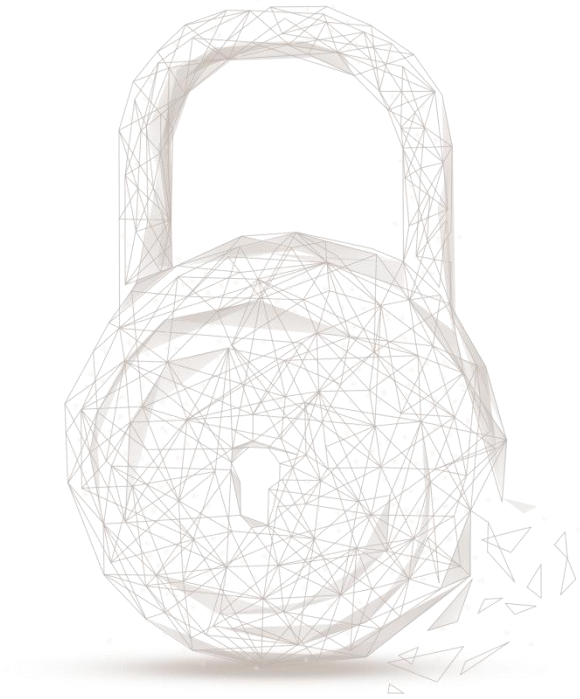
BEOSIN

Blockchain Security

Smart contract security audit report



Audit Number：202103101022

Report Query Name：HYN Bridge

Audit Project Name：HYN Bridge

Commit hash:

4b3990a1f9be76b6a89e7f818d8ccb88718345c6 (Origin version)

86f1dea731a9b06399b17d631deedf90d08e5a38 (Modified version)

Audit Link:

https://github.com/hyperion-hyn/hyn-bridge

Start Date：2021.03.08

Completion Date：2021.03.10

Overall Result：Pass

Audit Team: Beosin (Chengdu LianAn) Technology Co. Ltd.

Audit Categories and Results:

No.	Categories	Subitems	Results
1	Coding Conventions	Compiler Version Security	Pass
		Deprecated Items	Pass
		Redundant Code	Pass
		SafeMath Features	Pass
		require/assert Usage	Pass
		Gas Consumption	Pass
		Visibility Specifiers	Pass
		Fallback Usage	Pass
2	General Vulnerability	Integer Overflow/Underflow	Pass
		Reentrancy	Pass
		Pseudo-random Number Generator (PRNG)	Pass
		Transaction-Ordering Dependence	Pass

		DoS (Denial of Service)	Pass
		Access Control of Owner	Pass
		Low-level Function (call/delegatecall) Security	Pass
		Returned Value Security	Pass
		tx.origin Usage	Pass
		Replay Attack	Pass
		Overriding Variables	Pass
3	Business Security	Business Logics	Pass
		Business Implementations	Pass

Note: Audit results and suggestions in code comments

Audit Results Explained:

Beosin (Chengdu LianAn) Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of smart contracts project HYN Bridge, including Coding Standards, Security, and Business Logic. **The HYN Bridge project passed all audit items. The overall result is Pass.** The smart contract is able to function properly.

Audit Contents:

1. Coding Conventions

Check the code style that does not conform to Solidity code style.

1.1 Compiler Version Security

- Description: Check whether the code implementation of current contract contains the exposed solidity compiler bug.
- Result: Pass

1.2 Deprecated Items

- Description: Check whether the current contract has the deprecated items.
- Safety Recommendation: None
- Result: Pass

1.3 Redundant Code

- Description: Check whether the contract code has redundant codes.
- Safety Recommendation: None.
- Result: Pass

1.4 SafeMath Features

- Description: Check whether the SafeMath has been used. Or prevents the integer overflow/underflow in mathematical operation.
- Safety Recommendation: None
- Result: Pass

1.5 require/assert Usage

- Description: Check the use reasonability of 'require' and 'assert' in the contract.
- Safety Recommendation: None
- Result: Pass

1.6 Gas Consumption

- Description: Check whether the gas consumption exceeds the block gas limitation.
- Safety Recommendation: None
- Result: Pass

1.7 Visibility Specifiers

- Description: Check whether the visibility conforms to design requirement.
- Safety Recommendation: None
- Result: Pass

1.8 Fallback Usage

- Description: Check whether the Fallback function has been used correctly in the current contract.
- Safety Recommendation: None
- Result: Pass

2. General Vulnerability

Check whether the general vulnerabilities exist in the contract.

