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	2020/11/10	V1.0	Haojie Xu

Document information

Document Name	Audit Date	Audit results	Privacy level	Audit enquiry telephone
contracts_ethereum Smart Contract Security Audit Report	2020/11/10	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 November 8, 2020 to November 10, 2020. During this period, the Knownsec engineers audited the safety and regulatory aspects of contracts_ethereum 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..

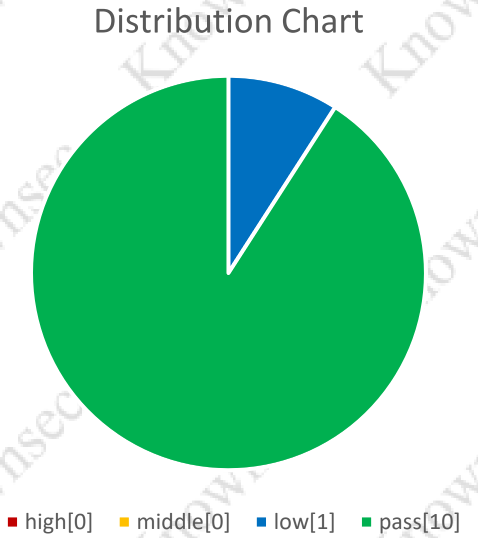
Target information for this test:

Project name	Project content
Token name	contracts_ethereum
Code type	Token code
Code language	solidity
Code address	https://github.com/bitcheck/contracts_ethereum

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	Pass	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	Low risk	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【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.

Test results: No related vulnerabilities in smart contract code.

Safety advice: None.

3.7. Denial of service attack detection【Pass】

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: No related vulnerabilities in smart contract code.

Safety advice: None.

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 【Low risk】

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.

Test results: Having related vulnerabilities in smart contract code.

```
    function mint(address account, uint256 amount) public onlyAuthorizedContract {
        if(_totalSupply.add(amount) > maxSupply) amount = _totalSupply.add(amount).sub(maxSupply);
        if(amount > 0) _mint(account, amount);
    }
```

Safety advice:

This problem is not a security problem, but some exchanges will limit the use of the additional issue function, and the specific situation needs to be determined according to the requirements of the exchange.

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

DividendPool.sol

```
/**
 *  $$$$$$\  $$\                 $$\
 * $$  __$$\ $$ |                $$ |
 * $$ /  \__|$$$$$$$\   $$$$$$\  $$ |  $$\  $$$$$$\   $$$$$$\
 * \$$$$$$\  $$  __$$\  \____$$\ $$ | $$  |$$  __$$\ $$  __$$\
 *  \____$$\ $$ |  $$ | $$$$$$$ |$$$$$$  / $$$$$$$$ |$$ |  \__|
 * $$\   $$ |$$ |  $$ |$$  __$$ |$$  _$$<  $$   ____|$$ |
 * \$$$$$$  |$$ |  $$ |\$$$$$$$ |$$ | \$$\ \$$$$$$$\ $$ |
 *  \______/ \__|  \__| \_______|\__|  \__| \_______|\__|
 * $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
 * ____________________________________________________________
*/

pragma solidity >=0.4.23 <0.6.0;

import "./Mocks/BTCHToken.sol";
import "./ReentrancyGuard.sol";
import "./Mocks/SafeMath.sol";

contract DividendPool is ReentrancyGuard {
    using SafeMath for uint256;
    uint256 public totalBalance = 0;
    address public tokenAddress;    // BTCH
    address public dividentAddress; // USDT Token
    address public feeAddress; // must be same as commonWithdrawAddress in ShakerV2.sol
    address public operator;

    // Share dividents of fee
    uint256 public currentStartTimestamp = 0;
    uint256 public totalDividents = 0;
    uint256 public sentDividents = 0;
    uint256 public getDividentsTimeout = 172800;// Have 2 days to getting current dividents

    event Dividend(address to, uint256 amount, uint256 timestamp);

    mapping(address => uint256) private lastGettingDividentsTime;
    mapping(address => uint256) public balances;

    BTCHToken public token;
    ERC20 public dividentToken;

    modifier onlyOperator {
        require(msg.sender == operator, "Only operator can call this function.");
        _;
    }

    constructor(
      address _tokenAddress,
      address _dividentAddress,
      address _feeAddress
    ) public {
        tokenAddress = _tokenAddress;
        token = BTCHToken(tokenAddress);
        dividentAddress = _dividentAddress;
        dividentToken = ERC20(dividentAddress);
        operator = msg.sender;
        feeAddress = _feeAddress;
    }

    function depositBTCH(uint256 amount) external nonReentrant {
        require(amount > 0);
        require(!(block.timestamp <= currentStartTimestamp + getDividentsTimeout)
|| !(block.timestamp >= currentStartTimestamp), "You can not deposit during taking divident
time");
        require(amount <= token.balanceOf(msg.sender), "Your balance is not enough");
        require(token.allowance(msg.sender, address(this)) >= amount, "Your allowance is
not enough");
        token.transferFrom(msg.sender, address(this), amount);
        balances[msg.sender] = balances[msg.sender].add(amount);
        totalBalance = totalBalance.add(amount);
    }
```

```
    function withdrawBTCH(uint256 amount) external nonReentrant {
        require(amount > 0);
        require(!(block.timestamp <= currentStartTimestamp + getDividentsTimeout)
|| !(block.timestamp >= currentStartTimestamp), "You can not withdraw during taking divident
time");
        require(amount <= balances[msg.sender], "Your deposit balance is not enough");
        token.transfer(msg.sender, amount);
        balances[msg.sender] = balances[msg.sender].sub(amount);
        totalBalance = totalBalance.sub(amount);
    }

    function updateTokenAddress(address _addr) external onlyOperator nonReentrant {
      require(_addr != address(0));
      tokenAddress = _addr;
      token = BTCHToken(tokenAddress);
    }

    function updateOperator(address _addr) external onlyOperator nonReentrant {
      require(_addr != address(0));
      operator = _addr;
    }

    function getBalance() external view returns(uint256) {
      return balances[msg.sender];
    }

    function getDividentsAmount() public view returns(uint256, uint256) {
      // Caculate normal dividents
      require(totalBalance > 0);
      return (totalDividents.mul(balances[msg.sender]).div(totalBalance),
lastGettingDividentsTime[msg.sender]);
    }

    function setDividentAddress(address _address) external onlyOperator {
      dividentAddress = _address;
      dividentToken = ERC20(dividentAddress);
    }

    function setFeeAddress(address _address) external onlyOperator {
        require(_address != address(0));
        feeAddress = _address;
    }

    function sendDividents() external nonReentrant {
      // Only shaker contract can call this function
      require(block.timestamp <= currentStartTimestamp + getDividentsTimeout &&
block.timestamp >= currentStartTimestamp, "Getting dividents not start or it's already
end");
      require(lastGettingDividentsTime[msg.sender] < currentStartTimestamp, "You have got
dividents already");
      (uint256 normalDividents,) = getDividentsAmount();

      // Send Dividents
      // The fee account must approve the this contract enough allowance of USDT as dividend
      require(dividentToken.allowance(feeAddress, address(this)) >= normalDividents,
"Allowance not enough");
      dividentToken.transferFrom(feeAddress, msg.sender, normalDividents);
      sentDividents = sentDividents.add(normalDividents);
      lastGettingDividentsTime[msg.sender] = block.timestamp;
      emit Dividend(msg.sender, normalDividents, block.timestamp);
    }

    /** Start Dividents by operator */
    function startDividents(uint256 from, uint256 amount) external onlyOperator
nonReentrant{
      require(from > block.timestamp);
      require(amount > 0);
      currentStartTimestamp = from;
      totalDividents = amount;
      sentDividents = 0;
    }

    function setGettingDividentsTimeout(uint256 _seconds) external onlyOperator {
      getDividentsTimeout = _seconds;
    }
```

```
  function getLastTakingDividentsTime() external view returns(uint256) {
    return lastGettingDividentsTime[msg.sender];
  }
}
```

