ParaSwap PortikusV2

Security Audit Report

February 20, 2026

Contents

1 Introduction

1.1 About ParaSwap PortikusV2

`Portikus` is an intent-based protocol designed to facilitate gasless swaps through the execution of signed user intents by authorized agents. The protocol's architecture is centered around a registry of agents and modules and a factory for adapter creation. The key aspects of the protocol include Intent Execution, Permission Management and Modularity and Extensibility.

1.2 Audit Scope

Initial audit:

- Repository: https://github.com/paraswap/portikus-contracts/tree/feat/v2
- Review Commit: a82704f
- Final Commit: 5e25c76

Cross Chain Across Audit:

- Repository: https://github.com/paraswap/portikus-contracts/tree/feat/cross-chain-across
- Review Commit: 54a8456
- Final Commit: 9a3dea0

Buy Settlement Module Audit:

- Repository: https://github.com/VeloraDEX/portikus-contracts/tree/feat/buy-settlement-module
- Review Commit: 61fafca
- Final Commit: 8c839a9

Multiple Bridges Audit:

- Repository: https://github.com/VeloraDEX/portikus-contracts/commits/feat/multiple-bridges/
- Review Commit: 37ff9b9
- Final Commit: fec4e48

Celer Bridge Module Audit:

- Repository: https://github.com/VeloraDEX/portikus-contracts/commits/feat/celer-bridge-module/
- Review Commit: 0213a68
- Final Commit: cdf9145

External Settlement Module Audit:

- Repository: https://github.com/VeloraDEX/portikus-contracts/tree/feat/external-settlement-module
- Review Commit: ae137f1
- Final Commit: f184a30

External Order Type Audit:

- Repository: https://github.com/VeloraDEX/portikus-contracts/commits/feat/external-order-type
- Review Commit: 2a75cf4
- Final Commit: 4fc3538

Delta Tokens And Productive Orders Audit:

- Repository: https://github.com/VeloraDEX/portikus-contracts/tree/feat/delta-tokens-productive-orders
- Review Commit: 0f6e534
- Final Commit: d591cc6

Twap Settlement Module Audit:

- Repository: https://github.com/VeloraDEX/portikus-contracts/tree/feat/twap-settlement-module
- Review Commit: 6847d77
- Final Commit: ba151b1

TWAP Buy Settlement Audit:

- Repository: https://github.com/VeloraDEX/portikus-contracts/tree/feat/twap-buy
- Review Commit: 744e018
- Final Commit: 95edf5a

1.3 Changelog

Version	Date
Initial Audit	September 25, 2024
Cross Chain Across Audit	February 5, 2025
Buy Settlement Module Audit	July 12, 2025
Multiple Bridges Audit	August 28, 2025
Celer Bridge Module Audit	September 26, 2025
External Settlement Module Audit	October 13, 2025
External Order Type Audit	December 10, 2025
Delta Tokens And Productive Orders Audit	January 27, 2026
Twap Settlement Module Audit	January 27, 2026
TWAP Buy Settlement Audit	February 20, 2026

2 | Overall Assessment

This report has been compiled to identify issues and vulnerabilities within the `ParaSwap PortikusV2` protocol. Throughout this audit, we identified 15 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	1	-	-	1
Medium	8	-	-	8
Low	6	2	-	4
Informational	-	-	-	-
Total	15	2	-	13

3 Vulnerability Summary

3.1 Overview

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 [H-1] Bypass of Fees for ETH Orders in directSettleBatch()

Target	Category	IMPACT	LIKELIHOOD	STATUS
DirectSettlementModule.sol	Business Logic	High	Medium	Addressed

The `DirectSettlementModule` contract provides the `directSettleBatch()` function to facilitate the agent in settling a batch of orders in a single call. During our code review, we noticed that the agent can bypass the fees (protocol fees and partner fees) by settling multiple orders whose destination token is the native ETH.

