Rango V2.1

Security Audit Report

August 15, 2025

Contents

1 Introduction

1.1 About Rango

`Rango` is a new layer on top of all Bridges and DEXs, working as a Bridge Aggregator and DEX Aggregator at the same time to enable seamless on-chain and cross-chain swaps, finding the most efficient, safe, cheap and fast route for swapping from any token on any blockchain to any other token to any blockchain.

1.2 Source Code

The following source code was reviewed during the audit:

- https://github.com/rango-exchange/rango-contracts-v2
- CommitID: ec28303

And this is the final version representing all fixes implemented for the issues identified in the audit:

- https://github.com/rango-exchange/rango-contracts-v2
- CommitID: 3b67856

1.3 Revision History

Version	Date	Description
v2.1.0	June 26, 2024	Initial Audit
v2.1.1	July 15, 2025	Generic Bridge and CCTPV2 Middleware

2 | Overall Assessment

This report has been compiled to identify issues and vulnerabilities within the `Rango V2.1` protocol. Throughout this audit, we identified a total of 7 issues spanning various severity levels. By employing auxiliary tool techniques to supplement our thorough manual code review, we have discovered the following findings.

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	-	-	-	-
High	-	-	-	-
Medium	4	-	-	4
Low	3	1	-	2
Informational	-	-	-	-
Undetermined	-	-	-	-

3 Vulnerability Summary

3.1 Overview

Click on an issue to jump to it, or scroll down to see them all.

~~M-1~~ Possible Revert in LibSwapper::collectFeesForSwap()

~~M-2~~ Improper Swap-out Amount in LibSwapper::onChainSwapsInternal()

~~M-3~~ Revisited Logic of LibSwapper::onChainSwapsPreBridge()

~~M-4~~ Lack of Input Validation in RangoCCTPV2Middleware::callRecieveMessage()

L-1 Potential Risks Associated with Centralization

~~L-2~~ Improper Event in LibInterchain::_handleUniswapV2()/_handleCurve()

~~L-3~~ Revisited Logic of LibSwapper2::returnExcessAmounts()

3.2 Security Level Reference

In web3 smart contract audits, vulnerabilities are typically classified into different severity levels based on the potential impact they can have on the security and functionality of the contract. Here are the definitions for critical-severity, high-severity, medium-severity, and low-severity vulnerabilities:

Severity	**Description**
C-X (Critical)	A severe security flaw with immediate and significant negative consequences. It poses high risks, such as unauthorized access, financial losses, or complete disruption of functionality. Requires immediate attention and remediation.
H-X (High)	Significant security issues that can lead to substantial risks. Although not as severe as critical vulnerabilities, they can still result in unauthorized access, manipulation of contract state, or financial losses. Prompt remediation is necessary.
M-X (Medium)	Moderately impactful security weaknesses that require attention and remediation. They may lead to limited unauthorized access, minor financial losses, or potential disruptions to functionality.
L-X (Low)	Minor security issues with limited impact. While they may not pose significant risks, it is still recommended to address them to maintain a robust and secure smart contract.
I-X (Informational)	Warnings and things to keep in mind when operating the protocol. No immediate action required.
U-X (Undetermined)	Identified security flaw requiring further investigation. Severity and impact need to be determined. Additional assessment and analysis are necessary.

3.3 Vulnerability Details

3.3.1 [M-1] Possible Revert in LibSwapper::callSwapsAndFees()

Target	Category	IMPACT	LIKELIHOOD	STATUS
LibSwapper.sol	Business Logic	Medium	Medium	Addressed

The `callSwapsAndFees()` function is designed to swap the user-specified `fromToken` into `toToken` on the source chain. It also handles charging a swap fee through the `collectFeesForSwap()` function (line 285). By design, this fee can be taken in either the `fromToken` or the `toToken`.