2.1 Integer Overflow/Underflow

- Description: Check whether there is an integer overflow/underflow in the contract and the calculation result is abnormal.
- Safety Recommendation: None
- Result: Pass

2.2 Reentrancy

- Description: An issue when code can call back into your contract and change state, such as withdrawing HT.
- Safety Recommendation: None
- Result: Pass

2.3 Pseudo-random Number Generator (PRNG)

- Description: Whether the results of random numbers can be predicted.
- Safety Recommendation: None
- Result: Pass

2.4 Transaction-Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Safety Recommendation: None
- Result: Pass

2.5 DoS (Denial of Service)

- Description: Whether exist DoS attack in the contract which is vulnerable because of unexpected reason.
- Safety Recommendation: None
- Result: Pass

2.6 Access Control of Owner

- Description: Whether the owner has excessive permissions, such as malicious issue, modifying the balance of others.
- Safety Recommendation: None
- Result: Pass

2.7 Low-level Function (call/delegatecall) Security

- Description: Check whether the usage of low-level functions like call/delegatecall have vulnerabilities.
- Safety Recommendation: None
- Result: Pass

2.8 Returned Value Security

- Description: Check whether the function checks the return value and responds to it accordingly.
- Safety Recommendation: None
- Result: Pass

2.9 tx.origin Usage

- Description: Check the use secure risk of 'tx.origin' in the contract.
- Safety Recommendation: None
- Result: Pass

2.10 Replay Attack

- Description: Check whether the implement possibility of Replay Attack exists in the contract.
- Safety Recommendation: None
- Result: Pass

2.11 Overriding Variables

- Description: Check whether the variables have been overridden and lead to wrong code execution.
- Safety Recommendation: None
- Result: Pass

3. Business Audit

3.1 The BridgedToken Contract

3.1.1 Basic contract information

- Description: This contract token implements the basic functions of ERC20 standard tokens, and token holders can call corresponding functions for token transfer, approve and other operations. And declare a *ethTokenAddr* for storing other tokens contract address.

```
pragma solidity 0.5.17;

import "@openzeppelin/contracts/token/ERC20/ERC20Detailed.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20Burnable.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20Mintable.sol";

contract BridgedToken is ERC20Burnable, ERC20Detailed, ERC20Mintable {
    address public ethTokenAddr;
    constructor(
        address _ethTokenAddr,
        string memory name,
        string memory symbol,
        uint8 decimals
    ) public ERC20Detailed(name, symbol, decimals) {
        ethTokenAddr = _ethTokenAddr;
    }
}
```

Figure 1 Source code of BridgedToken Contract

- Related functions: *name, symbol, decimals, balanceOf, transfer, transferFrom, allowance, approve, burn, burnFrom, mint, increaseAllowance, decreaseAllowance*
- Result: Pass

3.2 The MyERC20 Contract

3.2.1 Basic contract information

- Description: This contract token implements the basic functions of ERC20 standard tokens, and token holders can call corresponding functions for token transfer, approve and other operations.

```
pragma solidity 0.5.17;

import "@openzeppelin/contracts/token/ERC20/ERC20Detailed.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20Burnable.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20Mintable.sol";

contract MyERC20 is ERC20Burnable, ERC20Detailed, ERC20Mintable {
    constructor(
        string memory name,
        string memory symbol,
        uint8 decimals
    ) public ERC20Detailed(name, symbol, decimals) {}
}

```

Figure 2 Source code of MyERC20 Contract

- Related functions: *name, symbol, decimals, balanceOf, transfer, transferFrom, allowance, approve, burn, burnFrom, mint, increaseAllowance, decreaseAllowance*
- Result: Pass

