SMART CONTRACT AUDIT REPORT

for

RFQ (Celer IM)

Prepared By: Xiaomi Huang

PeckShield

October 19, 2022

Document Properties

Client	Celer Network
Title	Smart Contract Audit Report
Target	RFQ/Celer
Version	1.0-rc
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	October 19	Xuxian Jiang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction 4
1.1 About RFQ/Celer . . . 4
1.2 About PeckShield . . . 5
1.3 Methodology . . . 5
1.4 Disclaimer . . . 7

2 Findings 9
2.1 Summary . . . 9
2.2 Key Findings . . . 10

3 Detailed Results 11
3.1 Improved Validation in RFQ::_srcDeposit() . . . 11
3.2 Trust Issue Of Admin Keys . . . 12
3.3 Suggested Disallowance of Overpaid Native Coins . . . 14

4 Conclusion 15

References 16

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Celer IM`-based `RFQ` swap protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About RFQ/Celer

The `RFQ (request-for-quote)` contracts work together with the `Celer IM` platform to enable secure and efficient intra- or inter-chain token swaps. The inter-chain swap process takes three steps: 1) The user submits a quote request with a deposit to the `RFQ` contract on the source chain; 2) The market maker transfers tokens according to the quote to the user through the `RFQ` contract on the destination chain; and 3) The market maker receives the user's deposit by submitting proof of step 2 generated by `Celer IM` to the `RFQ` contract on the source chain. If the market maker fails to fulfill the quote, the user could get the fund back via a refund process. And the intra-chain swap is a simplified version of inter-chain swap with no involvement of `Celer IM`.

Table 1.1: Basic Information of RFQ/Celer

Item	Description
Target	RFQ/Celer
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 19, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Please note that this audit only covers the `contracts/message/apps/RFQ.sol` contract.

- https://github.com/celer-network/sgn-v2-contracts.git (73e7c46)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/celer-network/sgn-v2-contracts.git (15c9ce8)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Celer IM`-based `RFQ` swap protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	1	■
Informational	1	■
Total	3	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key RFQ/Celer Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Validation in RFQ::_srcDeposit()	Coding Practices	Fixed
PVE-002	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-003	Informational	Suggested Disallowance of Overpaid Native Coins	Coding Practices	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Improved Validation in RFQ::_srcDeposit()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RFQ`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

The `RFQ` swap protocol allows for convenient cross-chain token swaps. As mentioned earlier, the inter-chain swap process takes three steps: 1) The user submits a quote request with a deposit to the `RFQ` contract on the source chain; 2) The market maker transfers tokens according to the quote to the user through the `RFQ` contract on the destination chain; and 3) The market maker receives the user's deposit by submitting proof of step 2 generated by `Celer IM` to the `RFQ` contract on the source chain. While analyzing the first step, we notice the current implementation in depositing in the source chain can be strengthened.

To elaborate, we show below the implementation of the `_srcDeposit()` routine. This routine implements a rather straightforward logic in validating the input quote and sends a signal for the intended swap operation. The validation can be improved by also ensuring the given quote is not expired or at least no earlier than the current `_submissionDeadline`, i.e., `_quote.deadline>_submissionDeadline`.

```
    function _srcDeposit(
        Quote calldata _quote,
        uint64 _submissionDeadline,
        uint256 _msgFee
    ) private returns (bytes32) {
        require(_submissionDeadline > block.timestamp, "Rfq: submission deadline passed");
        require(
            _quote.receiver != address(0) && _quote.liquidityProvider != address(0),
            "Rfq: invalid receiver or liquidityProvider"
        );
```

```
        require(_quote.srcChainId == uint64(block.chainid), "Rfq: src chainId mismatch");
        require(_quote.sender == msg.sender, "Rfq: sender mismatch");
        bytes32 quoteHash = getQuoteHash(_quote);
        require(quotes[quoteHash] == QuoteStatus.Null, "Rfq: quote hash exists");
        uint256 rfqFee = getRfqFee(_quote.dstChainId, _quote.srcAmount);
        require(rfqFee <= _quote.srcAmount - _quote.srcReleaseAmount, "Rfq: insufficient protocol fee");

        quotes[quoteHash] = QuoteStatus.SrcDeposited;
        if (_quote.srcChainId != _quote.dstChainId) {
            address msgReceiver = remoteRfqContracts[_quote.dstChainId];
            require(msgReceiver != address(0), "Rfq: dst contract not set");
            bytes memory message = abi.encodePacked(quoteHash);
            sendMessage(msgReceiver, _quote.dstChainId, message, _msgFee);
        }
        emit SrcDeposited(quoteHash, _quote);
        return quoteHash;
    }
```

Listing 3.1: `RFQ::_srcDeposit()`

Recommendation Improve the above implementation so that the submitted quote is fresh and not expired yet.