While examining its implementation, we observe that the fee is collected (line 285) before executing the token swap (line 296). If the fee is meant to be collected in the `toToken` (line 358), the transaction will be reverted. This is because the contract does not hold any `toToken` before the swap is performed.

LibSwapper::callSwapsAndFees()

```
281 function callSwapsAndFees(SwapRequest memory request, Call[] calldata calls)
        private returns (bytes[] memory) {
282     ...

284     // Get Fees
285     LibSwapper.collectFeesForSwap(request);

287     // Execute swap Calls
288     bytes[] memory returnData = new bytes[](calls.length);
289     address tmpSwapFromToken;
290     for (uint256 i = 0; i < calls.length; i++) {
291         tmpSwapFromToken = calls[i].swapFromToken;
292         bool isTokenNative = tmpSwapFromToken == ETH;
293         if (isTokenNative == false)
294             approveMax(tmpSwapFromToken, calls[i].spender, calls[i].amount);

296         (bool success, bytes memory ret) = isTokenNative
297         ? calls[i].target.call{value : calls[i].amount}(calls[i].callData)
298         : calls[i].target.call(calls[i].callData);

300         emit CallResult(calls[i].target, success, ret);
301         if (!success)
302             revert(_getRevertMsg(ret));
303         returnData[i] = ret;
304     }
305     ...
306 }
```

LibSwapper::collectFeesForSwap()

```
function collectFeesForSwap(SwapRequest memory request) internal {
    BaseSwapperStorage storage baseSwapperStorage = getBaseSwapperStorage();
    // Get Platform fee
    bool hasPlatformFee = request.platformFee > 0;
    bool hasDestExecutorFee = request.destinationExecutorFee > 0;
    bool hasAffiliateFee = request.affiliateFee > 0;
    address feeToken = request.feeFromInputToken ? request.fromToken : request.
        toToken;
    if (hasPlatformFee  hasDestExecutorFee) {
        require(baseSwapperStorage.feeContractAddress != ETH, "Fee contract
            address not set");
        _sendToken(feeToken, request.platformFee + request.
            destinationExecutorFee, baseSwapperStorage.feeContractAddress,
            feeToken == ETH, false);
    }

    // Get affiliate fee
    if (hasAffiliateFee) {
        require(request.affiliatorAddress != ETH, "Invalid affiliatorAddress");
        _sendToken(feeToken, request.affiliateFee, request.affiliatorAddress,
            feeToken == ETH, false);
    }

    ...
}
```

Remediation The `callSwapsAndFees()` function should be called after the swap operation has been completed.

3.3.2 [M-2] Improper Swap-out Amount in onChainSwapsInternal()

Target	Category	IMPACT	LIKELIHOOD	STATUS
LibSwapper.sol	Business Logic	Medium	Medium	Addressed

The `onChainSwapsInternal()` function is responsible for swapping the user-specified `fromToken` into `toToken` and returning the resulting amount of `toToken`. It performs the token swap and charges the swap fee by calling the `callSwapsAndFees()` function (line 247). Especially, the amount of `toToken` received (i.e., `secondaryBalance`) is calculated by checking the contract's `toToken` balance before and after the swap process (line 259).

While examining its implementation, it turns out that if the swap fee is taken in the `toToken`, it incorrectly subtracts the swap fee from the `secondaryBalance` (line 262). This is erroneous because `secondaryBalance` already accounts for the swap fee deduction, resulting in a double deduction.