3.3 The TokenManager Contract

3.3.1 *auth* permission management

- Description: The contract manager owner can call *rely* and *deny* functions to grant or revoke the *auth* permission of the specified address.

```
15      function rely(address guy) external onlyOwner {
16          wards[guy] = 1;
17      }
18
19      function deny(address guy) external onlyOwner {
20          require(guy != owner(), "TokenManager/cannot deny the owner");
21          wards[guy] = 0;
22      }
23
24      // both owner and admin must approve
25      modifier auth {
26          require(wards[msg.sender] == 1, "TokenManager/not-authorized");
27          _;
28      }
```

Figure 3 Source code of *rely* and *deny*

- Related functions: *rely*, *deny*
- Result: Pass

3.3.2 *addToken* function

- Description: The auth of the contract can call *addToken* function to create a bridge token and bind it with eth token, then add the caller as a minter of the bridge token. It is required that the eth token address is not 0 address and is not bound to other addresses.

```
35      function addToken(
36          address ethTokenAddr,
37          string memory name,
38          string memory symbol,
39          uint8 decimals
40      ) public auth returns (address) {
41          require(
42              ethTokenAddr != address(0),
43              "TokenManager/ethToken is a zero address"
44          );
45          require(
46              mappedTokens[ethTokenAddr] == address(0),
47              "TokenManager/ethToken already mapped"
48          );
49
50          BridgedToken bridgedToken = new BridgedToken(
51              ethTokenAddr,
52              name,
53              symbol,
54              decimals
55          );
56          address bridgedTokenAddr = address(bridgedToken);
57
58          // store the mapping and created address
59          mappedTokens[ethTokenAddr] = bridgedTokenAddr;
60
61          // assign minter role to the caller
62          bridgedToken.addMinter(msg.sender);
63
64          emit TokenMapAck(ethTokenAddr, bridgedTokenAddr);
65          return bridgedTokenAddr;
66      }
```

Figure 4 Source code of *addtoken*

- Related functions: *addtoken, addMinter*
- Result: Pass

3.3.3 *registerToken* function

- Description: The auth of the contract can call *registerToken* function to bind the eth token address with another token address. It is required that the eth token address is not 0 address and is not bound to other addresses.

```
73      function registerToken(address ethTokenAddr, address hecoTokenAddr)
74          public
75          auth
76          returns (bool)
77      {
78          require(
79              ethTokenAddr != address(0),
80              "TokenManager/ethTokenAddr is a zero address"
81          );
82          require(
83              mappedTokens[ethTokenAddr] == address(0),
84              "TokenManager/ethTokenAddr already mapped"
85          );
86
87          // store the mapping and created address
88          mappedTokens[ethTokenAddr] = hecoTokenAddr;
89      }
```

Figure 5 Source code of *registertoken*

- Related functions: *registerToken*
- Result: Pass

3.3.4 *removeToken* function

- Description: The auth of the contract can call *removeToken* function to unbind the eth token address. It is required to enter the current total amount of its bound tokens as a verification.

```
96      function removeToken(address ethTokenAddr, uint256 supply) public auth {
97          require(
98              mappedTokens[ethTokenAddr] != address(0),
99              "TokenManager/ethToken mapping does not exists"
100         );
101         IERC20 oneToken = IERC20(mappedTokens[ethTokenAddr]);
102         require(
103             oneToken.totalSupply() == supply,
104             "TokenManager/remove has non-zero supply"
105         );
106         delete mappedTokens[ethTokenAddr];
107     }
```

Figure 6 Source code of *removeToken*

- Related functions: *removeToken*
- Result: Pass

3.4 The ERC20AtlasManager Contract

3.4.1 *lockToken* function

- Description: The contract implements the *lockToken* function to lock user tokens. By pre-authorizing the contract address, the user can call this function to send the specified token to the contract and lock it.

```
50      function lockToken(
51          address hynTokenAddr,
52          uint256 amount,
53          address recipient
54      ) public {
55          require(
56              recipient != address(0),
57              "AtlasManager/recipient is a zero address"
58          );
59          require(amount > 0, "AtlasManager/zero token locked");
60          IERC20 hynToken = IERC20(hynTokenAddr);
61          uint256 _balanceBefore = hynToken.balanceOf(msg.sender);
62          hynToken.safeTransferFrom(msg.sender, address(this), amount);
63          uint256 _balanceAfter = hynToken.balanceOf(msg.sender);
64          uint256 _actualAmount = _balanceBefore.sub(_balanceAfter);
65          emit Locked(address(hynToken), msg.sender, _actualAmount, recipient);
66      }
```

