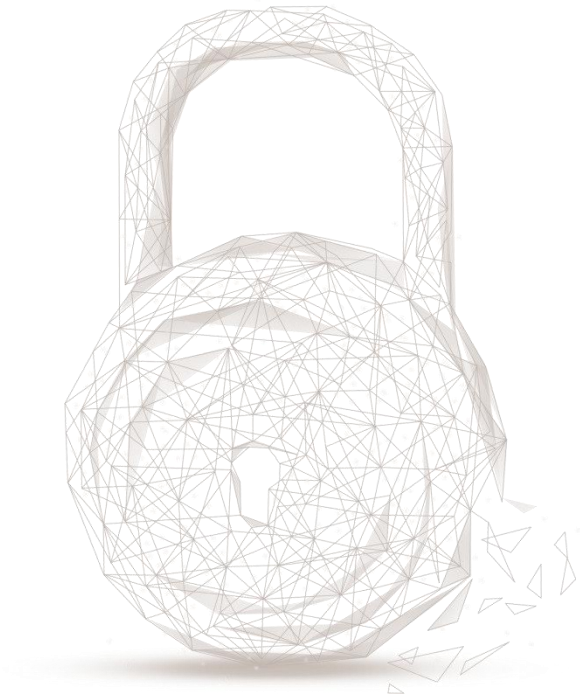
BEOSIN

Blockchain Security

Smart contract security audit report



Audit Number：202103051900

Report Query Name：Hot Cross BSC Bridge V1

Audit Project Name：Hot Cross BSC Bridge V1

Initial audit file Name:

proj-bsc-bridge-v1-solidity.zip

Initial audit file hash(SHA256):

0573C1B008B1AA2E15F52B1132D82AA5F2F2053118BDE2154FA8DA24093877E7

Start Date：2021.03.02

Completion Date：2021.03.05

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 Hot Cross BSC Bridge V1, including Coding Standards, Security, and Business Logic. **The Hot Cross BSC Bridge V1 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.

  The smart contracts of this project specify their corresponding minimum compiler version. Among them, using this version of the compiler to compile the BEP20 contract(./contracts/home/BEP20.sol), the compiler warning shown in the figure below will occur.

```
1070 contract ERC20 is Context, IERC20 {
1071     using SafeMath for uint256;
1072
1073     mapping (address => uint256) private _balances;
1074
1075     mapping (address => mapping (address => uint256)) private _allowances;
1076
1077     uint256 private _totalSupply;
1078
1079     string private _name;
1080     string private _symbol;
1081     uint8 private _decimals;
1082
1083     /**
1084      * @dev Sets the values for {name} and {symbol}, initializes {decimals} with
1085      * a default value of 18.
1086      *
1087      * To select a different value for {decimals}, use {_setupDecimals}.
1088      *
1089      * All three of these values are immutable: they can only be set once during
1090      * construction.
1091      */
1092     constructor (string memory name_, string memory symbol_) public {
1093         _name = name_;
1094         _symbol = symbol_;
1095         _decimals = 18;
1096     }
```

Figure 1 constructor of ERC20 contract

```
1345 abstract contract Ownable is Context {
1346     address private _owner;
1347
1348     event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
1349
1350     /**
1351      * @dev Initializes the contract setting the deployer as the initial owner.
1352      */
1353     constructor () internal {
1354         address msgSender = _msgSender();
1355         _owner = msgSender;
1356         emit OwnershipTransferred(address(0), msgSender);
1357     }
```

Figure 2 constructor of Ownable contract

- Safety Recommendation: Remove the visibility declaration of the constructor.
- Fix Result: Ignore
- 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 BNB.
- 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. In the *transferAndCall* function of the BEP20 contract of this project, 'call' is used to call the *requestUnlock* function of the HomeBridge contract, and the corresponding function selector id is specified in the function, and the return value is checked. There is no security risk.

```
53  function transferAndCall(address to, uint256 amount, bytes memory) external virtual returns (bool) {
54    require(transfer(to, amount), "BEP20:transferAndCall error");
55
56    if (Misc.isContract(to)) {
57      (bool success,) = to.call(abi.encodeWithSelector(START_UNLOCK_FUNCTION, msg.sender, amount));
58
59      require(
60        success,
61        "BEP20:transferAndCall error remote returned false"
62      );
63    }
64
65    return true;
66  }
```

Figure 5 source code of transferAndCall function

- 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.

