北京链安
Chains Guard Technology

CBK
Smart Contract
Audit Report

Beijing Chains Guard Technology

Documentation

File name	CBK Smart Contract Audit Report		
Serial number	CG-YMSJ-202009025		
Confidentiality	Business secret	Version	V1.1
Author	ChainsGuard Security Center	Date	2020-09-04

Scope of application

This security assessment is authorized by the project party. Beijing ChainsGuard Network Technology Co., Ltd. (hereinafter referred to as "ChainsGuard") conducts an in-depth security risk assessment of the CBK smart contract; the technical report submitted according to the assessment result is used for The security status of the smart contract makes security assessment and reinforcement recommendations. only limited to ChainsGuard and the internal personnel of the project party.

Version change record

Date	Version	Description	Modify by
2020-09-04	V1.1	Document creation	ChainsGuard Security Center

Catalogue

Disclaimer

The audit report is a technical security audit for the authorized party. The purpose of this audit is to provide the authorized party with a reference basis for conducting its business security assessment and optimization, The regulatory regime of the business model, or any other statement about the applicability of the application, as well as a statement or warranty that the application is in error-free behavior. This report cannot be used as a proof of that these tested systems and codes are absolutely secure and there are no other security risks.

The audit report only covers the code, installation packages and other materials provided by the authorized party, and its conclusion is only applicable to the corresponding version of the application. Once the relevant code, configuration, and operating environment change, the corresponding conclusion will no longer be applicable.

1 Introduction

1.1 Overview

This document includes the results of the audit performed by the Chains Guard Team on the CBK project, at the request of the CBK team. The goal of this audit is to review the smart contract code solidity implementation, s tudy potential security vulnerabilities, its general designand architecture, and uncover bugs that could compromise the software in production.

1.2 Audit Time

Evaluation test time	
Start time	2020-09-03
End time	2020-09-04

1.3 Audit Unit

Company Name	The Beijing ChainsGuard Network Technology Co., Ltd.
Web Site	https://www.chainsguard.com/

1.4 Audit Object

Contract Name	File SHA256
CBK	a9e8bd463776b210b29b78eba1354caddf0748acf8b1f4b861ad3aa58fd34426

2 Security Audit Summary

2.1 Vulnerability Statistics

Vulnerabilities	High risk	Medium risk	Low risk
0	0	0	1

[Note] A brief description of the hazard classification method is as follows

High: It directly causes the system to be controlled or the data to be destroyed. Once it occurs, it is a serious security event.

Medium: It may lead to the leakage of important information or may cause the system to be controlled.

Low: Non-critical information leaks or minor security issues generally do not lead to serious security incidents.

2.2 Audit items

For the ERC20 token contract, we focus on the audit of the following inspection items:

Attack surface	Check list	status	description

Reentrancy Attack	Cross-contract interaction	**Pass**	Unprotected sensitive functions call external contracts
	ETH Transfer	**Pass**	Unrestricted Gas transfer ETH has a hidden danger of reentry
Unauthorized access	Constructor does not match	**Pass**	Whether the contract name and constructor in the lower version do not match
	Privileged function exposure	**Pass**	Exposure of privileged functions caused by incorrect authentication methods
	tx.origin variable abuse	**Pass**	Whether the contract uses tx.origin for identity authentication
	Access control flaws	**Pass**	Unreasonable settings for the visibility of functions and state variables
Numerical overflow	Overflow & underflow	**Pass**	Does the contract have common overflow or underflow vulnerabilities
Race condition	Transaction order dependence	**Low risk**	Does the final state of the contract depend on the order of transactions
Denial of service	Unexpected transaction rollback	**Pass**	Is the contract vulnerable to revert cause denial of service
	Gas Price exceeded	**Pass**	Excessive Gas Price caused by excessive loop
call injection	call function abuse	**Pass**	The contract receives external input as a parameter of the call function
Fake recharge	Recharge result check	**Pass**	Whether the contract uses an incorrect method to check the recharge result
Miner privileges	Timestamp dependence	**Pass**	Does the contract rely on the timestamp to complete the main function
	fake-random number dependence	**Pass**	Does the contract rely on pseudo-random numbers to complete its main functions
Other checks	External input check	**Pass**	Whether the contract verifies the legality of external input
	Use untrusted libraries	**Information**	Whether the contract uses untrusted (unsafe) libraries
	Leakage of sensitive information	**Pass**	Does the contract have hidden dangers of leaking sensitive information

	Blackhole	Pass	Whether the contract locks ETH or tokens indefinitely
	Contract backdoor	Pass	Does the contract have a backdoor that can be controlled by the project party

3 Checklist

For smart contract source code audit, we focus on the detection of the following security points:

3.1 Reentrancy Attack

One of the main dangers of invoking external contracts is that they can take over the control process. In a reentrancy attack (also called a recursive call attack), a malicious contract will call back the calling contract before the first call to the function is completed. This may cause different calls of the function to interact in an undesirable way

Test result: **Pass**

3.2 Permission Control

Check whether the functions in the contract use public, private and other keywords correctly for visibility modification. Check whether the contract is correctly defined and use the modifier to restrict access to key functions to avoid unauthorized access.