Figure 7 Source code of *lockToken*

- Related functions: *lockToken, safeTransferFrom*
- Result: Pass

3.4.2 *lockTokenFor* function

- Description: The owner of the contract can call *lockTokenFor* function to lock any user-specified number of tokens to the contract address (requires the user to authorize the contract in advance).

```
75      function lockTokenFor(
76          address hynTokenAddr,
77          address userAddr,
78          uint256 amount,
79          address recipient
80      ) public onlyOwner {
81          require(
82              recipient != address(0),
83              "AtlasManager/recipient is a zero address"
84          );
85          require(amount > 0, "AtlasManager/zero token locked");
86          IERC20 hynToken = IERC20(hynTokenAddr);
87          uint256 _balanceBefore = hynToken.balanceOf(userAddr);
88          hynToken.safeTransferFrom(userAddr, address(this), amount);
89          uint256 _balanceAfter = hynToken.balanceOf(userAddr);
90          uint256 _actualAmount = _balanceBefore.sub(_balanceAfter);
91          emit Locked(address(hynToken), userAddr, _actualAmount, recipient);
92      }
```

Figure 8 Source code of *lockTokenFor*

- Safety recommendation: Owner authority is too large, the user's assets will be extremely risky if the owner's private key is lost.
- Repair result: It has been repaired. As the figure shown below, the project party has modified the authority to call the function and stated that it will hand over the authority to the MultiSigWallet contract for governance.

```
75      function lockTokenFor(
76          address hynTokenAddr,
77          address userAddr,
78          uint256 amount,
79          address recipient
80      ) public onlyWallet {
81          require(
82              recipient != address(0),
83              "AtlasManager/recipient is a zero address"
84          );
85          require(amount > 0, "AtlasManager/zero token locked");
86          IERC20 hynToken = IERC20(hynTokenAddr);
87          uint256 _balanceBefore = hynToken.balanceOf(userAddr);
88          hynToken.safeTransferFrom(userAddr, address(this), amount);
89          uint256 _balanceAfter = hynToken.balanceOf(userAddr);
90          uint256 _actualAmount = _balanceBefore.sub(_balanceAfter);
91          emit Locked(address(hynToken), userAddr, _actualAmount, recipient);
92      }
```

Figure 9 Source code of *lockTokenFor*

- Related functions: *lockTokenFor, safeTransferFrom*
- Result: Pass

3.4.3 *unlockToken* function

- Description: The owner of the contract can call *unlockToken* to send the locked tokens in the contract to the specified address. Need to enter the receiptId every time to unlock, and the receiptId can only be used once.

```
101     function unlockToken(
102         address hynTokenAddr,
103         uint256 amount,
104         address recipient,
105         bytes32 receiptId
106     ) public onlyOwner {
107         require(
108             !usedEvents_[receiptId],
109             "AtlasManager/The burn event cannot be reused"
110         );
111         IERC20 hynToken = IERC20(hynTokenAddr);
112         usedEvents_[receiptId] = true;
113         hynToken.safeTransfer(recipient, amount);
114         emit Unlocked(hynTokenAddr, amount, recipient, receiptId);
115     }
```

Figure 10 Source code of *unlockToken*

- Safety recommendation: Owner authority is too large, the user's assets will be extremely risky if the owner's private key is lost.

- Repair result: It has been repaired. As the figure shown below, the project party has modified the authority to call the function and stated that it will hand over the authority to the MultiSigWallet contract for governance.

```
101     function unlockToken(
102         address hynTokenAddr,
103         uint256 amount,
104         address recipient,
105         bytes32 receiptId
106     ) public onlyWallet {
107         require(
108             !usedEvents_[receiptId],
109             "AtlasManager/The burn event cannot be reused"
110         );
111         IERC20 hynToken = IERC20(hynTokenAddr);
112         usedEvents_[receiptId] = true;
113         hynToken.safeTransfer(recipient, amount);
114         emit Unlocked(hynTokenAddr, amount, recipient, receiptId);
115     }
```

