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	2020/10/15	V1.0	Haojie Xu

Document information

Document Name	Audit Date	Audit results	Privacy level	Audit enquiry telephone
Stake Smart Contract Security Audit Report	2020/10/15	PASS	Open project team	+86 400-060-9587

Copyright statement

Company statement

Beijing Knownsec Information Technology Co., Ltd. only conducts the agreed safety audit and issued the report based on the documents and materials provided to us by the project party as of the time of this report. We cannot judge the background, the practicality of the project, the compliance of business model and the legality of the project , and will not be responsible for this. The report is for reference only for internal decision-making of the project party. Without our written consent, the report shall not be disclosed or provided to other people or used for other purposes without permission. The report issued by us shall not be used as the basis for any behavior and decision of the third party. We shall not bear any responsibility for the consequences of the relevant decisions adopted by the user and the third party. We assume that as of the time of this report, the project has provided us with no information missing, tampered with, deleted or concealed. If the information provided is missing, tampered, deleted, concealed or reflected in a situation inconsistent with the actual situation, we will not be liable for the loss and adverse impact caused thereby.

Catalog

1. Review

The effective testing time of this report is from October 13, 2020 to October 15, 2020. During this period, the Knownsec engineers audited the safety and regulatory aspects of Xfinance 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..

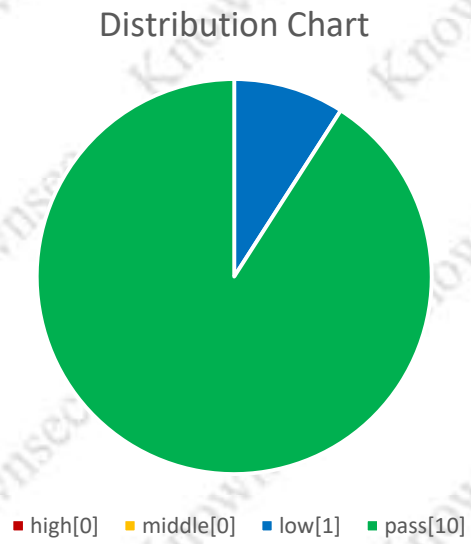
Target information for this test:

Project name	Project content
Token name	Stake
Code type	Token code
Code language	solidity
Code address	https://etherscan.io/address/0x5BEfBB272290dD5b8521D4a938f6c4757742c430#code

2. Analysis of code vulnerability

2.1. Distribution of vulnerability Levels

Vulnerability statistics			
high	Middle	low	pass
0	0	1	10



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	Pass	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.

Detection 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【Pass】

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.

Detection results: No related vulnerabilities in smart contract code.

Safety advice: None.

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.

```
197     function transferOwnership(address payable _newOwner) public onlyOwner {
198         owner = _newOwner;
199         emit OwnershipTransferred(msg.sender, _newOwner);
200     }
201 }
```

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 2020-09-12
*/

/**
 *Submitted for verification at Etherscan.io on 2020-08-29
*/

// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.6.0;

// ----------------------------------------------------------------------------
// 'XFI' Staking smart contract
// ----------------------------------------------------------------------------

// ----------------------------------------------------------------------------
// SafeMath library
// ----------------------------------------------------------------------------

/**
 * @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.
     */
    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;
        }

        function ceil(uint a, uint m) internal pure returns (uint r) {
            return (a + m - 1) / m * m;
        }
    }

    // ----------------------------------------------------------------------------
    // Owned contract
    // ----------------------------------------------------------------------------
    contract Owned {
        address payable public owner;

        event OwnershipTransferred(address indexed _from, address indexed _to);

        constructor() public {
            owner = msg.sender;
        }

        modifier onlyOwner {
            require(msg.sender == owner);
            _;
        }

        function transferOwnership(address payable _newOwner) public onlyOwner {
            owner = _newOwner;
            emit OwnershipTransferred(msg.sender, _newOwner);
        }
    }