Test result: **Pass**

3.3 Overflow & Underflow

The numerical processing in the smart contract needs to strictly check the arithmetic overflow/underflow problem, the conventional addition and subtraction arithmetic processing is easy to cause integer overflow or underflow, especially when processing the account amount of similar tokens, it is necessary to strictly judge the size of the account amount. Normally numerical calculation is recommended to use Zeppelin open source module SafeMath for processing.

Test result: **pass**

3.4 Call Injection

When using a low-level interface call() for contract interaction in a smart contract, the interface is improperly used. An attacker can control the parameters, method selectors, or execution byte contents of the call interface to perform dangerous operations such as unauthorized transfer, mint, and token burn.

Test result: **pass**

3.5 Race Conditions

Every transaction and smart contract calls requires miner to mine to con firm. Sharing a state between different functions may cause different proces sing results due to changes in the execution order.

Test result: **Low risk**

Vulnerability details: The location of the vulnerability is in the HxERC20# approve() function. The approval function can be called multiple times. Ethe reum In order to encourage miners to mine, miners can decide which trans actions to pack. In order to maximize benefits, miners usually choose to pa ck larger Gas Price transactions instead of relying on the order of transacti ons. Therefore, when the sender calls the approval function for the second time, if the consumer (attacker) knows the intention, and before the miner packs the second approval transaction, a higher Gas Price is used to spend the first authorization token . When the miner packs the second transactio n, a token authorization beyond the original intention will be formed.

Fix suggestion: add require((_allowed[msg.sender][spender] == 0) || (value == 0), "HxERC20: use increaseAllowance & decreaseAllowance instead"); co de to limit the second call, forcibly use increaseAllowance & decreaseAllowa nce function instead.

```
/**
 * @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
 * Beware that changing an allowance with this method brings the risk that someone may use both the old
 * and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this
 * race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards:
 * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
 * @param spender The address which will spend the funds.
 * @param value The amount of tokens to be spent.
 *
 * **ChainsGuard** Low risk: did not alleviate the hidden dangers of race conditions
 */
function approve(address spender, uint256 value) public returns (bool) {
    require(spender != address(0));
    // **ChainsGuard** require((_allowed[msg.sender][spender] == 0) || (value == 0), "HxERC20: use increaseAllowance & decreaseAllowance instead");

    _allowed[msg.sender][spender] = value;
    emit Approval(msg.sender, spender, value);
    return true;
}
```

3.6 Design Flaw

Unreasonable code logic processing or wrong coding will lead to abnormal contract execution, functional or security does not meet expectations, etc. Before the contract is officially deployed, the entire code needs to be checked for integrity and multi-faceted to ensure that the contract code logic is correct.

Test result: **pass**

3.7 DDoS Attack

"Unrecoverable malicious operations or controllable unlimited resource consumption" can all be called denial of service in smart contracts. An attacker may call a contract function by constructing malicious parameters, which may lead to logical processing problems that cannot be recovered by the deployed contract, and the service is denied The smart contract usually makes the entire code logic unable to continue execution and "stop service".

Test result: **pass**

3.8 Pseudo-random Number

The values such as block.timestamp, block.number, etc. in the smart contract are predictable. When the contract involves random number generation, do not use easily accessible variables or parameters as seeds to generate random numbers. A better method is to use The external service Oraclize performs random number generation and processing.

Test result: **pass**

3.9 Front-running

Every transaction and smart contract calls requires miners to confirm the mining. During this time period, it is extremely easy for an attacker to maliciously read the transaction or call details, and then use the high Gas miner incentive mechanism to give priority to their transactions or calls. form Front-Running.

Test result: **pass**

3.10 Short Address Attack

When the virtual machine parses the smart contract bytes, it will automatically filling the number of parameter bytes. The attacker can construct malformed data to make smart contract calls. Automatic byte filling will cause unintended parameter values to change.

Test result: **pass**

3.11 Data Privacy

The smart contract code needs to encrypt and protect the stored user's private information. Anyone can obtain the relevant information stored in the contract by querying the data on the chain. If the private information is not encrypted or the encryption is not strict, it is easy to cause privacy leakage.