Figure 11 Source code of *unlockToken*

- Related functions: *unlockToken, safeTransfer*
- Result: Pass

3.4.4 *lockHyn* function

- Description: The contract implements the *lockHyn* function to lock ETH(or other native currencies like HT), the user can call this function to send the ETH to the contract and lock it.

```
122     function lockHyn(
123         uint256 amount,
124         address recipient
125     ) public payable {
126         require(
127             recipient != address(0),
128             "AtlasManager/recipient is a zero address"
129         );
130         require(msg.value == amount, "HynManager/zero token locked");
131         emit Locked(address(HYN_ADDRESS), msg.sender, amount, recipient);
132     }
```

Figure 12 Source code of *lockHyn*

- Related functions: *lockHyn*
- Result: Pass

3.4.5 *unlockHyn* function

- Description: The owner of the contract can call *unlockHyn* to send the locked ETH in the contract to the specified address. Need to enter the receiptId every time to unlock, and the receiptId can only be used once.

```
140     function unlockHyn(
141         uint256 amount,
142         address payable recipient,
143         bytes32 receiptId
144     ) public onlyOwner {
145         require(
146             !usedEvents_[receiptId],
147             "AtlasManager/The burn event cannot be reused"
148         );
149         usedEvents_[receiptId] = true;
150         recipient.transfer(amount);
151         emit Unlocked(address(HYN_ADDRESS), amount, recipient, receiptId);
152     }
```

Figure 13 Source code of *unlockHyn*

- Safety recommendation: Owner authority is too large, the user's assets will be extremely risky if the owner's private key is lost.
- Repair result: It has been repaired. As the figure shown below, the project party has modified the authority to call the function and stated that it will hand over the authority to the MultiSigWallet contract for governance.

```
140     function unlockHyn(
141         uint256 amount,
142         address payable recipient,
143         bytes32 receiptId
144     ) public onlyWallet {
145         require(
146             !usedEvents_[receiptId],
147             "AtlasManager/The burn event cannot be reused"
148         );
149         usedEvents_[receiptId] = true;
150         recipient.transfer(amount);
151         emit Unlocked(address(HYN_ADDRESS), amount, recipient, receiptId);
152     }
```

Figure 14 Source code of *unlockHyn*

- Related functions: *unlockHyn*
- Result: Pass

3.5 The ERC20HecoManager Contract

3.5.1 *addToken* function

- Description: The contract import the TokenManager contract, The manager of the contract can call this function, the function is the same as *addToken* in TokenManager.

```
54      function addToken(
55          address tokenManager,
56          address hynTokenAddr,
57          string memory name,
58          string memory symbol,
59          uint8 decimals
60      ) public onlyWallet {
61          address heco20TokenAddr = TokenManager(tokenManager).addToken(
62              hynTokenAddr,
63              name,
64              symbol,
65              decimals
66          );
67          mappings[hynTokenAddr] = heco20TokenAddr;
68      }
```

Figure 15 Source code of *addToken*

- Related functions: *addToken*
- Result: Pass

3.5.2 *removeToken* function

- Description:The contract import the TokenManager contract, The owner of the contract can call this function, the function is the same as *removeToken* in TokenManager.

```
75      function removeToken(address tokenManager, address hynTokenAddr) public onlyOwner {
76          TokenManager(tokenManager).removeToken(hynTokenAddr, 0);
77      }
```

Figure 17 Source code of *removeToken*

- Safety recommendation: Owner authority is too large, the user's assets will be extremely risky if the owner's private key is lost.
- Repair result: It has been repaired. As the figure shown below, the project party has modified the authority to call the function and stated that it will hand over the authority to the MultiSigWallet contract for governance.

```
75      function removeToken(address tokenManager, address hynTokenAddr) public onlyWallet {
76          TokenManager(tokenManager).removeToken(hynTokenAddr, 0);
77      }
```

Figure 18 Source code of *removeToken*

- Related functions: *removeToken*
- Result: Pass

3.5.3 *burnToken* function

- Description: The contract implements the *burnToken* function to destroy tokens. After pre-authorization, the user can call this function to destroy a specified amount of specified tokens.

```
85      function burnToken(
86          address heco20Token,
87          uint256 amount,
88          address recipient
89      ) public {
90          BurnableToken(heco20Token).burnFrom(msg.sender, amount);
91          emit Burned(heco20Token, msg.sender, amount, recipient);
92      }
```