ERC20ShakerV2.sol

```
/**
 *  $$$$$$\  $$\                 $$\
 * $$  __$$\ $$ |                $$ |
 * $$ /  \__|$$$$$$$\   $$$$$$\  $$ |  $$\  $$$$$$\   $$$$$$\
 * \$$$$$$\  $$  __$$\  \____$$\ $$ | $$  |$$  __$$\ $$  __$$\
 *  \____$$\ $$ |  $$ | $$$$$$$ |$$$$$$  / $$$$$$$$ |$$ |  \__|
 * $$\   $$ |$$ |  $$ |$$  __$$ |$$  _$$<  $$   ____|$$ |
 * \$$$$$$  |$$ |  $$ |\$$$$$$$ |$$ | \$$\ \$$$$$$$\ $$ |
 *  \______/ \__|  \__| \_______|\__|  \__| \_______|\__|
 * $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
 * ______________________________________________________________
*/

pragma solidity >=0.4.23 <0.6.0;

import "./ShakerV2.sol";

contract ERC20ShakerV2 is ShakerV2 {
  address public token;

  constructor(
    address _operator,
    address _commonWithdrawAddress,
    address _token
  ) ShakerV2(_operator, _commonWithdrawAddress) public {
    token = _token;
  }

  function _processDeposit(uint256 _amount) internal {
    require(msg.value == 0, "ETH value is supposed to be 0 for ERC20 instance");
    _safeErc20TransferFrom(msg.sender, address(this), _amount);
  }

  function _processWithdraw(address payable _recipient, address _relayer, uint256 _fee,
uint256 _refund) internal {
    _safeErc20Transfer(_recipient, _refund.sub(_fee));
    if(_fee > 0) _safeErc20Transfer(_relayer, _fee);
  }

  function _safeErc20TransferFrom(address _from, address _to, uint256 _amount) internal
{
    (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x23b872dd /*
transferFrom */, _from, _to, _amount));
    require(success, "not enough allowed tokens");

    // if contract returns some data lets make sure that is `true` according to standard
    if (data.length > 0) {
      require(data.length == 32, "data length should be either 0 or 32 bytes");
      success = abi.decode(data, (bool));
      require(success, "not enough allowed tokens. Token returns false.");
    }
  }

  function _safeErc20Transfer(address _to, uint256 _amount) internal {
    (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0xa9059cbb /*
transfer */, _to, _amount));
    require(success, "not enough tokens");

    // if contract returns some data lets make sure that is `true` according to standard
    if (data.length > 0) {
      require(data.length == 32, "data length should be either 0 or 32 bytes");
      success = abi.decode(data, (bool));
      require(success, "not enough tokens. Token returns false.");
    }
  }
}
```