Test result: **pass**

3.12 Contract backdoor

In the blockchain ecosystem, some project parties themselves do not have the ability to develop smart contracts. They usually choose some smart contract automation generation tools for one-click generation and automatic deployment on the blockchain. This opens up opportunities for hackers to insert backdoor codes into smart contracts. Once the backdoor code is insert into the smart contract, the hacker can use its backdoor to manipulate the contract arbitrarily, which will cause fatal harm to the project party.

Test result: **pass**

3.13 Code Optimization

Each step of smart contract code execution needs to spend corresponding gas. Non-standard code writing will cause unnecessary gas waste. Code l

ogic optimization can help the smart contract logic to be clearer, the execution efficiency is higher, and the Gas costs less.

Test result: Needs optimization

Missing return statement: The HxOwnable#acceptOwnership() function does not return true after receiving the owner, which will cause unexpected bugs in calls that rely on this result.

```
    // **ChainsGuard** Pass
    function acceptOwnership() public onlyNewOwner returns(bool) {
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
        newOwner = address(0);
        // **ChainsGuard** Missing return statement: return true;
    }
```

Redundant code: Consider remove the require(account != address(0)); code in the Roles#add() and Roles#remove() functions, because they will always call the same code in the Roles#has() function. The remaining code can save a small amount of gas.

```
    /**
     * @dev give an account access to this role
     *
     * **ChainsGuard** Pass
     */
    function add(Role storage role, address account) internal {
        require(account != address(0));// **ChainsGuard** Redundant code: Remove to save gas
        require(!has(role, account));// **ChainsGuard** ensure the account is not yet a member of the role

        role.bearer[account] = true;
    }

    /**
     * @dev remove an account's access to this role
     *
     * **ChainsGuard** Pass: Remove account assigned role
     */
    function remove(Role storage role, address account) internal {
        require(account != address(0));// **ChainsGuard** Redundant code: Remove to save gas
        require(has(role, account));// **ChainsGuard** ensure the account is a member of the role

        role.bearer[account] = false;
    }
```

4 Security Warnings

4.1 Owner & Pauser Private-Key Protect

Owner and Pauser Roles can freeze and lock any user account, and the frozen account cannot continue to transfer tokens. Excessive authority design means greater responsibility. Owner and Pauser should protect their private keys to prevent theft of private keys from destroying the ecosystem.

4.2 Use untrusted libraries

The upgradeable implementation in this contract is not trusted. The code for the upgrade contract is derived from the https://github.com/OpenZeppelin/openzeppelin-labs repository. The code of this repository is experimen

tal, and OpenZeppelin does not recommend these code is used in a produ ction environment.

```
    /**
     * @dev Fallback function allowing to perform a delegatecall
     * to the given implementation. This function will return
     * whatever the implementation call returns
     */
    function () payable external {
        address impl = implementation;
        require(impl != address(0));
        assembly {
            let ptr := mload(0x40)
            calldatacopy(ptr, 0, calldatasize)
            let result := delegatecall(gas, impl, ptr, calldatasize, 0, 0)
            let size := returndatasize
            returndatacopy(ptr, 0, size)

            switch result
            case 0 { revert(ptr, size) }
            default { return(ptr, size) }
        }
    }
```

5 Summary of Security Audit

This contract, code style is good, no exploitable security issues found. The overall estimated safety status is **warning status.**

The intelligent contract audit results only provide the actual basis for the authorizer to formulate corresponding security measures and solutions.