Figure 19 Source code of *burnToken*

- Related functions: *burnToken*
- Result: Pass

3.5.4 *mintToken* function

- Description: The owner of the contract can call this function to mint a specified amount of specified tokens to a specified address. Need to enter the receiptId every time to mint, and the receiptId can only be used once.

```
101     function mintToken(
102         address heco20Token,
103         uint256 amount,
104         address recipient,
105         bytes32 receiptId
106     ) public onlyOwner {
107         require(
108             !usedEvents_[receiptId],
109             "HecoManager/The lock event cannot be reused"
110         );
111         usedEvents_[receiptId] = true;
112         MintableToken(heco20Token).mint(recipient, amount);
113         emit Minted(heco20Token, amount, recipient, receiptId);
114     }
```

Figure 20 Source code of *mintToken*

- Safety recommendation: Owner authority is too large, unlimited coins can be minted.
- Repair result: It has been repaired. As the figure shown below, the project party has modified the authority to call the function and stated that it will hand over the authority to the MultiSigWallet contract for governance.

```
101     function mintToken(
102         address heco20Token,
103         uint256 amount,
104         address recipient,
105         bytes32 receiptId
106     ) public onlyWallet {
107         require(
108             !usedEvents_[receiptId],
109             "HecoManager/The lock event cannot be reused"
110         );
111         usedEvents_[receiptId] = true;
112         MintableToken(heco20Token).mint(recipient, amount);
113         emit Minted(heco20Token, amount, recipient, receiptId);
114     }
```

Figure 21 Source code of *mintToken*

- Related functions: *mintToken*
- Result: Pass

3.6 The Migrations Contract

3.6.1 *setCompleted* function

- Description: The owner of the contract can call this function to set the latest migration completed time.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity >=0.4.25 <0.7.0;
3
4   contract Migrations {
5     address public owner;
6     uint public last_completed_migration;
7
8     modifier restricted() {
9       if (msg.sender == owner) _;
10    }
11
12    constructor() public {
13      owner = msg.sender;
14    }
15
16    function setCompleted(uint completed) public restricted {
17      last_completed_migration = completed;
18    }
19  }
```

Figure 22 Source code of *setCompleted*

- Related functions: *setCompleted*
- Result: Pass

3.7 The MultiSigWallet Contract

For the governance contract, the project party declares that the management authority of the ERC20HecoManager and ERC20AtlasManager contracts will be transferred to this contract. Run the project through the submit-confirm-execute mode.

3.7.1 *addOwner* function

- Description: The contract implements the *addOwner* function to assign owner competence to the specified address. Calling this function will first check whether the address already has owner competence, whether it is a 0 address, and whether the number of owners reaches the upper limit.

```
120     function addOwner(address owner)
121         public
122         onlyWallet
123         ownerDoesNotExist(owner)
124         notNull(owner)
125         validRequirement(owners.length + 1, required)
126     {
127         isOwner[owner] = true;
128         owners.push(owner);
129         emit OwnerAddition(owner);
130     }
```

Figure 23 Source code of *addOwner*

- Related functions: *addOwner, ownerDoesNotExist, notNull, validRequirement*
- Result: Pass

3.7.2 *removeOwner* function

- Description: The contract implements the *removeOwner* function to remove the owner competence of the specified address. Calling this function will first check whether the address has an owner; if the *required* value is greater than the number of owners after the removal, the *changeRequirement* function will be called to modify the *required* value.

```
134     function removeOwner(address owner) public onlyWallet ownerExists(owner) {
135         isOwner[owner] = false;
136         for (uint256 i = 0; i < owners.length - 1; i++)
137             if (owners[i] == owner) {
138                 owners[i] = owners[owners.length - 1];
139                 break;
140             }
141         owners.length -= 1;
142         if (required > owners.length) changeRequirement(owners.length);
143         emit OwnerRemoval(owner);
144     }
```

Figure 24 Source code of *removeOwner*

- Related functions: *removeOwner, changeRequirement, ownerExists*
- Result: Pass

3.7.3 *replaceOwner* function

- Description: The contract implements the *replaceOwner* function to transfer the owner competence of an address to a new address. Before calling this function, it will first check whether the two addresses have owner competence.