LibSwapper::onChainSwapsInternal()

```
function onChainSwapsInternal(
    SwapRequest memory request,
    Call[] calldata calls,
    uint256 extraNativeFee
) internal returns (bytes[] memory, uint) {

    uint toBalanceBefore = getBalanceOf(request.toToken);
    uint fromBalanceBefore = getBalanceOf(request.fromToken);
    uint256[] memory initialBalancesList = getInitialBalancesList(calls);

    // transfer tokens from user for SwapRequest and Calls that require transfer from user.
    transferTokensFromUserForSwapRequest(request);
    transferTokensFromUserForCalls(calls);

    bytes[] memory result = callSwapsAndFees(request, calls);

    // check if any extra tokens were taken from contract and return excess tokens if any.
    returnExcessAmounts(request, calls, initialBalancesList);

    // get balance after returning excesses.
    uint fromBalanceAfter = getBalanceOf(request.fromToken);

    ...

    uint toBalanceAfter = getBalanceOf(request.toToken);

    uint secondaryBalance = toBalanceAfter - toBalanceBefore;
    require(secondaryBalance >= request.minimumAmountExpected, "Output is less than minimum expected");

    return (result, secondaryBalance - (request.feeFromInputToken ? 0 : sumFees(request)));
}
```

Remediation Adjust the calculation to prevent the double deduction and accurately reflect the correct amount of `toToken`.

3.3.3 [M-3] Revisited Logic of LibSwapper::onChainSwapsPreBridge()

Target	Category	IMPACT	LIKELIHOOD	STATUS
LibSwapper.sol	Business Logic	Medium	Medium	Addressed

The `onChainSwapsPreBridge()` function is designed to perform a token swap on the source chain before bridging. Within this function, the minimum amount of native token required is checked at lines 218-219. However, the calculation of `minimumRequiredValue` does not account for the scenario where `toToken` is used as the fee token. This oversight can lead to the minimum check triggering a revert in such cases.

LibSwapper::onChainSwapsPreBridge()

```
211 function onChainSwapsPreBridge(
212     SwapRequest memory request,
213     Call[] calldata calls,
214     uint extraFee
215 ) internal returns (uint out) {

217     bool isNative = request.fromToken == ETH;
218     uint minimumRequiredValue = (isNative ? request.platformFee + request.
            affiliateFee + request.amountIn + request.destinationExecutorFee : 0) +
            extraFee;
219     require(msg.value >= minimumRequiredValue, 'Send more ETH to cover input
            amount + fee');

221     (, out) = onChainSwapsInternal(request, calls, extraFee);
222     // when there is a bridge after swap, set the receiver in swap event to
            address(0)
223     emitSwapEvent(request, out, ETH);

225     return out;
226 }
```

Remediation When `request.feeFromInputToken==false`, the swap fee should not be added into `minimumRequiredValue`.

3.3.4 [M-4] Lack of Input Validation in callRecieveMessage()

Target	Category	IMPACT	LIKELIHOOD	STATUS
RangoCCTPV2Middleware.sol	Business Logic	High	Low	Addressed

The `callRecieveMessage()` function verifies the authenticity of cross-chain messages using `IMessage-TransmitterV2(s.messageTransmitterV2).receiveMessage(message, signature)` (line 64). However, it fails to validate the user-provided `_mintToken` parameter against the expected token address (e.g., burnToken) specified in the decoded CCTP message body. This allows an attacker to initiate a valid USDC cross-chain transfer, obtain a legitimate message and signature, and call `callRecieveMessage(message, signature, WBTC_ADDRESS)` with an arbitrary token address. The function then uses `_mintToken` directly in `LibInterchain.handleDestinationMessage()` (lines 72 - 77) without further checks, whereby the attacker receives WBTC instead of the intended USDC, potentially causing significant financial loss or asset mismatch.