6 Smart Contract Code

```
1   pragma solidity ^0.4.24;
2
3   library HxSafeMath { //중요 1/3: 자식에서 compile 문제로 SafeMath->HxSafeMath
4       /**
5       * @dev Multiplies two unsigned integers, reverts on overflow.
6       */
7       function mul(uint256 a, uint256 b) internal pure returns (uint256) {
8           // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
9           // benefit is lost if 'b' is also tested.
10          // See: https://github.com/OpenZeppelin/openzeppelin-solidity/pull/522
11          if (a == 0) {
12              return 0;
13          }
14
15          uint256 c = a * b;
16          require(c / a == b);
17
18          return c;
19      }
20
```

```
21      /**
22      * @dev Integer division of two unsigned integers truncating the quotient, reverts on division by zero.
23      */
24      function div(uint256 a, uint256 b) internal pure returns (uint256) {
25          // Solidity only automatically asserts when dividing by 0
26          require(b > 0);
27          uint256 c = a / b;
28          // assert(a == b * c + a % b); // There is no case in which this doesn't hold
29
30          return c;
31      }
32
33      /**
34      * @dev Subtracts two unsigned integers, reverts on overflow (i.e. if subtrahend is greater than minuend).
35      */
36      function sub(uint256 a, uint256 b) internal pure returns (uint256) {
37          require(b <= a);
38          uint256 c = a - b;
39
40          return c;
41      }
```

```
42
43      /**
44      * @dev Adds two unsigned integers, reverts on overflow.
45      */
46      function add(uint256 a, uint256 b) internal pure returns (uint256) {
47          uint256 c = a + b;
48          require(c >= a);
49
50          return c;
51      }
52
53      /**
54      * @dev Divides two unsigned integers and returns the remainder (unsigned integer modulo),
55      * reverts when dividing by zero.
56      */
57      function mod(uint256 a, uint256 b) internal pure returns (uint256) {
58          require(b != 0);
59          return a % b;
60      }
61  }
62
```

```
63  library Roles {
64      struct Role {
65          mapping (address => bool) bearer;
66      }
67
68      /**
69       * @dev give an account access to this role
70       */
71      function add(Role storage role, address account) internal {
72          require(account != address(0));
73          require(!has(role, account));
74
75          role.bearer[account] = true;
76      }
77
78      /**
79       * @dev remove an account's access to this role
80       */
81      function remove(Role storage role, address account) internal {
82          require(account != address(0));
83          require(has(role, account));
```

```
84
85              role.bearer[account] = false;
86          }
87
88          /**
89           * @dev check if an account has this role
90           * @return bool
91           */
92          function has(Role storage role, address account) internal view returns (bool) {
93              require(account != address(0));
94              return role.bearer[account];
95          }
96      }
97
98      contract HxOwnable {    //중요:자식에서  compile 문제로  Ownable -> HxOwnable
99          address public owner;
100         address public newOwner;
101
102         event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
103
104         constructor() public {
```

```
105         owner = msg.sender;
106         newOwner = address(0);
107     }
108
109     modifier onlyOwner() {
110         require(msg.sender == owner);
111         _;
112     }
113     modifier onlyNewOwner() {
114         require(msg.sender != address(0));
115         require(msg.sender == newOwner);
116         _;
117     }
118
119     function isOwner(address account) public view returns (bool) {
120         if( account == owner ){
121             return true;
122         }
123         else {
124             return false;
125         }
```

```
126     }
127
128     function transferOwnership(address _newOwner) public onlyOwner {
129         require(_newOwner != address(0));
130         newOwner = _newOwner;
131     }
132
133     function acceptOwnership() public onlyNewOwner returns(bool) {
134         emit OwnershipTransferred(owner, newOwner);
135         owner = newOwner;
136         newOwner = address(0);
137     }
138 }
139
140 contract PauserRole is HxOwnable{
141     using Roles for Roles.Role;
142
143     event PauserAdded(address indexed account);
144     event PauserRemoved(address indexed account);
145
146     Roles.Role private _pausers;
```

```
147
148     constructor () internal {
149         _addPauser(msg.sender);
150     }
151
152     modifier onlyPauser() {
153         require(isPauser(msg.sender)|| isOwner(msg.sender));
154         _;
155     }
156
157     function isPauser(address account) public view returns (bool) {
158         return _pausers.has(account);
159     }
160
161     function addPauser(address account) public onlyPauser {
162         _addPauser(account);
163     }
164