```
149     function replaceOwner(address owner, address newOwner)
150         public
151         onlyWallet
152         ownerExists(owner)
153         ownerDoesNotExist(newOwner)
154     {
155         for (uint256 i = 0; i < owners.length; i++)
156             if (owners[i] == owner) {
157                 owners[i] = newOwner;
158                 break;
159             }
160         isOwner[owner] = false;
161         isOwner[newOwner] = true;
162         emit OwnerRemoval(owner);
163         emit OwnerAddition(newOwner);
164     }
```

Figure 25 Source code of *replaceOwner*

- Related functions: *replaceOwner, ownerExists, ownerDoesNotExist*
- Result: Pass

3.7.4 *changeRequirement* function

- Description: The contract implements *changeRequirement* function to modify the required value. Before modification, it will determine whether the modified value is legal, it cannot be greater than the number of owners, and cannot be modified to 0.

```
168     function changeRequirement(uint256 _required)
169         public
170         onlyWallet
171         validRequirement(owners.length, _required)
172     {
173         required = _required;
174         emit RequirementChange(_required);
175     }
```

Figure 26 Source code of *changeRequirement*

- Related functions: *changeRequirement, validRequirement*

- Result: Pass

3.7.5 *submitTransaction* function

- Description: The contract implements the *submitTransaction* function to submit a transaction. After submission, the *confirmTransaction* function is called for confirmation. Because the *confirmTransaction* function can only be called by the owner, ordinary users cannot call the *submitTransaction* function.

```
182     function submitTransaction(
183         address destination,
184         uint256 value,
185         bytes memory data
186     ) public returns (uint256 transactionId) {
187         transactionId = addTransaction(destination, value, data);
188         confirmTransaction(transactionId);
189     }
```

Figure 27 Source code of *submitTransaction*

- Related functions: *submitTransaction, addTransaction, confirmTransaction*
- Result: Pass

3.7.6 *confirmTransaction* function

- Description: The contract implements the *confirmTransaction* function to confirm a transaction. Before calling this function, it will check whether the caller is the owner, whether the transaction exists, and whether it has been confirmed, after confirmation, the *executeTransaction* function will be called to execute the transaction.

```
193     function confirmTransaction(uint256 transactionId)
194         public
195         ownerExists(msg.sender)
196         transactionExists(transactionId)
197         notConfirmed(transactionId, msg.sender)
198     {
199         confirmations[transactionId][msg.sender] = true;
200         emit Confirmation(msg.sender, transactionId);
201         executeTransaction(transactionId);
202     }
```

Figure 28 Source code of *confirmTransaction*

- Related functions: *confirmTransaction, ownerExists, transactionExists, notConfirmed, executeTransaction*
- Result: Pass

3.7.7 *revokeConfirmation* function

- Description: The contract implements the revokeConfirmation function to deny a transaction. Before calling this function, it will check whether the caller is the owner, whether the transaction exists, and whether it has been executed.

```
206     function revokeConfirmation(uint256 transactionId)
207         public
208         ownerExists(msg.sender)
209         confirmed(transactionId, msg.sender)
210         notExecuted(transactionId)
211     {
212         confirmations[transactionId][msg.sender] = false;
213         emit Revocation(msg.sender, transactionId);
214     }
```

Figure 29 Source code of *revokeConfirmation*

- Related functions: *revokeConfirmation, ownerExists, confirmed, notExecuted*
- Result: Pass

3.7.8 *executeTransaction* function

- Description: The contract implements *executeTransaction* function to execute a transaction. Before executing the transaction, it will first check whether the caller is the owner, whether it has been confirmed, whether it has been executed, and whether the number of confirmations is not less than *required*. Then call the internal function external_call to execute the transaction.