In the following, we show the code snippet from the `DirectSettlementModule::_post()` function, which is used to process fees and pay for the order. Specifically, if the destination token is ETH, there is a check to ensure the amount of ETH received (`msg.value`) is greater than the required amount (line 154). Afterward, the function invokes the `processFees()` function to compute and collect the fees (line 159). It is important to note that the fees are recorded for the fee owners but are not reduced from the `msg.value`. Finally, the output asset is transferred to the order beneficiary (line 162).

However, if an agent tries to settle a batch of orders with ETH as the destination token, it is easy to pass the check `msg.value < amount` (line 154) for each order, because `msg.value` represents the total ETH amount used to settle all the orders. As a result, the agent can provide a crafted amount of ETH that only covers the order beneficiaries' payments, excluding the fees. This leaves a bad debt of ETH in the adapter, and the fee owners cannot be paid their fees.

Based on this, it is recommended to add a check in the `directSettleBatch()` function to ensure that the received `msg.value` is sufficient to cover the total required amount, including both the orders and the associated fees.

DirectSettlementModule::_post()

```
135 function _post(Order memory order, uint256 amount, bytes32 orderHash) internal {
136   // Init returnAmount, protocolFee and partnerFee
137   uint256 returnAmount;
138   uint256 protocolFee;
139   uint256 partnerFee;
140   // If beneficiary is not set, transfer to the owner
141   address beneficiary;
142   if (order.beneficiary == address(0)) {
143       beneficiary = order.owner;
144   } else {
145       beneficiary = order.beneficiary;
146   }
```

```
147     // Revert if the amount is less than the destAmount
148     if (amount < order.destAmount) {
149         revert InsufficientReturnAmount();
150     }
151     // Receive the output assets and process fees
152     if (order.destToken == ERC20UtilsLib.ETH_ADDRESS) {
153         // Check if the received ETH is less than the amount
154         if (msg.value < amount) {
155             revert InsufficientReturnAmount();
156         }
157         // Process fees
158         (returnAmount, partnerFee, protocolFee) =
159             order.partnerAndFee.processFees(ERC20UtilsLib.ETH_ADDRESS, amount,
                    order.expectedDestAmount);
160     } else {...}
161     // Transfer the output asset to the beneficiary
162     order.destToken.transferTo(beneficiary, returnAmount);
163     ...
164 }
```

Note that the same issue exists in the `FillableDirectSettlementModule` contract as well.

Remediation Add a proper check in the `directSettleBatch()` function to ensure that the received `msg.value` is sufficient to cover the total required amount.

3.3.2 [M-1] Failure to Pull Tokens from Agent via Permit2

Target	Category	IMPACT	LIKELIHOOD	STATUS
Multiple Files	Business Logic	High	Medium	Addressed

The `DirectSettlementModule` has been enhanced to support both buy and sell orders with a more secure reverse flow approach, where `_postSell()` handles sell orders, transfers destination tokens from agent to beneficiary and collects source tokens from owner. However, the contract uses `safeTransferFrom()` to transfer destination tokens from the agent, which doesn't properly handle cases where tokens are permitted or transferred via Permit2. The problem manifests in two scenarios:

- **Double Transfer Attempt**: When `_executeAgentPermit()` has already transferred destination tokens from the agent via `Permit2::permitTransferFrom()`, `_postSell()` attempts to transfer the same tokens again using `safeTransferFrom()`. This causes the agent to pay double the destination tokens to settle the order.
- **Settlement Failure**: When the agent's allowance is granted through Permit2, `_postSell()` cannot transfer the agent's destination tokens through `safeTransferFrom()`, causing the order settlement to fail.

Note that the same issue exists in the `_postSell()` and `_postBuy()` functions of all settlement modules.

DirectSettlementModule::_postSell()

```
264 function _postSell(Order calldata order, uint256 amount, bytes32 orderHash,
        bytes calldata bridgeData) internal {
265     // Ensure the amount meets the minimum required by the order
266     if (amount < order.destAmount) {
267         revert InsufficientReturnAmount();
268     }
269     ...
270     // 1. First transfer destination tokens to the beneficiary (reverse flow)
271     if (order.destToken != ERC20UtilsLib.ETH_ADDRESS) {
272         // For ERC20 tokens, transfer from agent to this contract
273         order.destToken.safeTransferFrom(msg.sender, address(this), amount);
274     }
275     ...
276 }
```

Remediation Replace `safeTransferFrom()` with `ERC20UtilsLib::transferFrom()`, which properly handles both Permit2 allowances and traditional ERC20 allowances based on the permit data length.