165     function removePauser(address account) public onlyOwner {
166         _removePauser(account);
167     }
```

```
168
169     function renouncePauser() public {
170         _removePauser(msg.sender);
171     }
172
173     function _addPauser(address account) internal {
174         _pausers.add(account);
175         emit PauserAdded(account);
176     }
177
178     function _removePauser(address account) internal {
179         _pausers.remove(account);
180         emit PauserRemoved(account);
181     }
182 }
183
184 contract Pausable is PauserRole {
185     event Paused(address account);
186     event Unpaused(address account);
187
188     bool private _paused;
```

```
189
190     constructor () internal {
191         _paused = false;
192     }
193
194     /**
195      * @return true if the contract is paused, false otherwise.
196      */
197     function paused() public view returns (bool) {
198         return _paused;
199     }
200
201     /**
202      * @dev Modifier to make a function callable only when the contract is not paused.
203      */
204     modifier whenNotPaused() {
205         require(!_paused);
206         _;
207     }
208
209     /**
```

```
210      * @dev Modifier to make a function callable only when the contract is paused.
211      */
212     modifier whenPaused() {
213         require(_paused);
214         _;
215     }
216
217     /**
218      * @dev called by the owner to pause, triggers stopped state
219      */
220     function pause() public onlyPauser whenNotPaused {
221         _paused = true;
222         emit Paused(msg.sender);
223     }
224
225     /**
226      * @dev called by the owner to unpause, returns to normal state
227      */
228     function unpause() public onlyPauser whenPaused {
229         _paused = false;
230         emit Unpaused(msg.sender);
```

```
231     }
232 }
233
234 interface IERC20 {
235     function transfer(address to, uint256 value) external returns (bool);
236
237     function approve(address spender, uint256 value) external returns (bool);
238
239     function transferFrom(address from, address to, uint256 value) external returns (bool);
240
241     function totalSupply() external view returns (uint256);
242
243     function balanceOf(address who) external view returns (uint256);
244
245     function allowance(address owner, address spender) external view returns (uint256);
246
247     event Transfer(address indexed from, address indexed to, uint256 value);
248
249     event Approval(address indexed owner, address indexed spender, uint256 value);
250 }
251
```

```
252 contract HxERC20 is IERC20 { //중요:자식에서 compile 문제로 ERC20 -> HxERC20
253     using HxSafeMath for uint256;
254
255     mapping (address => uint256) internal _balances;
256
257     mapping (address => mapping (address => uint256)) internal _allowed;
258
259     uint256 private _totalSupply;
260
261     /**
262     * @dev Total number of tokens in existence
263     */
264     function totalSupply() public view returns (uint256) {
265         return _totalSupply;
266     }
267
268     /**
269     * @dev Gets the balance of the specified address.
270     * @param owner The address to query the balance of.
271     * @return An uint256 representing the amount owned by the passed address.
272     */
```

```
273     function balanceOf(address owner) public view returns (uint256) {
274         return _balances[owner];
275     }
276
277     /**
278      * @dev Function to check the amount of tokens that an owner allowed to a spender.
279      * @param owner address The address which owns the funds.
280      * @param spender address The address which will spend the funds.
281      * @return A uint256 specifying the amount of tokens still available for the spender.
282      */
283     function allowance(address owner, address spender) public view returns (uint256) {
284         return _allowed[owner][spender];
285     }
286
287     /**
288     * @dev Transfer token for a specified address
289     * @param to The address to transfer to.
290     * @param value The amount to be transferred.
291     */
292     function transfer(address to, uint256 value) public returns (bool) {
293         _transfer(msg.sender, to, value);
```

```
294         return true;
295     }
296
297     /**
298      * @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
299      * Beware that changing an allowance with this method brings the risk that someone may use both the old
300      * and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this
301      * race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards:
302      * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
303      * @param spender The address which will spend the funds.
304      * @param value The amount of tokens to be spent.
305      */
306     function approve(address spender, uint256 value) public returns (bool) {
307         require(spender != address(0));
308
309         _allowed[msg.sender][spender] = value;
310         emit Approval(msg.sender, spender, value);