    // ----------------------------------------------------------------------------
    // ERC Token Standard #20 Interface
    // ----------------------------------------------------------------------------
    interface IERC20 {
        function totalSupply() external view returns (uint256);
        function balanceOf(address tokenOwner) external view returns (uint256 balance);
        function allowance(address tokenOwner, address spender) external view returns
(uint256 remaining);
        function transfer(address to, uint256 tokens) external returns (bool success);
        function approve(address spender, uint256 tokens) external returns (bool success);
        function transferFrom(address from, address to, uint256 tokens) external returns
(bool success);
        function burnTokens(uint256 _amount) external;
        event Transfer(address indexed from, address indexed to, uint256 tokens);
        event Approval(address indexed tokenOwner, address indexed spender, uint256 tokens);
    }
```

```
    // ----------------------------------------------------------------------------
    // ERC20 Token, with the addition of symbol, name and decimals and assisted
    // token transfers
    // ----------------------------------------------------------------------------
    contract Stake is Owned {
        using SafeMath for uint256;

        address public XFI = 0x5BEfBB272290dD5b8521D4a938f6c4757742c430;

        uint256 public totalStakes = 0;
        uint256 stakingFee = 25; // 2.5%
        uint256 unstakingFee = 25; // 2.5%
        uint256 public totalDividends = 0;
        uint256 private scaledRemainder = 0;
        uint256 private scaling = uint256(10) ** 12;
        uint public round = 1;

        struct USER{
            uint256 stakedTokens;
            uint256 lastDividends;
            uint256 fromTotalDividend;
            uint round;
            uint256 remainder;
        }

        mapping(address => USER) stakers;
        mapping (uint => uint256) public payouts;               // keeps record of each
payout

        event STAKED(address staker, uint256 tokens, uint256 stakingFee);
        event UNSTAKED(address staker, uint256 tokens, uint256 unstakingFee);
        event PAYOUT(uint256 round, uint256 tokens, address sender);
        event CLAIMEDREWARD(address staker, uint256 reward);

        // ------------------------------------------------------------------------
        // Token holders can stake their tokens using this function
        // @param tokens number of tokens to stake
        // ------------------------------------------------------------------------
        function STAKE(uint256 tokens) external {
            require(IERC20(XFI).transferFrom(msg.sender, address(this), tokens), "Tokens
cannot be transferred from user account");

            uint256 _stakingFee = 0;
            if(totalStakes > 0)
                _stakingFee= (onePercent(tokens).mul(stakingFee)).div(10);

            if(totalStakes > 0)
                // distribute the staking fee accumulated before updating the user's stake
                _addPayout(_stakingFee);

            // add pending rewards to remainder to be claimed by user later, if there is any
existing stake
            uint256 owing = pendingReward(msg.sender);
            stakers[msg.sender].remainder += owing;

            stakers[msg.sender].stakedTokens =
(tokens.sub(_stakingFee)).add(stakers[msg.sender].stakedTokens);
            stakers[msg.sender].lastDividends = owing;
            stakers[msg.sender].fromTotalDividend= totalDividends;
            stakers[msg.sender].round =  round;

            totalStakes = totalStakes.add(tokens.sub(_stakingFee));

            emit STAKED(msg.sender, tokens.sub(_stakingFee), _stakingFee);
        }

        // ------------------------------------------------------------------------
        // Owners can send the funds to be distributed to stakers using this function
        // @param tokens number of tokens to distribute
        // ------------------------------------------------------------------------
        function ADDFUNDS(uint256 tokens) external {
            require(IERC20(XFI).transferFrom(msg.sender, address(this), tokens), "Tokens
cannot be transferred from funder account");
            _addPayout(tokens);
        }

        // ------------------------------------------------------------------------
```

```
        // Private function to register payouts
        // -----------------------------------------------------------------------
        function _addPayout(uint256 tokens) private{
            // divide the funds among the currently staked tokens
            // scale the deposit and add the previous remainder
            uint256 available = (tokens.mul(scaling)).add(scaledRemainder);
            uint256 dividendPerToken = available.div(totalStakes);
            scaledRemainder = available.mod(totalStakes);

            totalDividends = totalDividends.add(dividendPerToken);
            payouts[round] = payouts[round-1].add(dividendPerToken);