ShakerTokenManager.sol

```
/**
```

```
 *  $$$$$$\  $$\                 $$\
 * $$  __$$\ $$ |                $$ |
 * $$ /  \__|$$$$$$$\   $$$$$$\  $$ |  $$\  $$$$$$\   $$$$$$\
 * \$$$$$$\  $$  __$$\  \____$$\ $$ | $$  |$$  __$$\ $$  __$$\
 *  \____$$\ $$ |  $$ | $$$$$$$ |$$$$$$  / $$$$$$$$ |$$ |  \__|
 * $$\   $$ |$$ |  $$ |$$  __$$ |$$  _$$<  $$   ____|$$ |
 * \$$$$$$  |$$ |  $$ |\$$$$$$$ |$$ | \$$\ \$$$$$$$\ $$ |
 *  \______/ \__|  \__| \_______|\__|  \__| \_______|\__|
 * $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
 * _____________________________________________________________
*/

pragma solidity >=0.4.23 <0.6.0;

import "./Mocks/BTCHToken.sol";
import "./Mocks/ERC20.sol";
import "./ReentrancyGuard.sol";

/**
 * The bonus will calculated with 5 factors:
 * 1- Base bonus factor. This is base bonus factor, here we set it as 0.05
 * 2- Amount after exponent. This will reduce the weight of whale capital. Here we set it as 2/3. That means if somebody deposit 100,000, will just caculatd busnus
 *    according to 100,000 ** (2/3) = 2154.435
 * 3- Hours factor between Deposit Withdraw. This is let the people store the money in contract longer.
 *    If below 1 hour, there will be no bonus token.
 *    from 1-24 hours, the factor is 0.05; 24-48 hours, factor is 0.15, etc.
 *    This factor will be modified by council according to the market.
 * 4- Stage factor: If total mint token is 300,000, we will devided it into several stages. Each stage has special bonus times. Ex. if stage factor is 5,
 *    means in this stage, miner will get 5 times than normal.
 * 5- Price elastical factor. We want the mint quantity for each deposit can be different under different market. If the price is higher than normal, the factor will
 *    become smaller automaticly, and if the price go down, the factor will become smaller also. It is a Gaussian distribution, and the average price (normal price)
 *    is fee of deposit and withdrawal.
 *
 * So the bonus amount will be:
 * Bonus amount = Amount after exponent * Base bonus factor * hours factor * stage factor * price elastical factor.
 *
 * In this version, we will keep price elastical factor as 1.
 *
 */
contract ShakerTokenManager is ReentrancyGuard {
   using SafeMath for uint256;

   uint256 public bonusTokenDecimals = 6; // bonus token decimals, BTCH
   uint256 public depositTokenDecimals = 6; // deposit and withdrawal token decimals, USDT

   // Params
   uint256 public baseFactor = 50; // 50 means 0.05
   uint256[] public intervalOfDepositWithdraw = [1, 24, 48, 96, 192, 384, 720]; // hours of inverval between deposit and withdraw
   uint256[] public intervalOfDepositWithdrawFactor = [5000, 15000, 16800, 20600, 28600, 45500, 81500]; // 5000 will be devided by 1e5, means 0.05
   uint256[] public stageFactors = [10000, 5000, 2500, 1250, 625]; // Stage factor, 5000 means 5, 2500 means 2.5, etc.
   uint256 public eachStageAmount = 7200000 * 10 ** bonusTokenDecimals; // Each stage amount
   uint256[] public exponent = [2, 3];// means 2/3
   uint256 public feeRate = 33333; // 33.333% will be 33333
   uint256 public minChargeFeeAmount = 10 * 10 ** depositTokenDecimals;// Below this amount, will only charge  very special fee
   uint256 public minChargeFee = 0; // min amount of special charge.
   uint256 public minChargeFeeRate = 180; // percent rate of special charge, if need to charge 1.5%, this will be set 150
   uint256 public minMintAmount = 10 * 10 ** depositTokenDecimals;
   uint256 public taxRate = 2000;// means 20%
   uint256 public depositerShareRate = 5000; // depositer and withdrawer will share the bonus, this rate is for sender(depositer). 5000 means 0.500, 50%;

   address public operator;
   address public taxBereauAddress; // address to get tax
   address public shakerContractAddress;
   address public tokenAddress; // BTCH token
```

```
    BTCHToken public token = BTCHToken(tokenAddress);

    modifier onlyOperator {
        require(msg.sender == operator, "Only operator can call this function.");
        _;
    }

    modifier onlyShaker {
        require(msg.sender == shakerContractAddress, "Only bitcheck contract can call this
function.");
        _;
    }

    constructor(address _shakerContractAddress, address _taxBereauAddress) public {
        operator = msg.sender;
        shakerContractAddress = _shakerContractAddress;
        taxBereauAddress = _taxBereauAddress;
    }

    function sendBonus(uint256 _amount, uint256 _hours, address _depositer, address
_withdrawer) external nonReentrant onlyShaker returns(bool) {
        uint256 mintAmount = this.getMintAmount(_amount, _hours);
        uint256 tax = mintAmount.mul(taxRate).div(10000);
        uint256 notax = mintAmount.sub(tax);
        token.mint(_depositer, (notax.mul(depositerShareRate).div(10000)));
        token.mint(_withdrawer,
(notax.mul(uint256(10000).sub(depositerShareRate)).div(10000)));
        token.mint(taxBereauAddress, tax);
        return true;
    }

    function burn(uint256 _amount, address _from) external nonReentrant onlyShaker
returns(bool) {
        token.burn(_from, _amount);
        return true;
    }

    function getMintAmount(uint256 _amount, uint256 _hours ) external view returns(uint256)
{
        // return back bonus token amount with decimals
        require(_amount < 1e18);
        if(_amount <= minMintAmount) return 0;
        uint256 amountExponented = getExponent(_amount);
        uint256 stageFactor = getStageFactor();
        uint256 intervalFactor = getIntervalFactor(_hours);
        uint256 priceFactor = getPriceElasticFactor();
        return