The transfer information of this project needs to be signed by the validators, and the user needs to use the signature data to unlock the corresponding token on Ethereum. In order to prevent replay attack, this contract function will record the data that has been signed and check it every time it is signed to avoid repeated signatures and repeated use of signed data.

- 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 BEP20 Contract

3.1.1 Basic token information

- Description: According to the project architecture, the BEP20 contract implements a basic BEP20 token, and its basic information is as follows:

Token name	(Enter when the contract is deployed)
Token symbol	(Enter when the contract is deployed)
Token decimals	18
Total supply	Initial supply is 0 (Mintable; Burnable)
Token type	BEP20

Table 1 Basic Token Information

3.1.2 BEP20 Token Functions

- Description: This contract token implements the basic functions of BEP20 standard tokens, and token holders can call corresponding functions for token transfer, approve and other operations.
- Related functions: *name, symbol, decimals, balanceOf, transfer, transferFrom, allowance, approve*
- Result: Pass

3.1.3 BEP20 Token Minting

- Description: The owner of contract can call the *mint* function to mint tokens to the specified address. At present, BEP20 tokens have no cap on minting, but according to the business logic of the project, this audit is to meet the business needs.

```
22  function mint(
23      address to,
24      uint256 amount
25  ) onlyOwner public virtual {
26      _mint(to, amount);
27  }
```

Figure 6 source code of mint function

- Related functions: *mint*
- Safety Suggestion: None
- Result: Pass

3.1.4 BEP20 Token Burning

- Description: The contract manager owner can call the *burn* function to destroy his own tokens. According to the system design requirements, the owner of this contract is the HomeBridge contract, so only the HomeBridge contract can destroy tokens sent by other users(by *transferAndCall* function).

```
33  function burn(
34      uint256 amount
35  ) onlyOwner public {
36      _burn(msg.sender, amount);
37  }
```

Figure 7 source code of burn function

- Related functions: *burn*
- Safety Suggestion: None
- Result: Pass

3.1.5 Special token transfer function

- Description: The user holding BEP20 token can call the *transferAndCall* function of this contract to transfer the token. If the corresponding target address is a contract, the *requestUnlock* function of the corresponding contract will be called.

```
53    function transferAndCall(address to, uint256 amount, bytes memory) external virtual returns (bool) {
54      require(transfer(to, amount), "BEP20:transferAndCall error");
55
56      if (Misc.isContract(to)) {
57        (bool success,) = to.call(abi.encodeWithSelector(START_UNLOCK_FUNCTION, msg.sender, amount));
58
59        require(
60          success,
61          "BEP20:transferAndCall error remote returned false"
62        );
63      }
64
65      return true;
66    }
```

Figure 8 source code of transferAndCall function

- Related functions: *transferAndCall*
- Safety Suggestion: None
- Result: Pass

3.1.6 Contract management

- Description: The main contract BEP20 inherits Ownable contract, the owner can call the *transferOwnership* function to transfer the contract management authority to the specified address; and the *renounceOwnership* function can be called to renounce the management authority.
- Related functions: *transferOwnership, renounceOwnership*
- Safety Suggestion: None
- Result: Pass

3.2 The ValidatorRegistry Contract

3.2.1 Contract initialization

- Description: After the contract is deployed, any user can call the *initialize* function of the contract to initialize the contract. This function needs incoming the initial validator addresses and quorum.
- Related functions: *initialize*
- Safety Suggestion: None
- Result: Pass

3.2.2 Contract management

- Description: The contract owner can call the *transferOwnership* function to transfer the contract management authority to the specified address; and the *renounceOwnership* function can be called to renounce the management authority.
- Related functions: *transferOwnership, renounceOwnership*
- Safety Suggestion: None

- Result: Pass

3.2.3 Manage validator

- Description: The contract manager(owner) can call the *addValidator* function to add the specified address to the validator list. Note: Addresses that are already validator can still be added. The owner can also call the *removeValidator* function to remove the specified address from the validator list.
- Related functions: *addValidator, removeValidator*
- Safety Suggestion: None
- Result: Pass

