is deprecated
        function balanceOf(address who) public override view returns (uint) {
            if (deprecated) {
                return UpgradedStandardToken(upgradedAddress).balanceOf(who);
            } else {
                return super.balanceOf(who);
            }
        }

        // Forward ERC20 methods to upgraded contract if this one is deprecated
        function approve(address _spender, uint _value) public override onlyPayloadSize(2
* 32) {
            if (deprecated) {
                return UpgradedStandardToken(upgradedAddress).approveByLegacy(msg.sender,
_spender, _value);
            } else {
                return super.approve(_spender, _value);
            }
        }

        // Forward ERC20 methods to upgraded contract if this one is deprecated
        function allowance(address _owner, address _spender) public override view returns
(uint remaining) {
            if (deprecated) {
                return StandardToken(upgradedAddress).allowance(_owner, _spender);
            } else {
                return super.allowance(_owner, _spender);
            }
        }

        // deprecate current contract in favour of a new one
        function deprecate(address _upgradedAddress) public onlyOwner {
            deprecated = true;
            upgradedAddress = _upgradedAddress;
            Deprecate(_upgradedAddress);
        }

        // deprecate current contract if favour of a new one
        function totalSupply() public override view returns (uint) {
            if (deprecated) {
                return StandardToken(upgradedAddress).totalSupply();
            } else {
                return _totalSupply;
            }
        }

        // Issue a new amount of tokens
        // these tokens are deposited into the owner address
        //
        // @param _amount Number of tokens to be issued
        function issue(uint amount) public onlyOwner {
            require(_totalSupply + amount > _totalSupply);
            require(balances[owner] + amount > balances[owner]);

            balances[owner] += amount;
            _totalSupply += amount;
            Issue(amount);
        }

        // Redeem tokens.
        // These tokens are withdrawn from the owner address
        // if the balance must be enough to cover the redeem
        // or the call will fail.
        // @param _amount Number of tokens to be issued
        function redeem(uint amount) public onlyOwner {
            require(_totalSupply >= amount);
            require(balances[owner] >= amount);

            _totalSupply -= amount;
            balances[owner] -= amount;
            Redeem(amount);
        }

        function setParams(uint newBasisPoints, uint newMaxFee) public onlyOwner {
            // Ensure transparency by hardcoding limit beyond which fees can never be added
            require(newBasisPoints < 20);
            require(newMaxFee < 50);
            basisPointsRate = newBasisPoints;
            maximumFee = newMaxFee.mul(10**decimals);

            Params(basisPointsRate, maximumFee);
        }

        // Called when new token are issued
        event Issue(uint amount);

        // Called when tokens are redeemed
        event Redeem(uint amount);

        // Called when contract is deprecated
        event Deprecate(address newAddress);

        // Called if contract ever adds fees
        event Params(uint feeBasisPoints, uint maxFee);
    }
```

5. Appendix B: vulnerability risk rating criteria

Smart contract vulnerability rating standard	
Vulnerability rating	Vulnerability rating description
High risk vulnerability	The loophole which can directly cause the contract or the user's fund loss, such as the value overflow loophole which can cause the value of the substitute currency to zero, the false recharge loophole that can cause the exchange to lose the substitute coin, can cause the contract account to lose the ETH or the reentry loophole of the substitute currency, and so on; It can cause the loss of ownership rights of token contract, such as: the key function access control defect or call injection leads to the key function access control bypassing, and the loophole that the token contract can not work properly. Such as: a denial-of-service vulnerability due to sending ETHs to a malicious address, and a denial-of-service vulnerability due to gas depletion.
Middle risk vulnerability	High risk vulnerabilities that need specific addresses to trigger, such as numerical overflow vulnerabilities that can be triggered by the owner of a token contract, access control defects of non-critical functions, and logical design defects that do not result in direct capital losses, etc.
Low risk vulnerability	A vulnerability that is difficult to trigger, or that will harm a limited number after triggering, such as a numerical overflow that requires a large number of ETH or tokens to trigger, and a vulnerability that the attacker cannot directly profit from after triggering a numerical overflow. Rely on risks by specifying the order of transactions triggered by a high gas.

6. Appendix C：Introduction of test tool

6.1. Manticore

Manticore is a symbolic execution tool for analysis of binaries and smart contracts. It discovers inputs that crash programs via memory safety violations. Manticore records an instruction-level trace of execution for each generated input and exposes programmatic access to its analysis engine via a Python API.

6.2. Oyente

Oyente is a smart contract analysis tool that Oyente can use to detect common bugs in smart contracts, such as reentrancy, transaction ordering dependencies, and more. More conveniently, Oyente's design is modular, so this allows advanced users to implement and insert their own detection logic to check for custom attributes in their contracts.

6.3. securify.sh

Securify can verify common security issues with smart contracts, such as transactional out-of-order and lack of input validation. It analyzes all possible execution paths of the program while fully automated. In addition, Securify has a specific language for specifying vulnerabilities. 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 smart contract vulnerabilities. Maian deals with the contract's bytecode and tries to establish a series of transactions to find and confirm errors.

6.6. ethersplay

Ethersplay is an EVM disassembler that contains related 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 build blockchain contracts and debug transactions using the Solidity language.

6.9. Knownsec Penetration Tester Special Toolkit

Knownsec penetration tester special tool kit, developed and collected by Knownsec penetration testing engineers, includes batch automatic testing tools dedicated to testers, self-developed tools, scripts, or utility tools.