amountExponented.mul(priceFactor).mul(baseFactor).mul(intervalFactor).mul(stageFactor)
.div(1e11);
    }

    function getFee(uint256 _amount) external view returns(uint256) {
        // return fee amount, including decimals
        require(_amount < 1e18);
        if(_amount <= minChargeFeeAmount) return getSpecialFee(_amount);
        uint256 amountExponented = getExponent(_amount);
        return amountExponented.mul(feeRate).div(1e5);
    }

    function getExponent(uint256 _amount) internal view returns(uint256) {
        // if 2000, the _amount should be 2000 * 10**decimals, return back 2000**(2/3) *
10**decimals
        if(_amount > 1e18) return 0;
        uint256 e = nthRoot(_amount, exponent[1], bonusTokenDecimals, 1e18);
        return e.mul(e).div(10 ** (bonusTokenDecimals + depositTokenDecimals * exponent[0]
/ exponent[1]));
    }

    function getStageFactor() internal view returns(uint256) {
        uint256 tokenTotalSupply = getTokenTotalSupply();
        uint256 stage = tokenTotalSupply.div(eachStageAmount);
        return stageFactors[stage > stageFactors.length - 1 ? stageFactors.length - 1 :
stage];
    }

    function getIntervalFactor(uint256 _hours) internal view returns(uint256) {
        uint256 id = intervalOfDepositWithdraw.length - 1;
```

```
        for(uint8 i = 0; i < intervalOfDepositWithdraw.length; i++) {
            if(intervalOfDepositWithdraw[i] > _hours) {
                id = i == 0 ? 999 : i - 1;
                break;
            }
        }
        return id == 999 ? 0 : intervalOfDepositWithdrawFactor[id];
    }

    // For tesing, Later will update ######
    function getPriceElasticFactor() internal pure returns(uint256) {
        return 1;
    }

    function getTokenTotalSupply() public view returns(uint256) {
        return token.totalSupply();
    }

    function getSpecialFee(uint256 _amount) internal view returns(uint256) {
        return _amount.mul(minChargeFeeRate).div(10000).add(minChargeFee);
    }
    // calculates a^(1/n) to dp decimal places
    // maxIts bounds the number of iterations performed
    function nthRoot(uint _a, uint _n, uint _dp, uint _maxIts) internal pure returns(uint)
{
        assert (_n > 1);

        // The scale factor is a crude way to turn everything into integer calcs.
        // Actually do (a * (10 ^ ((dp + 1) * n))) ^ (1/n)
        // We calculate to one extra dp and round at the end
        uint one = 10 ** (1 + _dp);
        uint a0 = one ** _n * _a;

        // Initial guess: 1.0
        uint xNew = one;
        uint x;
        uint iter = 0;
        while (xNew != x && iter < _maxIts) {
            x = xNew;
            uint t0 = x ** (_n - 1);
            if (x * t0 > a0) {
                xNew = x - (x - a0 / t0) / _n;
            } else {
                xNew = x + (a0 / t0 - x) / _n;
            }
            ++iter;
        }

        // Round to nearest in the last dp.
        return (xNew + 5) / 10;
    }

    function setStageFactors(uint256[] calldata _stageFactors) external onlyOperator {
        stageFactors = _stageFactors;
    }

    function setIntervalOfDepositWithdraw(uint256[] calldata _intervalOfDepositWithdraw,
uint256[] calldata _intervalOfDepositWithdrawFactor) external onlyOperator {
      intervalOfDepositWithdrawFactor = _intervalOfDepositWithdrawFactor;
      intervalOfDepositWithdraw = _intervalOfDepositWithdraw;
    }

    function setBaseFactor(uint256 _baseFactor) external onlyOperator {
        baseFactor = _baseFactor;
    }

    function setBonusTokenDecimals(uint256 _decimals) external onlyOperator {
        require(_decimals >= 0);
        bonusTokenDecimals = _decimals;
    }

    function setDeositTokenDecimals(uint256 _decimals) external onlyOperator {
        require(_decimals >= 0);
        depositTokenDecimals = _decimals;
    }

    function setTokenAddress(address _address) external onlyOperator {
```

```
        tokenAddress = _address;
        token = BTCHToken(tokenAddress);
    }

    function setShakerContractAddress(address _shakerContractAddress) external
onlyOperator {
        shakerContractAddress = _shakerContractAddress;
    }

    function setExponent(uint256[] calldata _exp) external onlyOperator {
        require(_exp.length == 2 && _exp[1] >= 0);
        exponent = _exp;
    }

    function setEachStageAmount(uint256 _eachStageAmount) external onlyOperator {
        require(_eachStageAmount >= 0);
        eachStageAmount = _eachStageAmount;
    }

    function setMinChargeFeeParams(uint256 _maxAmount, uint256 _minFee, uint256 _feeRate)
external onlyOperator {
        minChargeFeeAmount = _maxAmount;
        minChargeFee = _minFee;
        minChargeFeeRate = _feeRate;
    }

    function setMinMintAmount(uint256 _amount) external onlyOperator {
        require(_amount >= 0);
        minMintAmount = _amount;
    }

    function setFeeRate(uint256 _feeRate) external onlyOperator {
        require(_feeRate <= 100000 && _feeRate >= 0);
        feeRate = _feeRate;
    }

    function setTaxBereauAddress(address _taxBereauAddress) external onlyOperator {
        taxBereauAddress = _taxBereauAddress;
    }

    function setTaxRate(uint256 _rate) external onlyOperator {
        require(_rate <= 10000 && _rate >= 0);
        taxRate = _rate;
    }

    function setDepositerShareRate(uint256 _rate) external onlyOperator {
        require(_rate <= 10000 && _rate >= 0);
        depositerShareRate = _rate;
    }

    function updateOperator(address _newOperator) external onlyOperator {
        require(_newOperator != address(0));
        operator = _newOperator;
    }

}
```

