SMART CONTRACT AUDIT REPORT

for

RangoCometIntermediary

Prepared By: Xiaomi Huang

PeckShield

July 30, 2023

Document Properties

Client	Rango
Title	Smart Contract Audit Report
Target	RangoCometIntermediary
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 30, 2023	Xuxian Jiang	Final Release
1.0-rc	July 24, 2023	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

Given the opportunity to review the design document and related smart contract source code of the `RangoCometIntermediary` 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 RangoCometIntermediary

`Rango` implements a cross-chain decentralized exchange (`DEX`), which provides the cross-chain swap service with a one-transaction user experience. It also implements the multi-bridge aggregation, including `PolyNetwork`, `Synapse`, `cBridge`, `Axelar`, and etc. This audit covers `RangoCometIntermediary`, which acts as an intermediary to interact with `Comet` in a cross-chain manner. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of RangoCometIntermediary

Item	Description
Target	RangoCometIntermediary
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	July 30, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/rango-finance/rango-comet-intermediary.git (fa5877b)

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

- https://github.com/rango-finance/rango-comet-intermediary.git (27fd348)

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

Likelihood

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.

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

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

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 `RangoCometIntermediary` contract. 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	1	■
Medium	1	■
Low	1	■
Informational	0	
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 high-severity vulnerability, 1 medium-severity vulnerability, and 1 low-severity vulnerability.

Table 2.1: Key RangoCometIntermediary Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Revisited TransferHelper::safeTransferToken() Logic	Business Logic	Resolved
PVE-002	Low	Inconsistent Native Asset Support With Comet	Business Logic	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Revisited TransferHelper::safeTransferToken() Logic

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: TransferHelper
- Category: Business Logic [4]
- CWE subcategory: N/A

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the transfer() routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., USDT, as our example. We show the related code snippet below. Specifically, the transfer() routine does not have a return value defined and implemented. However, the IERC20 interface has defined the transfer() interface with a bool return value. As a result, the call to transfer() may expect a return value. With the lack of return value of USDT's transfer(), the call will be unfortunately reverted.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.1: USDT::transfer()

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

In current implementation, if we examine the `TransferHelper::safeTransferToken()` routine that is designed to transfer the requested token to the intended recipient. To accommodate the specific idiosyncrasy, there is a need to revise the transfer error status as `!(success && (data.length == 0 || abi.decode(data, bool)))`, instead of `!success && (data.length == 0 || abi.decode(data, (bool)))` (line 435).

```
425   function safeTransferToken(
426     address token,
427     address to,
428     uint256 value,
429     bool raiseError
430   ) internal returns (bool) {
431     // bytes4(keccak256(bytes('transfer(address,uint256)')));
432     (bool success, bytes memory data) = token.call(
433       abi.encodeWithSelector(0xa9059cbb, to, value)
434     );
435     bool hasError = !success && (data.length == 0  abi.decode(data, (bool)));
436     if (hasError && raiseError) {
437       revert TransferHelper__TransferFailed();
438     }
439     return !hasError;
440   }
```

Listing 3.2: TransferHelper::safeTransferToken()

Recommendation Revise the above logic to properly check the transfer status. The same issue is also applicable to the `safeApprove()` routine in the same contract. Note that the `safeTransferFrom()` routine implements the correct logic.

Status The issue has been fixed by this commit: `27fd348`.

3.2 Inconsistent Native Asset Support With Comet

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: RangoCometIntermediary
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

As mentioned earlier, `RangoCometIntermediary` acts as an intermediary to interact with `Comet` in a cross-chain manner. While examining its interaction with `Comet`, we notice the native asset support may be inconsistent.

To elaborate, we show below the related code snippet of the `withdrawAndCrosschainSupply()` routine. This routine allows users to withdraw funds from a `Comet` instance on one chain, then bridge and supply it to another `Comet` instance on another chain. It comes to our attention that it allows the `sourceToken` to be `ETHER_ADDRESS`, which is `address(0)`. However, the `address(0)` asset is not being supported in current `Comet` instances.

```
function withdrawAndCrosschainSupply(
  address sourceComet,
  address sourceToken,
  address sourceRango,
  uint256 amount,
  bytes calldata rangoData
) external nonReentrant {
  IComet comet = IComet(sourceComet);
  comet.withdrawFrom(msg.sender, address(this), sourceToken, amount);

  bool foundRango = false;
  for (uint256 i = 0; i < s_rango.length; i++) {
    if (s_rango[i] == sourceRango) {
      foundRango = true;
      break;
    }
  }
  if (!foundRango) {
    revert RangoCometIntermediary__InvalidRangoContract();
  }

  bool success = false;
  if (sourceToken == ETHER_ADDRESS) {
    // send native coin to rango
    (success, ) = payable(sourceRango).call{value: amount}(rangoData);
  }
  ...
```

```
}
```

Listing 3.3: `RangoCometIntermediary::withdrawAndCrosschainSupply()`

Recommendation Correct the native asset support and make it consistent with `Comet`.

Status The issue has been fixed by this commit: `27fd348`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `RangoCometIntermediary`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `RangoCometIntermediary` 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 addRangoContract(address rangoContractAddress) external onlyOwner {
  s_rango.push(rangoContractAddress);
  emit RangoAddressAdded(rangoContractAddress);
}

/**
 * @notice Owner must be able to remove a contract from rango list
 * @param rangoContractAddress the contract address which should be removed from rango
     contracts list
 */
function removeRangoContract(
  address rangoContractAddress
) external onlyOwner {
  uint256 index = s_rango.length + 1;
  for (uint256 i = 0; i < s_rango.length; i++) {
    if (s_rango[i] == rangoContractAddress) {
      index = i;
      break;
    }
  }
  if (index < s_rango.length) {
    s_rango[index] = s_rango[s_rango.length - 1];
    s_rango.pop();
    emit RangoAddressRemoved(rangoContractAddress);
```

```
    } else {
      revert("Rango contract address not found");
    }
  }

  /**
   * @notice In case anything bad happens and token gets stuck in intermediary contract,
        owner should be able to refund it to user
   * @param token the token which should be refunded, address(0) for native token
   * @param amount the amount of token that needs to be refunded
   * @param user the user that must be receiving the refund
   */
  function refund(
    address token,
    uint256 amount,
    address user
  ) external onlyOwner {
    TransferHelper.safeTransfer(token, user, amount, true);
  }
```

Listing 3.4: Example Privileged Operations in `RangoCometIntermediary`

We emphasize that the privilege assignment is indeed necessary and consistent with the protocol design. However, it will be worrisome if the privileged account is a plain EOA account. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Note that a compromised privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Suggest a multi-sig account plays the privileged `owner` account to mitigate this issue. Additionally, all changes to privileged operations may need to be mediated with necessary timelocks.

Status The issue has been mitigated with the use of a multisig account to manage the admin key.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `RangoCometIntermediary` contract, which acts as an intermediary to interact with `Comet` in a cross-chain manner. It enriches the `Rango` protocol to better implement a cross-chain decentralized exchange (`DEX`) and improve the cross-chain swap service with a one-transaction user experience. 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

[4] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.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.