            emit PAYOUT(round, tokens, msg.sender);
            round++;
        }

        // -----------------------------------------------------------------------
        // Stakers can claim their pending rewards using this function
        // -----------------------------------------------------------------------
        function CLAIMREWARD() public {
            if(totalDividends > stakers[msg.sender].fromTotalDividend){
                uint256 owing = pendingReward(msg.sender);

                owing = owing.add(stakers[msg.sender].remainder);
                stakers[msg.sender].remainder = 0;

                require(IERC20(XFI).transfer(msg.sender,owing), "ERROR: error in sending
reward from contract");

                emit CLAIMEDREWARD(msg.sender, owing);

                stakers[msg.sender].lastDividends = owing; // unscaled
                stakers[msg.sender].round = round; // update the round
                stakers[msg.sender].fromTotalDividend = totalDividends; // scaled
            }
        }

        // -----------------------------------------------------------------------
        // Get the pending rewards of the staker
        // @param _staker the address of the staker
        // -----------------------------------------------------------------------
        function pendingReward(address staker) private returns (uint256) {
            uint256 amount =  ((totalDividends.sub(payouts[stakers[staker].round -
1])).mul(stakers[staker].stakedTokens)).div(scaling);
            stakers[staker].remainder +=
((totalDividends.sub(payouts[stakers[staker].round -
1])).mul(stakers[staker].stakedTokens)) % scaling ;
            return amount;
        }

        function getPendingReward(address staker) public view returns(uint256
_pendingReward) {
            uint256 amount =  ((totalDividends.sub(payouts[stakers[staker].round -
1])).mul(stakers[staker].stakedTokens)).div(scaling);
            amount += ((totalDividends.sub(payouts[stakers[staker].round -
1])).mul(stakers[staker].stakedTokens)) % scaling ;
            return (amount + stakers[staker].remainder);
        }

        // -----------------------------------------------------------------------
        // Stakers can un stake the staked tokens using this function
        // @param tokens the number of tokens to withdraw
        // -----------------------------------------------------------------------
        function WITHDRAW(uint256 tokens) external {

            require(stakers[msg.sender].stakedTokens >= tokens && tokens > 0, "Invalid token
amount to withdraw");

            uint256 _unstakingFee = (onePercent(tokens).mul(unstakingFee)).div(10);

            // add pending rewards to remainder to be claimed by user later, if there is any
existing stake
            uint256 owing = pendingReward(msg.sender);
            stakers[msg.sender].remainder += owing;

            require(IERC20(XFI).transfer(msg.sender, tokens.sub(_unstakingFee)), "Error in
un-staking tokens");
```

```
        stakers[msg.sender].stakedTokens =
stakers[msg.sender].stakedTokens.sub(tokens);
        stakers[msg.sender].lastDividends = owing;
        stakers[msg.sender].fromTotalDividend= totalDividends;
        stakers[msg.sender].round =  round;

        totalStakes = totalStakes.sub(tokens);

        if(totalStakes > 0)
            // distribute the un staking fee accumulated after updating the user's stake
            _addPayout(_unstakingFee);

        emit UNSTAKED(msg.sender, tokens.sub(_unstakingFee), _unstakingFee);
    }

    // -----------------------------------------------------------------------
    // Private function to calculate 1% percentage
    // -----------------------------------------------------------------------
    function onePercent(uint256 _tokens) private pure returns (uint256){
        uint256 roundValue = _tokens.ceil(100);
        uint onePercentofTokens = roundValue.mul(100).div(100 * 10**uint(2));
        return onePercentofTokens;
    }

    // -----------------------------------------------------------------------
    // Get the number of tokens staked by a staker
    // @param _staker the address of the staker
    // -----------------------------------------------------------------------
    function yourStakedXFI(address staker) external view returns(uint256 stakedXFI){
        return stakers[staker].stakedTokens;
    }

    // -----------------------------------------------------------------------
    // Get the XFI balance of the token holder
    // @param user the address of the token holder
    // -----------------------------------------------------------------------
    function yourXFIBalance(address user) external view returns(uint256 XFIBalance){
        return IERC20(XFI).balanceOf(user);
    }
}
```

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.