ShakerV2.sol

```
/**
 *  $$$$$$\  $$\                 $$\
 * $$  __$$\ $$ |                $$ |
 * $$ /  \__|$$$$$$$\   $$$$$$\  $$ |  $$\  $$$$$$\   $$$$$$\
 * \$$$$$$\  $$  __$$\  \____$$\ $$ | $$  |$$  __$$\ $$  __$$\
 *  \____$$\ $$ |  $$ | $$$$$$$ |$$$$$$  / $$$$$$$$ |$$ |  \__|
 * $$\   $$ |$$ |  $$ |$$  __$$ |$$  _$$<  $$   ____|$$ |
 * \$$$$$$  |$$ |  $$ |\$$$$$$$ |$$ | \$$\ \$$$$$$$\ $$ |
 *  \______/ \__|  \__| \_______|\__|  \__| \_______|\__|
 * $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
 * ______________________________________________________________
*/

pragma solidity >=0.4.23 <0.6.0;

import "./ReentrancyGuard.sol";
import "./StringUtils.sol";
import "./ShakerTokenManager.sol";
```

```
contract ShakerV2 is ReentrancyGuard, StringUtils {
    using SafeMath for uint256;
    uint256 public totalAmount = 0; // Total amount of deposit
    uint256 public totalBalance = 0; // Total balance of deposit after Withdrawal

    address public operator;          // Super operator account to control the contract
    address public councilAddress;     // Council address of DAO
    uint256 public councilJudgementFee = 0; // Council charge for judgement
    uint256 public councilJudgementFeeRate = 1700; // If the desired rate is 17%,
commonFeeRate should set to 1700

    ShakerTokenManager public tokenManager;

    struct Commitment {               // Deposit Commitment
        bool          status;          // If there is no this commitment or balance is zeor,
false
        uint256       amount;           // Deposit balance
        address payable sender;        // Who make this deposit
        uint256       effectiveTime;     // Forward cheque time
        uint256       timestamp;        // Deposit timestamp
        bool          canEndorse;
        bool          lockable;         // If can be locked/refund
    }
    // Mapping of commitments, must be private. The key is hashKey =
hash(commitment,recipient)
    // The contract will hide the recipient and commitment while make deposit.
    mapping(bytes32 => Commitment) private commitments;

    // Relayer is service to do the deposit and Withdrawal on server, this address is for
recieving fee
    mapping(address => address) private relayerWithdrawAddress;

    // If the msg.sender(relayer) has not registered Withdrawal address, the fee will send
to this address
    address public commonWithdrawAddress;

    // If withdrawal is not throught relayer, use this common fee. Be care of decimal of
token
    // uint256 public commonFee = 0;

    // If withdrawal is not throught relayer, use this rate. Total fee is: commoneFee + amount
* commonFeeRate.
    // If the desired rate is 4%, commonFeeRate should set to 400
    // uint256 public commonFeeRate = 25; // 0.25%

    struct LockReason {
        string  description;
        uint8   status; // 1- locked, 2- confirm by recipient, 0- never happend, 3- unlocked
by council, 4- cancel by sender
        uint256 datetime;
        uint256 refund;
        address payable locker;
        bool    recipientAgree;
        bool    senderAgree;
        bool    toCouncil;
    }
    // lockakReason key is hashKey = hash(commitment, recipient)
    mapping(bytes32 => LockReason) private lockReason;

    modifier onlyOperator {
        require(msg.sender == operator, "Only operator can call this function.");
        _;
    }

    modifier onlyRelayer {
        require(relayerWithdrawAddress[msg.sender] != address(0x0), "Only relayer can call
this function.");
        _;
    }

    modifier onlyCouncil {
        require(msg.sender == councilAddress, "Only council account can call this
function.");
        _;
    }
```

```
    event Deposit(address sender, bytes32 hashkey, uint256 amount, uint256 timestamp);
    event Withdrawal(string commitment, uint256 fee, uint256 amount, uint256 timestamp);

    constructor(
        address _operator,
        address _commonWithdrawAddress
    ) public {
        operator = _operator;
        councilAddress = _operator;
        commonWithdrawAddress = _commonWithdrawAddress;
    }

    function depositERC20Batch(
        bytes32[] calldata _hashKey,
        uint256[] calldata _amounts,
        uint256[] calldata _effectiveTime
    ) external payable nonReentrant {
        for(uint256 i = 0; i < _amounts.length; i++) {
            _deposit(_hashKey[i], _amounts[i], _effectiveTime[i]);
        }
    }

    function _deposit(
        bytes32 _hashKey,
        uint256 _amount,
        uint256 _effectiveTime
    ) internal {
        require(!commitments[_hashKey].status, "The commitment has been submitted or used
out.");
        require(_amount > 0);

        _processDeposit(_amount);

        commitments[_hashKey].status = true;
        commitments[_hashKey].amount = _amount;
        commitments[_hashKey].sender = msg.sender;
        commitments[_hashKey].effectiveTime = _effectiveTime < block.timestamp ?
block.timestamp : _effectiveTime;
        commitments[_hashKey].timestamp = block.timestamp;
        commitments[_hashKey].canEndorse = false;
        commitments[_hashKey].lockable = true;

        totalAmount = totalAmount.add(_amount);
        totalBalance = totalBalance.add(_amount);

        emit Deposit(msg.sender, _hashKey, _amount, block.timestamp);
    }

    function _processDeposit(uint256 _amount) internal;

    function withdrawERC20Batch(
        bytes32[] calldata _commitments,
        uint256[] calldata _amounts,
        uint256[] calldata _fees,
        address[] calldata _relayers
    ) external payable nonReentrant {
        for(uint256 i = 0; i < _commitments.length; i++)
_withdraw(bytes32ToString(_commitments[i]), _amounts[i], _fees[i], _relayers[i]);
    }

    function _withdraw(
        string memory _commitment,
        uint256 _amount,            // Withdrawal amount
        uint256 _fee,               // Fee caculated by relayer
        address _relayer            // Relayer address
    ) internal {
        bytes32 _hashkey = getHashkey(_commitment);
        require(commitments[_hashkey].amount > 0, 'The commitment of this recipient is not
exist or used out');
        require(lockReason[_hashkey].status != 1, 'This deposit was locked');