3.2.4 Update Quorum

- Description: The contract manager(owner) can call the *setQuorum* function to update the quorum. This function will check the new quorum value 'quorum_' and require it to be no less than the current number of validator.

```
80    function setQuorum(uint256 quorum_) public onlyOwner {
81      require(validatorCount >= quorum_, "ValidatorRegistry: not enough validators to support quorum");
82      require(quorum_ != 0, "ValidatorRegistry: quorum cannot be 0");
83
84      quorum = quorum_;
85
86      emit QuorumChanged(quorum_);
87    }
```

Figure 9 source code of setQuorum function

- Related functions: *setQuorum*
- Safety Suggestion: None
- Result: Pass

3.3 The ForeignBridge Contract

3.3.1 Contract initialization

- Description: After the contract is deployed, any user can call the *initialize* function of the contract to initialize the contract. As shown in the figure below, this function will call the *__BaseBridge_init* function to initialize the owner and block number; call the *__UnlockProcessor_init* function to initialize the token contract and validatorRegistry contract address.

```
16    function initialize(
17      IERC20 token,
18      ValidatorRegistry validatorRegistry
19    ) public initializer {
20      __BaseBridge_init();
21      __UnlockProcessor_init(token, validatorRegistry);
22    }
```

Figure 10 source code of initialize function

- Related functions: *initialize*, *__BaseBridge_init, __UnlockProcessor_init*

- Safety Suggestion: None
- Result: Pass

3.3.2 Contract management

- Description: The contract owner can call the *transferOwnership* function to transfer the contract management authority to the specified address; and the *renounceOwnership* function can be called to renounce the management authority.
- Related functions: *transferOwnership, renounceOwnership*
- Safety Suggestion: None
- Result: Pass

3.3.3 Unlock locked tokens

- Description: Any user can input signature data to unlock the locked token at the specified address(It will check whether the contract is paused). As shown in the figure below, the function will first verify the validity of the signature data; then parse the signature message and verify it; finally, after the all verification is passed, call the *executeRequest* function to execute the unlock request.

```
49  function unlock(bytes memory message, bytes memory signatures) whenNotPaused external {
50    ECDSA.verifyValidatorSigs(message, signatures, validatorRegistry);
51    (
52      address recipient,
53      uint256 amount,
54      bytes32 txHash,
55      uint256 logIndex,
56      address contractAddress
57    ) = ECDSA.parseMessage(message);
58    bytes32 messageId = keccak256(abi.encodePacked(message));
59
60    require(
61      contractAddress == address(this),
62      "UnlockProcessor:contract address mismatch"
63    );
64    require(
65      !isUnlockRequestProcessed(messageId),
66      "UnlockProcessor:unlock request processed"
67    );
68
69    setUnlockRequestProcessed(messageId);
70    require(executeRequest(recipient, amount), "UnlockProcessor:error unlocking tokens");
71
72    emit UnlockCompleted(recipient, amount, txHash, logIndex);
73  }
```

Figure 11 source code of unlock function

The *verifyValidatorSi*gs function is used to verify the validity of the signature, including:

(1) Signature data length;

(2) Whether the signer is the validator specified by the validatorRegistry contract;

(3) Whether the signer has duplicate signatures.

```
95    function verifyValidatorSigs(
96      bytes memory message,
97      bytes memory signatures,
98      ValidatorRegistry validatorRegistry
99    ) external view {
100     require(isMessageValid(message), "ECDSA:Invalid message");
101     uint256 quorum = validatorRegistry.quorum();
102     uint256 sigsCount;
103
104     assembly {
105       sigsCount := and(mload(add(signatures, 1)), 0xff)
106     }
107
108     require(sigsCount >= quorum, "ECDSA:quorum not reached");
109     address[] memory validators = new address[](quorum);
110
111     for (uint256 i = 0; i < quorum; i++) {
112       uint8 v;
113       bytes32 r;
114       bytes32 s;
115       uint256 posr = 33 + sigsCount + 32 * i;
116       uint256 poss = posr + 32 * sigsCount;
117
118       assembly {
119         v := mload(add(signatures, add(2, i)))
120         r := mload(add(signatures, posr))
121         s := mload(add(signatures, poss))
122       }
123
124       address validator = getSigner(message, v, r, s);
125       require(validatorRegistry.isValidator(validator), "ECDSA:address not validator");
126       require(!arrayContains(validators, validator), "ECDSA:sig already processed");
127       validators[i] = validator;
128     }
129   }
```