Status The issue has been addressed by the following commit: `c66326d4`.

3.2 Trust Issue Of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `RFQ`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `RFQ` implementation, there is a privileged account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function setRemoteRfqContracts(uint64[] calldata _chainIds, address[] calldata _remoteRfqContracts)
        external
        onlyOwner
    {
        require(_chainIds.length == _remoteRfqContracts.length, "Rfq: length mismatch");
```

```
        for (uint256 i = 0; i < _chainIds.length; i++) {
            remoteRfqContracts[_chainIds[i]] = _remoteRfqContracts[i];
        }
        emit RfqContractsUpdated(_chainIds, _remoteRfqContracts);
    }

    function setFeePerc(uint64[] calldata _chainIds, uint32[] calldata _feePercs)
        external onlyGovernor {
        require(_chainIds.length == _feePercs.length, "Rfq: length mismatch");
        for (uint256 i = 0; i < _chainIds.length; i++) {
            require(_feePercs[i] < 1e6, "Rfq: fee percentage too large");
            if (_chainIds[i] == 0) {
                feePercGlobal = _feePercs[i];
            } else {
                feePercOverride[_chainIds[i]] = _feePercs[i];
            }
        }
        emit FeePercUpdated(_chainIds, _feePercs);
    }

    function setTreasuryAddr(address _treasuryAddr) external onlyOwner {
        treasuryAddr = _treasuryAddr;
        emit TreasuryAddrUpdated(_treasuryAddr);
    }
```

Listing 3.2: Example Privileged Operations in RFQ

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed by the team.

3.3 Suggested Disallowance of Overpaid Native Coins

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: RFQ
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

Besides the inter-chain swap support, the RFQ swap also supports the traditional intra-chain swap, which in essence is a simplified version of inter-chain swap with no involvement of Celer IM. In the process of analyzing the intra-chain swap logic, we notice the native coin swap can be improved.

In the following, we show below the related sameChainTransferNative() function. As the name indicates, this function is used to support native coin swap within the same chain. It comes to our attention that the native coin swap enforces the following requirement, i.e., `require(msg.value >= _quote.dstAmount, "Rfq: insufficient amount")` (line 164). While it indeed receives sufficient input tokens, there is a possibility of receiving overpaid tokens. And the overpaid tokens may be stuck in the current RFQ contract. To avoid that, we suggest to revise the above enforcement as follows: `require(msg.value == _quote.dstAmount, "Rfq: insufficient amount")`.

```
161     function sameChainTransferNative(Quote calldata _quote, bool _releaseNative)
            external payable whenNotPaused {
162         require(_quote.srcChainId == _quote.dstChainId, "Rfq: not same chain swap");
163         require(_quote.dstToken == nativeWrap, "Rfq: dst token mismatch");
164         require(msg.value >= _quote.dstAmount, "Rfq: insufficient amount");
165         (bytes32 quoteHash, ) = _dstTransferCheck(_quote);
166         _transferNativeToken(_quote.receiver, _quote.dstAmount);
167         _srcRelease(_quote, quoteHash, _releaseNative);
168         emit DstTransferred(quoteHash, _quote.receiver, _quote.dstToken, _quote.
                dstAmount);
169     }
```

Listing 3.3: RFQ::sameChainTransferNative()

Recommendation Reject the cases when the user may accidently send more native coins than necessary.

Status The issue has been addressed by the following commit: c66326d4.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `RFQ` swap, which works together with the `Celer IM` platform to enable secure and efficient intra- or inter-chain token swaps. Our analysis shows that the current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[3] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[4] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[5] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[6] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[7] PeckShield. PeckShield Inc. https://www.peckshield.com.