        uint256 refundAmount = _amount < commitments[_hashkey].amount ? _amount :
commitments[_hashkey].amount; //Take all if _refund == 0
        require(refundAmount > 0, "Refund amount can not be zero");
        require(block.timestamp >= commitments[_hashkey].effectiveTime, "The deposit is
locked until the effectiveTime");
        require(refundAmount >= _fee, "Refund amount should be more than fee");
```

```
        address relayer = relayerWithdrawAddress[_relayer] == address(0x0) ?
commonWithdrawAddress : relayerWithdrawAddress[_relayer];
        uint256 _fee1 = tokenManager.getFee(refundAmount);
        require(_fee1 <= refundAmount, "The fee can not be more than refund amount");
        uint256 _fee2 = relayerWithdrawAddress[_relayer] == address(0x0) ? _fee1 : _fee;
// If not through relay, use commonFee
        _processWithdraw(msg.sender, relayer, _fee2, refundAmount);

        commitments[_hashkey].amount = (commitments[_hashkey].amount).sub(refundAmount);
        commitments[_hashkey].status = commitments[_hashkey].amount <= 0 ? false : true;
        totalBalance = totalBalance.sub(refundAmount);

        uint256 _hours =
(block.timestamp.sub(commitments[_hashkey].timestamp)).div(3600);
        tokenManager.sendBonus(refundAmount, _hours, commitments[_hashkey].sender,
msg.sender);

        emit Withdrawal(_commitment, _fee, refundAmount, block.timestamp);
    }

    function _processWithdraw(address payable _recipient, address _relayer, uint256 _fee,
uint256 _refund) internal;
    function _safeErc20Transfer(address _to, uint256 _amount) internal;

    function getHashkey(string memory _commitment) internal view returns(bytes32) {
        string memory commitAndTo = concat(_commitment, addressToString(msg.sender));
        return keccak256(abi.encodePacked(commitAndTo));
    }

    function endorseERC20Batch(
        uint256[] calldata _amounts,
        bytes32[] calldata _oldCommitments,
        bytes32[] calldata _newHashKeys,
        uint256[] calldata _effectiveTimes
    ) external payable nonReentrant {
        for(uint256 i = 0; i < _amounts.length; i++) _endorse(_amounts[i],
bytes32ToString(_oldCommitments[i]), _newHashKeys[i], _effectiveTimes[i]);
    }

    function _endorse(
        uint256 _amount,
        string memory _oldCommitment,
        bytes32 _newHashKey,
        uint256 _effectiveTime
    ) internal {
        bytes32 _oldHashKey = getHashkey(_oldCommitment);
        require(lockReason[_oldHashKey].status != 1, 'This deposit was locked');
        require(commitments[_oldHashKey].status, "Old commitment can not find");
        require(!commitments[_newHashKey].status, "The new commitment has been submitted
or used out");
        require(commitments[_oldHashKey].canEndorse, "Old commitment can not endorse");
        require(commitments[_oldHashKey].amount > 0, "No balance amount of this proof");
        uint256 refundAmount = _amount < commitments[_oldHashKey].amount ? _amount :
commitments[_oldHashKey].amount; //Take all if _refund == 0
        require(refundAmount > 0, "Refund amount can not be zero");

        if(_effectiveTime > 0 && block.timestamp >= commitments[_oldHashKey].effectiveTime)
commitments[_oldHashKey].effectiveTime = _effectiveTime; // Effective
        else commitments[_newHashKey].effectiveTime =
commitments[_oldHashKey].effectiveTime; // Not effective

        commitments[_newHashKey].status = true;
        commitments[_newHashKey].amount = refundAmount;
        commitments[_newHashKey].sender = msg.sender;
        commitments[_newHashKey].timestamp = block.timestamp;
        commitments[_newHashKey].canEndorse = false;
        commitments[_newHashKey].lockable = true;

        commitments[_oldHashKey].amount =
(commitments[_oldHashKey].amount).sub(refundAmount);
        commitments[_oldHashKey].status = commitments[_oldHashKey].amount <= 0 ? false :
true;

        emit Withdrawal(_oldCommitment, 0, refundAmount, block.timestamp);
        emit Deposit(msg.sender, _newHashKey, refundAmount, block.timestamp);
    }
```

```
    /** @dev whether a note is already spent */
    function isSpent(bytes32 _hashkey) public view returns(bool) {
        return commitments[_hashkey].amount == 0 ? true : false;
    }

    /** @dev whether an array of notes is already spent */
    function isSpentArray(bytes32[] calldata _hashkeys) external view returns(bool[] memory
spent) {
        spent = new bool[](_hashkeys.length);
        for(uint i = 0; i < _hashkeys.length; i++) spent[i] = isSpent(_hashkeys[i]);
    }

    /** @dev operator can change his address */
    function updateOperator(address _newOperator) external nonReentrant onlyOperator {
        operator = _newOperator;
    }

    /** @dev update authority relayer */
    function updateRelayer(address _relayer, address _withdrawAddress) external
nonReentrant onlyOperator {
        relayerWithdrawAddress[_relayer] = _withdrawAddress;
    }

    /** @dev get relayer Withdrawal address */
    function getRelayerWithdrawAddress() view external onlyRelayer returns(address) {
        return relayerWithdrawAddress[msg.sender];
    }

    /** @dev update commonWithdrawAddress */
    function updateCommonWithdrawAddress(address _commonWithdrawAddress) external
nonReentrant onlyOperator {
        commonWithdrawAddress = _commonWithdrawAddress;
    }

    /** @dev set council address */
    function setCouncial(address _councilAddress) external nonReentrant onlyOperator {
        councilAddress = _councilAddress;
    }

    /** @dev lock commitment, this operation can be only called by note holder */
    function lockERC20Batch (
        bytes32         _hashkey,
        uint256         _refund,
        string  calldata  _description
    ) external payable nonReentrant {
        _lock(_hashkey, _refund, _description);
    }

    function _lock(
        bytes32 _hashkey,
        uint256 _refund,
        string memory _description
    ) internal {
        require(msg.sender == commitments[_hashkey].sender, 'Locker must be sender');
        require(commitments[_hashkey].lockable, 'This commitment must be lockable');