RangoCCTPV2Middleware::callRecieveMessage()

```
56 function callRecieveMessage(bytes calldata message, bytes calldata signature,
       address _mintToken)
57     external
58     nonReentrant
59     onlyWhenNotPaused
60 {
61     CCTPV2Storage storage s = getCCTPV2Storage();
62     // Call the receiveMessage function of the message transmitter
63     if (!IMessageTransmitterV2(s.messageTransmitterV2).receiveMessage(message,
           signature)) {
64         revert RangoCCTPV2Middleware__MessageTransmissionFailed();
65     }

67     CCTPV2Message memory decodedMessage = decodeMessage(message);
68     CCTPV2MessageBody memory decodedMessageBody = this.decodeMessageBody(
           decodedMessage.messageBody);

70     Interchain.RangoInterChainMessage memory m =
71         abi.decode((decodedMessageBody.hookData), (Interchain.
               RangoInterChainMessage));
72     (address receivedToken, uint256 dstAmount, IRango2.CrossChainOperationStatus
           status) = LibInterchain
73         .handleDestinationMessage(
74         _mintToken,
75         decodedMessageBody.amount - decodedMessageBody.feeExecuted,
76         m
77     );

79     emit RangoBridgeCompleted(
80         m.requestId, receivedToken, m.originalSender, m.recipient, dstAmount,
               status, m.dAppTag
81     );
82 }
```

Remediation Ensure that the user-provided `_mintToken` parameter matches the `burnToken` field in the decoded CCTP message body and validate that the contract receives the correct token amount.

3.3.5 [L-1] Potential Risks Associated with Centralization

Target	Category	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Security	Low	Low	Acknowledged

In the `Rango V2.1` protocol, the existence of a privileged `owner` account introduces centralization risks, as it holds significant control and authority over critical operations governing the protocol. In the following, we show the representative function potentially affected by the privileges associated with the privileged account.

DiamondCutFacet::diamondCut()

```
contract DiamondCutFacet is IDiamondCut {
    /// @notice Add/replace/remove any number of functions and optionally execute
    ///         a function with delegatecall
    /// @param _diamondCut Contains the facet addresses and function selectors
    /// @param _init The address of the contract or facet to execute _calldata
    /// @param _calldata A function call, including function selector and
        arguments
    ///                  _calldata is executed with delegatecall on _init
    function diamondCut(
        FacetCut[] calldata _diamondCut,
        address _init,
        bytes calldata _calldata
    ) external override {
        LibDiamond.enforceIsContractOwner();
        LibDiamond.diamondCut(_diamondCut, _init, _calldata);
    }
}
```

Remediation To mitigate the identified issue, it is recommended to introduce multi-sig mechanism to undertake the role of the privileged account. Moreover, it is advisable to implement timelocks to govern all modifications to the privileged operations.

Response By Team This issue has been confirmed by the team.

3.3.6 [L-2] Improper Event in _handleUniswapV2()/_handleCurve()

Target	Category	IMPACT	LIKELIHOOD	STATUS
LibInterchain.sol	Coding Practices	Low	Low	Addressed

Events are crucial for blockchain transparency, reliability, and interoperability. They play a vital role in updating user interfaces, confirming transactions, and facilitating cross-contract communication in decentralized applications (DApps). Incorrect event statuses can mislead users and systems that depend on these events.

While examining the implementation of `_handleUniswapV2()` (which is used to interact with `UniswapV2` for token exchange), we observe that it emits an `ActionDone` event with the `success` flag set to true regardless of whether the token exchange operation succeeds or fails. This behavior is misleading and incorrect, as it fails to accurately reflect the actual outcome of the exchange operation. Similarly, the `_handleCurve()` function exhibits the same issue.

LibInterchain::_handleUniswapV2()

```
try
    IUniswapV2(action.dexAddress).swapExactTokensForTokens(
        _amount,
        action.amountOutMin,
        action.path,
        address(this),
        action.deadline
    )
returns (uint256[] memory) {
    emit ActionDone(Interchain.ActionType.UNI_V2, action.dexAddress, true, "");
    // Note: instead of using return amounts of swapExactTokensForTokens,
    //       we get the diff balance of before and after. This prevents errors
        for tokens with transfer fees
    uint toBalanceAfter = LibSwapper.getBalanceOf(toToken);
    SafeERC20.forceApprove(IERC20(action.path[0]), action.dexAddress, 0);
    return (true, toBalanceAfter - toBalanceBefore, toToken);
} catch {
    emit ActionDone(Interchain.ActionType.UNI_V2, action.dexAddress, true, "
        Uniswap-V2 call failed");
    SafeERC20.forceApprove(IERC20(action.path[0]), action.dexAddress, 0);
    return (false, _amount, shouldDeposit ? weth : _token);
}
```

Remediation Improve the `ActionDone` event in the `_handleUniswapV2()/_handleCurve()` functions.