311         return true;
312     }
313
314     /**
```

```
315     * @dev Transfer tokens from one address to another.
316     * Note that while this function emits an Approval event, this is not required as per the specification,
317     * and other compliant implementations may not emit the event.
318     * @param from address The address which you want to send tokens from
319     * @param to address The address which you want to transfer to
320     * @param value uint256 the amount of tokens to be transferred
321     */
322    function transferFrom(address from, address to, uint256 value) public returns (bool) {
323        _allowed[from][msg.sender] = _allowed[from][msg.sender].sub(value);
324        _transfer(from, to, value);
325        emit Approval(from, msg.sender, _allowed[from][msg.sender]);
326        return true;
327    }
328
329    /**
330     * @dev Increase the amount of tokens that an owner allowed to a spender.
331     * approve should be called when allowed_[_spender] == 0. To increment
332     * allowed value is better to use this function to avoid 2 calls (and wait until
333     * the first transaction is mined)
334     * From MonolithDAO Token.sol
335     * Emits an Approval event.
```

```
336      * @param spender The address which will spend the funds.
337      * @param addedValue The amount of tokens to increase the allowance by.
338      */
339     function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
340         require(spender != address(0));
341
342         _allowed[msg.sender][spender] = _allowed[msg.sender][spender].add(addedValue);
343         emit Approval(msg.sender, spender, _allowed[msg.sender][spender]);
344         return true;
345     }
346
347     /**
348      * @dev Decrease the amount of tokens that an owner allowed to a spender.
349      * approve should be called when allowed_[_spender] == 0. To decrement
350      * allowed value is better to use this function to avoid 2 calls (and wait until
351      * the first transaction is mined)
352      * From MonolithDAO Token.sol
353      * Emits an Approval event.
354      * @param spender The address which will spend the funds.
355      * @param subtractedValue The amount of tokens to decrease the allowance by.
356      */
```

```
357     function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
358         require(spender != address(0));
359
360         _allowed[msg.sender][spender] = _allowed[msg.sender][spender].sub(subtractedValue);
361         emit Approval(msg.sender, spender, _allowed[msg.sender][spender]);
362         return true;
363     }
364
365     /**
366     * @dev Transfer token for a specified addresses
367     * @param from The address to transfer from.
368     * @param to The address to transfer to.
369     * @param value The amount to be transferred.
370     */
371     function _transfer(address from, address to, uint256 value) internal {
372         require(to != address(0));
373
374         _balances[from] = _balances[from].sub(value);
375         _balances[to] = _balances[to].add(value);
376         emit Transfer(from, to, value);
377     }
```

```
378
379     /**
380      * @dev Internal function that mints an amount of the token and assigns it to
381      * an account. This encapsulates the modification of balances such that the
382      * proper events are emitted.
383      * @param account The account that will receive the created tokens.
384      * @param value The amount that will be created.
385      */
386     function _mint(address account, uint256 value) internal {
387         require(account != address(0));
388
389         _totalSupply = _totalSupply.add(value);
390         _balances[account] = _balances[account].add(value);
391         emit Transfer(address(0), account, value);
392     }
393
394     /**
395      * @dev Internal function that burns an amount of the token of a given
396      * account.
397      * @param account The account whose tokens will be burnt.
398      * @param value The amount that will be burnt.
```

```
399      */
400     function _burn(address account, uint256 value) internal {
401         require(account != address(0));
402
403         _totalSupply = _totalSupply.sub(value);
404         _balances[account] = _balances[account].sub(value);
405         emit Transfer(account, address(0), value);
406     }
407
408     /**
409      * @dev Internal function that burns an amount of the token of a given
410      * account, deducting from the sender's allowance for said account. Uses the
411      * internal burn function.
412      * Emits an Approval event (reflecting the reduced allowance).
413      * @param account The account whose tokens will be burnt.
414      * @param value The amount that will be burnt.
415      */
416     function _burnFrom(address account, uint256 value) internal {
417         _allowed[account][msg.sender] = _allowed[account][msg.sender].sub(value);
418         _burn(account, value);