```
218     function executeTransaction(uint256 transactionId)
219         public
220         ownerExists(msg.sender)
221         confirmed(transactionId, msg.sender)
222         notExecuted(transactionId)
223     {
224         if (isConfirmed(transactionId)) {
225             Transaction storage txn = transactions[transactionId];
226             txn.executed = true;
227             if (
228                 external_call(
229                     txn.destination,
230                     txn.value,
231                     txn.data.length,
232                     txn.data
233                 )
234             ) emit Execution(transactionId);
235             else {
236                 emit ExecutionFailure(transactionId);
237                 txn.executed = false;
238             }
239         }
240     }
```

Figure 30 Source code of *executeTransaction*

- Related functions: *executeTransaction, ownerExists, confirmed, notExecuted, isConfirmed, external_call*
- Result: Pass

3.7.9 Related query function

- Description: The contract implements *isConfirmed* function to query whether the specified transaction meets the confirmation number requirement. *getConfirmationCount* function to query the current number of confirmations for a specified transaction. *getTransactionCount* function to query the number of successfully submitted transactions. *getOwners* function to query the address of current owner. *getConfirmations* function to query confirms the owner of the specified transaction. *getTransactionIds* function to query the successfully submitted transactions within the specified interval.

```
272     function isConfirmed(uint256 transactionId) public view returns (bool) {
273         uint256 count = 0;
274         for (uint256 i = 0; i < owners.length; i++) {
275             if (confirmations[transactionId][owners[i]]) count += 1;
276             if (count == required) return true;
277         }
278     }
```

Figure 31 Source code of *isConfirmed*

```
310     function getConfirmationCount(uint256 transactionId)
311         public
312         view
313         returns (uint256 count)
314     {
315         for (uint256 i = 0; i < owners.length; i++)
316             if (confirmations[transactionId][owners[i]]) count += 1;
317     }
```

Figure 32 Source code of *getConfirmationCount*

```
323      function getTransactionCount(bool pending, bool executed)
324          public
325          view
326          returns (uint256 count)
327      {
328          for (uint256 i = 0; i < transactionCount; i++)
329              if (
330                  (pending && !transactions[i].executed) ||
331                  (executed && transactions[i].executed)
332              ) count += 1;
333      }
```

Figure 33 Source code of *getTransactionCount*

```
337     function getOwners() public view returns (address[] memory) {
338         return owners;
339     }
340
341     /// @dev Returns array with owner addresses, which confirmed transaction.
342     /// @param transactionId Transaction ID.
343     /// @return Returns array of owner addresses.
344     function getConfirmations(uint256 transactionId)
345         public
346         view
347         returns (address[] memory _confirmations)
348     {
349         address[] memory confirmationsTemp = new address[](owners.length);
350         uint256 count = 0;
351         uint256 i;
352         for (i = 0; i < owners.length; i++)
353             if (confirmations[transactionId][owners[i]]) {
354                 confirmationsTemp[count] = owners[i];
355                 count += 1;
356             }
357         _confirmations = new address[](count);
358         for (i = 0; i < count; i++) _confirmations[i] = confirmationsTemp[i];
359     }
```

Figure 34 Source code of *getOwners* and *getConfirmations*

```
367     function getTransactionIds(
368         uint256 from,
369         uint256 to,
370         bool pending,
371         bool executed
372     ) public view returns (uint256[] memory _transactionIds) {
373         uint256[] memory transactionIdsTemp = new uint256[](transactionCount);
374         uint256 count = 0;
375         uint256 i;
376         for (i = 0; i < transactionCount; i++)
377             if (
378                 (pending && !transactions[i].executed) ||
379                 (executed && transactions[i].executed)
380             ) {
381                 transactionIdsTemp[count] = i;
382                 count += 1;
383             }
384         _transactionIds = new uint256[](to - from);
385         for (i = from; i < to; i++)
386             _transactionIds[i - from] = transactionIdsTemp[i];
387     }
388 }
```

Figure 35 Source code of *getTransactionIds*

- Related functions: *isConfirmed, getConfirmationCount, getTransactionCount, getOwners, getConfirmations, getTransactionIds*
- Result: Pass

4. Conclusion

Beosin(ChengduLianAn) conducted a detailed audit on the design and code implementation of the smart contracts project HYN Bridge. All the issues found during the audit have been written into this audit report. The overall audit result of the smart contract project HYN Bridge is **Pass**.

Official Website

https://lianantech.com

E-mail

vaas@lianantech.com

Twitter

https://twitter.com/Beosin_com