3.3.7 [L-3] Revisited Logic of LibSwapper2::returnExcessAmounts()

Target	Category	IMPACT	LIKELIHOOD	STATUS
LibSwapper2.sol	Business Logic	Low	Medium	Addressed

The `returnExcessAmounts()` function calculates excess amounts by comparing current balances with initial balances for each swapToToken, but it fails to account for the `extraNativeFee` parameter that is specifically reserved for cross-chain bridge operations. When ETH is an intermediate token in the swap process, the function incorrectly returns all excess ETH (including the bridge fee portion) to users, leaving insufficient ETH for bridge transaction costs and causing cross-chain operations to fail.

LibSwapper2::onChainSwapsInternal()

```
function onChainSwapsInternal(
    SwapRequest memory request,
    Call[] calldata calls,
    uint256 extraNativeFee
) internal returns (bytes[] memory, uint) {
    uint toBalanceBefore = getBalanceOf(request.toToken);
    uint fromBalanceBefore = getBalanceOf(request.fromToken);
    uint256[] memory initialBalancesList = getInitialBalancesList(calls);

    // transfer tokens from user for SwapRequest and Calls that require transfer
        from user.
    transferTokensFromUserForSwapRequest(request);
    transferTokensFromUserForCalls(calls);

    bytes[] memory result = callSwapsAndFees(request, calls);

    returnExcessAmounts(request, calls, initialBalancesList);
    ...
}
```

LibSwapper2::returnExcessAmounts()

```
function returnExcessAmounts(
    SwapRequest memory request,
    Call[] calldata calls,
    uint256[] memory initialBalancesList) internal {
    uint excessAmountToToken;
    address tmpSwapToToken;
    uint currentBalanceTo;
    for (uint256 i = 0; i < calls.length; i++) {
        tmpSwapToToken = calls[i].swapToToken;
        currentBalanceTo = getBalanceOf(tmpSwapToToken);
        excessAmountToToken = currentBalanceTo - initialBalancesList[i];
        if (excessAmountToToken > 0 && tmpSwapToToken != request.toToken) {
            _sendToken(tmpSwapToToken, excessAmountToToken, msg.sender);
        }
    }
}
```

Remediation Add the `extraNativeFee` parameter to the `returnExcessAmounts()` function and improve the implementation to reserve the specified amount of ETH for bridge transaction fees when ETH is an intermediate token.

4 Appendix

4.1 About AstraSec

`AstraSec` is a blockchain security company that serves to provide high-quality auditing services for blockchain-based protocols. With a team of blockchain specialists, `AstraSec` maintains a strong commitment to excellence and client satisfaction. The audit team members have extensive audit experience for various famous DeFi projects. `AstraSec`'s comprehensive approach and deep blockchain understanding make it a trusted partner for the clients.

4.2 Disclaimer

The information provided in this audit report is for reference only and does not constitute any legal, financial, or investment advice. Any views, suggestions, or conclusions in the audit report are based on the limited information and conditions obtained during the audit process and may be subject to unknown risks and uncertainties. While we make every effort to ensure the accuracy and completeness of the audit report, we are not responsible for any errors or omissions in the report.

We recommend users to carefully consider the information in the audit report based on their own independent judgment and professional advice before making any decisions. We are not responsible for the consequences of the use of the audit report, including but not limited to any losses or damages resulting from reliance on the audit report.

This audit report is for reference only and should not be considered a substitute for legal documents or contracts.

4.3 Contact

Phone	+86 156 0639 2692
Email	contact@astrasec.ai
Twitter	https://twitter.com/AstraSecAI