419         emit Approval(account, msg.sender, _allowed[account][msg.sender]);
```

```
420     }
421 }
422
423 contract ERC20Pausable is HxERC20, Pausable {
424     function transfer(address to, uint256 value) public whenNotPaused returns (bool) {
425         return super.transfer(to, value);
426     }
427
428     function transferFrom(address from, address to, uint256 value) public whenNotPaused returns (bool) {
429         return super.transferFrom(from, to, value);
430     }
431
432     /*
433      * approve/increaseApprove/decreaseApprove can be set when Paused state
434      */
435
436     /*
437      * function approve(address spender, uint256 value) public whenNotPaused returns (bool) {
438      *       return super.approve(spender, value);
439      * }
440      *
```

```
441         * function increaseAllowance(address spender, uint addedValue) public whenNotPaused returns (bool
    success) {
442         *       return super.increaseAllowance(spender, addedValue);
443         * }
444         *
445         * function decreaseAllowance(address spender, uint subtractedValue) public whenNotPaused returns
    (bool success) {
446         *       return super.decreaseAllowance(spender, subtractedValue);
447         * }
448         */
449 }
450
451 contract ERC20Detailed is IERC20 {
452     string private _name;
453     string private _symbol;
454     uint8 private _decimals;
455
456     constructor (string memory name, string memory symbol, uint8 decimals) public {
457         _name = name;
458         _symbol = symbol;
459         _decimals = decimals;
```

```
460     }
461
462     /**
463      * @return the name of the token.
464      */
465     function name() public view returns (string memory) {
466         return _name;
467     }
468
469     /**
470      * @return the symbol of the token.
471      */
472     function symbol() public view returns (string memory) {
473         return _symbol;
474     }
475
476     /**
477      * @return the number of decimals of the token.
478      */
479     function decimals() public view returns (uint8) {
480         return _decimals;
```

```
481     }
482 }
483
484 contract CBK is ERC20Detailed, ERC20Pausable {
485
486     struct LockInfo {
487         uint256 _releaseTime;
488         uint256 _amount;
489     }
490
491     address public implementation;
492
493     mapping (address => LockInfo[]) public timelockList;
494     mapping (address => bool) public frozenAccount;
495
496     event Freeze(address indexed holder);
497     event Unfreeze(address indexed holder);
498     event Lock(address indexed holder, uint256 value, uint256 releaseTime);
499     event Unlock(address indexed holder, uint256 value);
500
501     modifier notFrozen(address _holder) {
```

```
502         require(!frozenAccount[_holder]);
503         _;
504     }
505
506     constructor() ERC20Detailed("Cobak Token", "CBK", 18) payable public    {
507
508         _mint(msg.sender, 100000000 * (10 ** 18));
509     }
510
511     function balanceOf(address owner) public view returns (uint256) {
512
513         uint256 totalBalance = super.balanceOf(owner);
514         if( timelockList[owner].length >0 ){
515             for(uint i=0; i<timelockList[owner].length;i++){
516                 totalBalance = totalBalance.add(timelockList[owner][i]._amount);
517             }
518         }
519
520         return totalBalance;
521     }
522
```

```
523     function transfer(address to, uint256 value) public notFrozen(msg.sender) returns (bool) {
524         if (timelockList[msg.sender].length > 0 ) {
525             _autoUnlock(msg.sender);
526         }
527         return super.transfer(to, value);
528     }
529
530     function transferFrom(address from, address to, uint256 value) public notFrozen(from) returns (bool) {
531         if (timelockList[from].length > 0) {
532             _autoUnlock(from);
533         }
534         return super.transferFrom(from, to, value);
535     }
536
537     function freezeAccount(address holder) public onlyPauser returns (bool) {
538         require(!frozenAccount[holder]);
539         frozenAccount[holder] = true;
540         emit Freeze(holder);
541         return true;
542     }
543
```

```
544     function unfreezeAccount(address holder) public onlyPauser returns (bool) {
545         require(frozenAccount[holder]);
546         frozenAccount[holder] = false;
547         emit Unfreeze(holder);
548         return true;
549     }
550
551     function lock(address holder, uint256 value, uint256 releaseTime) public onlyPauser returns (bool) {
552         require(_balances[holder] >= value,"There is not enough balances of holder.");
553         _lock(holder,value,releaseTime);
554
555         return true;
556     }
557
558     function transferWithLock(address holder, uint256 value, uint256 releaseTime) public onlyPauser returns
        (bool) {
559         _transfer(msg.sender, holder, value);
560         _lock(holder,value,releaseTime);
561         return true;
562     }
563
```