3.3.3 [M-2] Incorrect Permit Amount for Fillable Direct Settlement

Target	Category	IMPACT	LIKELIHOOD	STATUS
FillableDirectSettlementModule	Business Logic	High	High	Addressed

In the `FillableDirectSettlementModule` contract, the `executeAgentPermit()` function is responsible for executing the permit for the Agent, allowing the contract to transfer tokens on the agent's behalf. The amount to be permitted should be carefully determined based on the order type: for buy orders, the permit amount should reflect the actual fillable destination amount; for sell orders, it should be the amount provided by the agent.

However, in the current implementation, the function uses the whole order's `destAmount` as the permit amount for buy orders, rather than the actual fillable destination amount (i.e., `orderData.fillAmountOut`). This can result in the contract requesting a permit for a larger amount than is actually needed to fulfill the order. As a result, if the permit is given via Permit2, the agent may unintentionally grant a higher allowance than necessary, or the contract may transfer more destination tokens from the agent than required for the specific order fill.

FillableDirectSettlementModule::_executeAgentPermit()

```
184 function _executeAgentPermit(
185     Order calldata order,
186     bool isBuy,
187     uint256 amount,
188     bytes calldata agentPermit
189 ) internal
190 {
191     // Skip if no permit data is provided
192     if (agentPermit.length == 0) return;

194     // For non-ETH destination tokens, execute permit
195     if (order.destToken != ERC20UtilsLib.ETH_ADDRESS) {
196         uint256 permitAmount = isBuy ? order.destAmount : amount;
197         order.destToken.permit(
198             agentPermit,
199             msg.sender, // agent is the msg.sender
200             order.deadline,
201             permitAmount,
202             address(this) // recipient is this contract
203         );
204     }
205 }
```

Remediation Update the `executeAgentPermit()` function to use `orderData.fillAmountOut` as the permit amount for buy orders.

3.3.4 [M-3] Revised Logic to Install Module in install()

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

The `ModuleManagerLib::install()` function is responsible for installing new modules into the adapter. This process involves fetching the function selectors of the module and updating internal mappings to link the module's functions with its storage.

The code snippet below highlights the logic of how a module is added to the modules array and how the function selectors are associated with the module in the `moduleToSelectors` mapping. Afterward, the reverse association from the function selectors to the module is set up in the `selectorToModule` mapping.

However, there is a lack of recording the module's position (`ms.moduleToSelectors[module].moduleAddressPosition`), which is necessary to maintain the modules list. Out analysis shows that the module's position should be set to `ms.modules.length -1`.

Additionally, there is a potential flaw in how the function selectors's positions are calculated and tracked. Specifically, the position of each function selector (`functionSelectorPosition`) is calculated starting from the module's selectors length (`ms.moduleToSelectors[module].selectors.length`), rather than from 0 (line 101). A recommended approach is to start the function selectors's positions from 0.

Example Privileged Operations in `ExecutorManager`

```
90  function install(address module) external {
91    // Get adapter module storage
92    ModuleStorage storage ms = modulesStorage();
93    // Get module function selectors
94    bytes4[] memory selectors = IModule(module).selectors();
95    // Add module to modules
96    ms.modules.push(module);
97    // Set selectors in moduleToSelectors
98    ms.moduleToSelectors[module].selectors = selectors;

100   // Get selector position
101   uint32 selectorPosition = uint32(ms.moduleToSelectors[module].selectors.length
          );
102   // Set module in selectorToModule
103   for (uint256 i = 0; i < selectors.length; i++) {
104       address oldModule = ms.selectorToModule[selectors[i]].moduleAddress;
105       // If a selector is already set, revert as it would cause a conflict
106       if (oldModule != address(0)) {
107           // If a selector is already set the owner should uninstall the old
                  module first
108           revert SelectorAlreadySet(selectors[i], oldModule);
```

```
109         }
110         ms.selectorToModule[selectors[i]].functionSelectorPosition =
                selectorPosition;
111         ms.selectorToModule[selectors[i]].moduleAddress = module;
112         // Increase selectorPosition
113         selectorPosition++;
114     }
115 }
```

Remediation Revisit the `install()` function to properly update the the module's position and the function selectors's positions.