Figure 12 source code of verifyValidatorSigs function

The *executeRequest* function performs the transfer operation.

```
80    function executeRequest(
81      address recipient,
82      uint256 value
83    ) private returns (bool) {
84      return erc20.transfer(recipient, value);
85    }
```

Figure 13 source code of executeRequest function

- Related functions: *unlock, verifyValidatorSigs , executeRequest*
- Safety Suggestion: None
- Result: Pass

3.3.4 Withdraw other tokens owned by the contract

- Description: The contract manager owner can call the *withdrawTokens* function to withdraw other tokens owned by the contract (not the BEP20 token).

```
function withdrawTokens(IERC20 token, address recipient, uint256 amount) onlyOwner external {
  require(token != erc20, "ForeignBridge:Cannot manually release erc20 Tokens");
  token.transfer(recipient, amount);
}
```

Figure 14 source code of withdrawTokens function

- Related functions: *withdrawTokens*
- Safety Suggestion: None
- Result: Pass

3.4 The HomeBridge Contract

3.4.1 Contract initialization

- Description: After the contract is deployed, any user can call the *initialize* function of the contract to initialize the contract. As shown in the figure below, this function will call the *__BaseBridge_init* function to initialize the owner and block number; call the *__MintRequest_init* function to initialize the token and validatorRegistry contract address corresponding to the bridge.

```
function initialize(
  BEP20 token,
  ValidatorRegistry validatorRegistry_
) public initializer {
  __BaseBridge_init();
  __MintRequest_init(token, validatorRegistry_);
}
```

Figure 15 source code of initialize function

- Related functions: *initialize, __BaseBridge_init, __MintRequest_init,*
- Safety Suggestion: None
- Result: Pass

3.4.2 Contract management

- Description: The contract owner can call the *transferOwnership* function to transfer the contract management authority to the specified address; and the *renounceOwnership* function can be called to renounce the management authority.
- Related functions: *transferOwnership, renounceOwnership*
- Safety Suggestion: None
- Result: Pass

3.4.3 Process Minting Request

- Description:The validator call the *processRequest* function to sign the minting request of the corresponding address of the BSC chain(It will check whether the contract is paused). As shown in the figure below, the function will construct the corresponding message id based on the transaction

information, and then the caller will sign and record it; finally, count whether the number of signers has reached the specified quorum, and if so, call the *executeRequest* function to execute the request.

In extreme cases, user can use a contract to execute the same token transfer twice under the same transaction(Use contract function to finish), then the post-signature transfer information in the two transfers will fail when signing.

```
57  function processRequest(
58    address recipient,
59    uint256 value,
60    bytes32 txHash,
61    uint256 logIndex
62  ) external whenNotPaused onlyValidator {
63    bytes32 messageId = keccak256(abi.encodePacked(recipient, value, txHash, logIndex));
64
65    require(
66      !isMsgProcessed(messageId),
67      "MintRequest:request has been processed"
68    );
69
70    bytes32 requestId = keccak256(abi.encodePacked(msg.sender, messageId));
71
72    require(
73      !requestExists(requestId),
74      "MintRequest:request exists"
75    );
76
77    storeRequest(requestId);
78    // We don't need to protect against overflow since the num of validators cannot exceed the max int
79    uint256 numOfSigs = getMsgSignatures(messageId) + 1;
80    setMsgSignatures(messageId, numOfSigs);
81
82    emit RequestSigned(msg.sender, messageId);
83
84    if (numOfSigs >= validatorRegistry.quorum()) {
85      executeRequest(recipient, value);
86      setMsgProcessed(messageId);
87
88      emit RequestProcessed(recipient, value, txHash, logIndex);
89    }
90  }
```

Figure 16 source code of processRequest function

In the *executeRequest* function, this contract will be used as the minter of the BEP20 contract to call the *mint* function of the BEP20 contract to mint tokens for the specified address.

```
97   function executeRequest(
98     address recipient,
99     uint256 value
100  ) private {
101    bep20.mint(recipient, value);
102  }
```