        require(commitments[_hashkey].amount >= _refund, 'Balance amount must be enough');

        lockReason[_hashkey] = LockReason(
            _description,
            1,
            block.timestamp,
            _refund == 0 ? commitments[_hashkey].amount : _refund,
            msg.sender,
            false,
            false,
            false
        );
    }

    function getLockReason(bytes32 _hashkey) public view returns(
        string memory  description,
        uint8          status,
        uint256        datetime,
        uint256        refund,
        address        locker,
        bool           recipientAgree,
        bool           senderAgree,
```

```
        bool          toCouncil
    ) {
        LockReason memory data = lockReason[_hashkey];
        return (
            data.description,
            data.status,
            data.datetime,
            data.refund,
            data.locker,
            data.recipientAgree,
            data.senderAgree,
            data.toCouncil
        );

    }

    function unlockByCouncil(bytes32 _hashkey, uint8 _result) external nonReentrant
onlyCouncil {
        // _result = 1: sender win
        // _result = 2: recipient win
        require(_result == 1 || _result == 2);
        if(lockReason[_hashkey].status == 1 && lockReason[_hashkey].toCouncil) {
            lockReason[_hashkey].status = 3;
            // If the council decided to return back money to the sender
            uint256 councilFee = getJudgementFee(lockReason[_hashkey].refund);
            if(_result == 1) {
                _processWithdraw(lockReason[_hashkey].locker, councilAddress, councilFee,
lockReason[_hashkey].refund);
                totalBalance = totalBalance.sub(lockReason[_hashkey].refund);
                commitments[_hashkey].amount =
(commitments[_hashkey].amount).sub(lockReason[_hashkey].refund);
                commitments[_hashkey].status = commitments[_hashkey].amount == 0 ? false :
true;
            } else {
                lockReason[_hashkey].status = 3;
                _safeErc20Transfer(councilAddress, councilFee);
                totalBalance = totalBalance.sub(councilFee);
                commitments[_hashkey].amount =
(commitments[_hashkey].amount).sub(councilFee);
                commitments[_hashkey].status = commitments[_hashkey].amount == 0 ? false :
true;
            }
        }
    }

    /**
     * recipient should agree to let sender refund, otherwise, will bring to the council
to make a judgement
     * This is 1st step if dispute happend
     */
    function unlockByRecipent(bytes32 _hashkey, bytes32 _commitment, uint8 _status)
external nonReentrant {
        bytes32 _recipientHashKey = getHashkey(bytes32ToString(_commitment));
        bool isSender = msg.sender == commitments[_hashkey].sender;
        bool isRecipent = _hashkey == _recipientHashKey;

        require(isSender || isRecipent, 'Must be called by recipient or original sender');
        require(_status == 1 || _status == 2);
        require(lockReason[_hashkey].status == 1);

        if(isSender) {
            // Sender accept to keep cheque available
            lockReason[_hashkey].status = _status == 2 ? 4 : 1;
            lockReason[_hashkey].senderAgree = _status == 2;
            lockReason[_hashkey].toCouncil = _status == 2;
        } else if(isRecipent) {
            // recipient accept to refund back to sender
            lockReason[_hashkey].status = _status;
            lockReason[_hashkey].recipientAgree = _status == 2;
            lockReason[_hashkey].toCouncil = _status == 2;
            // return back to sender
            if(_status == 2) {
                _processWithdraw(commitments[_hashkey].sender, address(0x0), 0,
lockReason[_hashkey].refund);
                totalBalance = totalBalance.sub(lockReason[_hashkey].refund);
                commitments[_hashkey].amount =
(commitments[_hashkey].amount).sub(lockReason[_hashkey].refund);
```

```
            commitments[_hashkey].status = commitments[_hashkey].amount == 0 ? false : true;
         } else {
            lockReason[_hashkey].toCouncil = true;
         }
      }
   }

   /**
    * Cancel effectiveTime and change cheque to at sight
    */
   function changeToAtSight(bytes32 _hashkey) external nonReentrant returns(bool) {
       require(msg.sender == commitments[_hashkey].sender, 'Only sender can change this cheque to at sight');
       if(commitments[_hashkey].effectiveTime > block.timestamp) commitments[_hashkey].effectiveTime = block.timestamp;
       return true;
   }

   function setCanEndorse(bytes32 _hashkey, bool status) external nonReentrant returns(bool) {
       require(msg.sender == commitments[_hashkey].sender, 'Only sender can change endorsable');
       commitments[_hashkey].canEndorse = status;
   }

   function setLockable(bytes32 _hashKey, bool status) external nonReentrant returns(bool) {
       require(msg.sender == commitments[_hashKey].sender, 'Only sender can change lockable');
       require(commitments[_hashKey].lockable == true && status == false, 'Can only change from lockable to non-lockable');
       commitments[_hashKey].lockable = status;
       commitments[_hashKey].canEndorse = true; // If the commitment can not be lock, it must be endorsed
   }

   function getDepositDataByHashkey(bytes32 _hashkey) external view returns(uint256 effectiveTime, uint256 amount, bool lockable, bool canEndorse) {
       effectiveTime = commitments[_hashkey].effectiveTime;
       amount = commitments[_hashkey].amount;
       lockable = commitments[_hashkey].lockable;
       canEndorse = commitments[_hashkey].canEndorse;
   }

   function updateCouncilJudgementFee(uint256 _fee, uint256 _rate) external nonReentrant onlyCouncil {
       councilJudgementFee = _fee;
       councilJudgementFeeRate = _rate;
   }

   function updateBonusTokenManager(address _BonusTokenManagerAddress) external nonReentrant onlyOperator {
       tokenManager = ShakerTokenManager(_BonusTokenManagerAddress);
   }

   function getJudgementFee(uint256 _amount) internal view returns(uint256) {
       return _amount * councilJudgementFeeRate / 10000 + councilJudgementFee;
   }
}
```