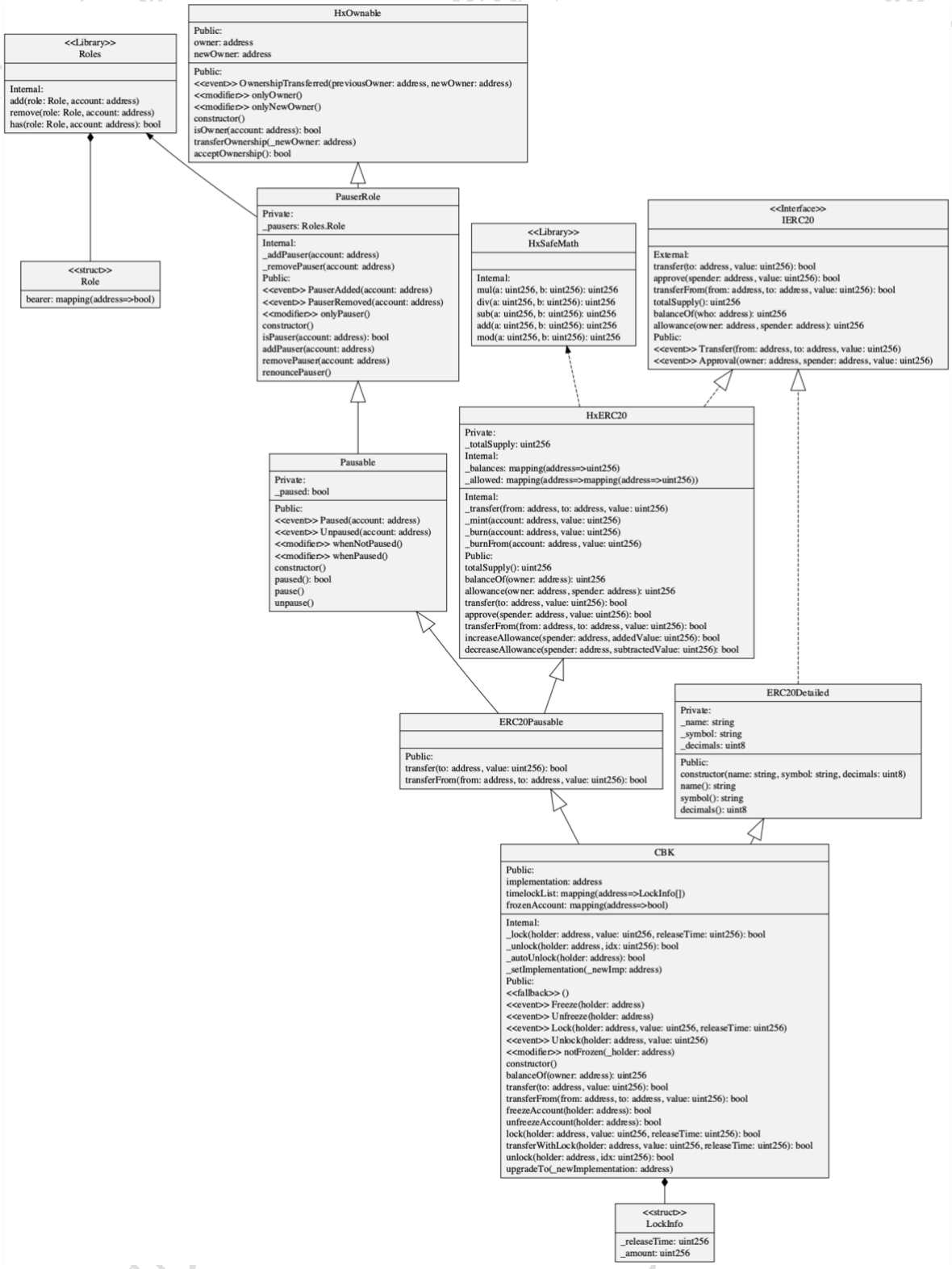
```
564     function unlock(address holder, uint256 idx) public onlyPauser returns (bool) {
565         require( timelockList[holder].length > idx, "There is not lock info.");
566         _unlock(holder,idx);
567         return true;
568     }
569
570     /**
571      * @dev Upgrades the implementation address
572      * @param _newImplementation address of the new implementation
573      */
574     function upgradeTo(address _newImplementation) public onlyOwner {
575         require(implementation != _newImplementation);
576         _setImplementation(_newImplementation);
577     }
578
579     function _lock(address holder, uint256 value, uint256 releaseTime) internal returns(bool) {
580         _balances[holder] = _balances[holder].sub(value);
581         timelockList[holder].push( LockInfo(releaseTime, value) );
582
583         emit Lock(holder, value, releaseTime);
584         return true;
```

```
585     }
586
587     function _unlock(address holder, uint256 idx) internal returns(bool) {
588         LockInfo storage lockinfo = timelockList[holder][idx];
589         uint256 releaseAmount = lockinfo._amount;
590
591         delete timelockList[holder][idx];
592         timelockList[holder][idx] = timelockList[holder][timelockList[holder].length.sub(1)];
593         timelockList[holder].length -=1;
594
595         emit Unlock(holder, releaseAmount);
596         _balances[holder] = _balances[holder].add(releaseAmount);
597
598         return true;
599     }
600
601     function _autoUnlock(address holder) internal returns(bool) {
602         for(uint256 idx =0; idx < timelockList[holder].length ; idx++ ) {
603             if (timelockList[holder][idx]._releaseTime <= now) {
604                 // If lockupinfo was deleted, loop restart at same position.
605                 if( _unlock(holder, idx) ) {
```

```
606                         idx -=1;
607                     }
608                 }
609             }
610             return true;
611     }
612
613     /**
614      * @dev Sets the address of the current implementation
615      * @param _newImp address of the new implementation
616      */
617     function _setImplementation(address _newImp) internal {
618         implementation = _newImp;
619     }
620
621     /**
622      * @dev Fallback function allowing to perform a delegatecall
623      * to the given implementation. This function will return
624      * whatever the implementation call returns
625      */
626     function () payable external {
```

```
627             address impl = implementation;
628             require(impl != address(0));
629             assembly {
630                 let ptr := mload(0x40)
631                 calldatacopy(ptr, 0, calldatasize)
632                 let result := delegatecall(gas, impl, ptr, calldatasize, 0, 0)
633                 let size := returndatasize
634                 returndatacopy(ptr, 0, size)
635
636                 switch result
637                 case 0 { revert(ptr, size) }
638                 default { return(ptr, size) }
639             }
640         }
641 }
```

7 Smart Contract UML



Appendix A. Explanation of Security Risk Status Levels

Security risk status statement	
1	**good status** The contract is in good running condition, and there are no or only sporadic low-risk security problems. At this time, as long as the existing security policy is maintained, the safety level requirements of the system can be met.
2	**warning status** There are some loopholes or security risks in the smart contract, which have not been used on a large scale. At this time, targeted reinforcement or improvement should be carried out according to the problems found in the evaluation, and then redeployment.
3	**serious status** Smart contract has been widely used. Serious loopholes or security problems that may seriously threaten the normal operation of the contract are found in the intelligent contract. At this time, measures should be taken immediately to redeploy the strengthened intelligent contract.
4	**emergency status** The tokens related to the intelligent contract have been opened for trading. Serious loopholes or security problems that may seriously threaten the normal operation of the contract have been found in the intelligent contract, which may cause serious damage to economic interests. At this point, should immediately stop the contract related token trading, immediately take measures to redeploy the strengthened intelligent contract.