Figure 17 source code of executeRequest function

- Related functions: *processRequest, executeRequest*

- Safety Suggestion: Add additional fields to the transfer information to distinguish between two identical token transfers for the same transaction.
- Fix Result: Ignore.
- Result: Pass

3.4.4 Request to unlock token

- Description: This function requires the caller to be the BEP20 contract(It will check whether the contract is paused). The actual usage is: the user calls the *transferAndCall* function of the BEP20 contract to send tokens to this contract, and calls the *requestUnlock* function of this contract to destroy this part of BEP20 tokens.

```
20  function requestUnlock(
21    address from,
22    uint256 value
23  ) external whenNotPaused returns (bool) {
24    require(
25      msg.sender == address(bep20),
26      "UnlockRequest:sender is not the BEP20"
27    );
28
29    bep20.burn(value);
30
31    emit RequestTokenUnlock(from, value);
32    return true;
33  }
```

Figure 18 source code of requestUnlock function

- Related functions: *requestUnlock*
- Safety Suggestion: None
- Result: Pass

3.4.5 Submit Unlock Signature

- Description: The validator can call *submitUnlockSignature* function to submit the signature for token unlocking at the specified address(It will check whether the contract is paused). As shown in the figure below, the function will check the caller and only submit its own signature; check that the corresponding information has not been executed to avoid replay attacks; check the signature and record it to avoid repeated submission of signatures; finally it will judge whether the number of signers has reached the quorum, if it reaches, modify the status of the corresponding message id.

```
66    function submitUnlockSignature(
67      bytes memory message,
68      bytes memory signature
69    ) external whenNotPaused onlyValidator {
70      require(ECDSA.isMessageValid(message), "UnlockRequest:invalid message");
71      require(msg.sender == ECDSA.recoverAddress(message, signature), "UnlockRequest:signature mismatch");
72
73      bytes32 messageId = keccak256(abi.encodePacked(message));
74
75      require(
76        !isMsgProcessed(messageId),
77        "UnlockRequest:request has been processed"
78      );
79
80      bytes32 requestId = keccak256(abi.encodePacked(msg.sender, messageId));
81
82      require(
83        !requestExists(requestId),
84        "UnlockRequest:request exists"
85      );
86
87      storeRequest(requestId);
88      // We don't need to protect against overflow since the num of validators cannot exceed the max int
89      uint256 numOfSigs = getMsgSignatures(messageId) + 1;
90
91      if(numOfSigs == 1) {
92        storeMessage(messageId, message);
93      }
94
95      setMsgSignatures(messageId, numOfSigs);
96
97      bytes32 sigIndexId = keccak256(abi.encodePacked(messageId, numOfSigs - 1));
98      storeSignature(sigIndexId, signature);
99
100     emit UnlockRequestSigned(msg.sender, messageId);
101
102     if (numOfSigs >= validatorRegistry.quorum()) {
103       setMsgProcessed(messageId);
104       emit UnlockSigsCollected(messageId, numOfSigs);
105     }
106   }
```

Figure 19 source code of submitUnlockSignature function

- Related functions: *submitUnlockSignature*
- Safety Suggestion: None
- Result: Pass

3.4.6 Withdraw tokens owned by the contract

- Description: The contract manager owner can call the *withdrawTokens* function to withdraw tokens owned by the contract.

```
33    function withdrawTokens(IERC20 token, address recipient, uint256 amount) onlyOwner external {
34      // we should allow witdrawal of BEP20 token that are accedentally sent to this contract.
35      // User should never directly send BEP20 to the bridge address; they should use transferAndCall
36      token.transfer(recipient, amount);
37    }
```

Figure 20 source code of withdrawTokens function

- Related functions: *withdrawTokens*
- Safety Suggestion: None
- Result: Pass

4. Conclusion

Beosin(ChengduLianAn) conducted a detailed audit on the design and code implementation of the smart contracts project Hot Cross BSC Bridge V1. All the issues found during the audit have been written into this audit report. The overall audit result of the smart contract project Hot Cross BSC Bridge V1 is **Pass**.

Official Website

https://lianantech.com

E-mail

vaas@lianantech.com

Twitter

https://twitter.com/Beosin_com