Mocks/BTCHToken.sol

```
/**
 *  $$$$$$\  $$\                 $$\
 * $$  __$$\ $$ |                $$ |
 * $$ /  \__|$$$$$$$\   $$$$$$\  $$ |  $$\  $$$$$$\   $$$$$$\
 * \$$$$$$\  $$  __$$\  \____$$\ $$ | $$  |$$  __$$\ $$  __$$\
 *  \____$$\ $$ |  $$ | $$$$$$$ |$$$$$$  / $$$$$$$$ |$$ |  \__|
 * $$\   $$ |$$ |  $$ |$$  __$$ |$$  _$$<  $$   ____|$$ |
 * \$$$$$$  |$$ |  $$ |\$$$$$$$ |$$ | \$$\ \$$$$$$$\ $$ |
 *  \______/ \__|  \__| \_______|\__|  \__| \_______|\__|
 * $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
 * ______________________________________________________________
*/

pragma solidity >=0.4.23 <0.6.0;
```

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

contract BTCHToken is ERC20, ERC20Detailed {
    address public authorizedContract;
    address public operator;
    uint256 public maxSupply = 36000000 * 10 ** 6;

    constructor (address _authorizedContract) public ERC20Detailed("BitCheck DAO", "BTCH",
6) {
        // Decimal is 6
        operator = msg.sender;
        authorizedContract = _authorizedContract;
        // zero pre-mine
    }

    modifier onlyOperator {
        require(msg.sender == operator, "Only operator can call this function.");
        _;
    }

    modifier onlyAuthorizedContract {
        require(msg.sender == authorizedContract, "Only authorized contract can call this
function.");
        _;
    }

    function mint(address account, uint256 amount) public onlyAuthorizedContract {
        if(_totalSupply.add(amount) > maxSupply) amount =
_totalSupply.add(amount).sub(maxSupply);
        if(amount > 0) _mint(account, amount);
    }

    function burn(address account, uint256 amount) public onlyAuthorizedContract {
        if(_totalSupply < amount) amount = amount.sub(_totalSupply);
        if(amount > 0) _burn(account, amount);
    }

    function updateOperator(address _newOperator) external onlyOperator {
        require(_newOperator != address(0));
        operator = _newOperator;
    }

    function updateAuthorizedContract(address _authorizedContract) external onlyOperator
{
        require(_authorizedContract != address(0));
        authorizedContract = _authorizedContract;
    }

}
```

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.