3.3.5 [M-4] Revised Bridging of ETH in _bridgeWithAcross()

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

The `_bridgeWithAcross()` function in the BridgeLib library is responsible for bridging tokens to the destination chain using the Across protocol. The function supports both ETH and ERC-20 tokens. However, it treats all input tokens as ERC-20 incorrectly and fails to handle ETH.

Specifically, for ERC-20 tokens, the function approves the Across to pull tokens during the deposit process. However, ETH does not require approval, and attempting to approve ETH will result in a transaction revert. Additionally, if the token to bridge is ETH, the Across protocol requires the input token to be specified as WETH (wrapped ETH), which is not properly handled in current implementation.

To properly support ETH bridging, we recommend removing the approval step for ETH, as it is unnecessary and causes transaction failures. Additionally, use WETH as the input token to comply with Across protocol requirements.

BridgeLib::_bridgeWithAcross()

```
30 function _bridgeWithAcross(
31     address pool,
32     Order memory order,
33     address beneficiary,
34     uint256 inputAmount,
35     bytes calldata data
36 )
37     internal
38 {
39     BridgeData memory bridgeData = abi.decode(data, (BridgeData));
40     uint256 bridgeFee =
```

```
41          bridgeData.relayerFee > order.bridge.maxRelayerFee ? order.bridge.
                maxRelayerFee : bridgeData.relayerFee;
42      uint256 outputAmount = inputAmount - bridgeFee;

44      // As pool address is a proxy, give approval only for the swap amount
45      order.destToken.safeApproveWithRetry(pool, inputAmount);

47      AcrossPoolInterface(pool).depositV3(
48        order.owner,
49        beneficiary,
50        order.destToken,
51        order.bridge.outputToken,
52        inputAmount,
53        outputAmount,
54        ...
55      );
56  }
```

Remediation Revisit the `_bridgeWithAcross()` function to remove the approval step for ETH and use WETH as the input token for ETH to align with Across protocol requirements.

3.3.6 [M-5] Incomplete Order Information in FillableOrderHashLib::hash()

Target	Category	IMPACT	LIKELIHOOD	STATUS
FillableOrderHashLib.sol	Data Integrity	Medium	Medium	🔗Addressed

The `Portikus` protocol allows authorized agents to settle user orders based on user signatures. However, in the `FillableOrderHashLib::hash()` function, the order hash calculation does not include the newly added bridge information. As a result, an agent could potentially modify the bridge details of a signed order, altering the user's original intent.

To mitigate this risk, we recommend including the bridge information in the fillable order hash to ensure the integrity of the order and prevent potential manipulation.

FillableOrderHashLib::hash()

```
33  function hash(Order memory order) internal pure returns (bytes32) {
34    return keccak256(
35        abi.encode(
36            _FILLABLE_ORDER_TYPEHASH,
37            order.owner,
38            order.beneficiary,
39            order.srcToken,
40            order.destToken,
41            order.srcAmount,
```

```
            order.destAmount,
            order.expectedDestAmount,
            order.deadline,
            order.nonce,
            order.partnerAndFee,
            keccak256(order.permit)
        )
    );
}
```

Remediation Revisit the `FillableOrderHashLib::hash()` function to include the bridge information in the fillable order hash.

3.3.7 [M-6] Decimals Scaling Error in _convertToDestDecimals()

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

The `Portikus` protocol implements a multiple-bridges feature to facilitate token bridging across chains, utilizing the `_convertToDestDecimals()` function to scale token amounts based on decimal differences between source and destination chains. The function uses a scalingFactor to adjust the amount, where a positive scalingFactor indicates the destination chain has more decimals, and a negative scalingFactor indicates the destination chain has fewer decimals.

However, the current logic incorrectly implements the scaling direction: it divides the amount by the scale when `scalingFactor > 0` (intended to convert to a chain with more decimals) and multiplies the amount by the scale when `scalingFactor < 0` (intended to convert to a chain with fewer decimals). The correct logic should be the opposite: multiply the source amount by the absolute value of scalingFactor when `scalingFactor > 0` to scale up to more decimals, and divide the source amount by the absolute value of scalingFactor when `scalingFactor < 0` to scale down to fewer decimals. This inversion leads to incorrect token amounts during cross-chain transfers, potentially causing financial discrepancies or losses.

BridgeModuleBase::_convertToDestDecimals()

```
    /// @notice Converts amount from source decimals to destination decimals
    /// @param amount Amount in source decimals
    /// @param scalingFactor Signed scaling factor (destDecimals - srcDecimals)
    /// @return Amount converted to destination decimals
    function _convertToDestDecimals(uint256 amount, int8 scalingFactor) internal
        pure returns (uint256) {
        if (scalingFactor == 0) return amount;
```

```
80          // Validate scaling factor to prevent overflow
81          int8 absFactor = scalingFactor >= 0 ? scalingFactor : -scalingFactor;
82          require(uint8(absFactor) <= 18, "Bridge: scaling factor too large");

84          uint256 scale = 10 ** uint8(absFactor);

86          if (scalingFactor > 0) {
87              // destDecimals > srcDecimals: divide to convert src -> dest
88              return amount / scale;
89          } else {
90              // srcDecimals > destDecimals: multiply to convert src -> dest
91              return amount * scale;
92          }
93      }
```

Remediation Correct the scaling logic in the `_convertToDestDecimals()` function to multiply the amount by the scale when `scalingFactor > 0` and divide the amount by the scale when `scalingFactor < 0`.

3.3.8 [M-7] Unauthorized Refund Access via Transfer ID Collision

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

In the `CelerBridgeModule` contract, the `_calculateTransferId()` function generates a transferId to validate refund requests for the `celerRefund()` function. The transferId is computed using the actualBridgeAmount (line 469) from the provided request protobuf instead of the original bridge amount specified in the input order. This design flaw allows two distinct orders with different bridge amounts to generate identical transferId if other parameters (e.g., beneficiary, destToken, destinationChainId, nonce) are identical. A malicious actor can exploit this by crafting a fraudulent order with a smaller bridge amount to produce a matching transferId, enabling them to claim the refund for a failed order owned by another user. This bypasses the intended ownership validation, as the orderOwners mapping only verifies the order's owner but does not ensure transferId uniqueness.

_calculateTransferId()

```
451     function _calculateTransferId(
452         Order memory order,
453         CelerBridgeData memory protocolData,
454         uint256 actualBridgeAmount
455     )
```

```
456         internal
457         view
458         returns (bytes32 transferId)
459     {
460         address beneficiary = order.beneficiary == address(0) ? order.owner :
                order.beneficiary;

462         if (protocolData.bridgeType == CelerBridgeType.Liquidity) {
463             // Liquidity bridge transferId calculation
464             transferId = keccak256(
465                 abi.encodePacked(
466                     address(this), // msg.sender (CelerBridgeModule)
467                     beneficiary, // receiver
468                     order.destToken, // token
469                     actualBridgeAmount, // ACTUAL amount that was bridged
470                     uint64(order.bridge.destinationChainId), // dstChainId
471                     protocolData.nonce, // nonce
472                     uint64(block.chainid) // current chain ID
473                 )
474             );
475         }
476         ...
477     }
```

Remediation Update `_bridgeInternal()` to store a mapping of each orderHash to its corresponding transferId. In `celerRefund()`, validate the provided transferId against the stored value to ensure only the legitimate order can trigger the refund.

3.3.9 [M-8] Lack of Received Amount Check in _executeBuyOrderFlashLoan()

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

The `externalSettleBuyWithFlashLoan()` function is designed to settle BUY orders using Aave flashloans for capital efficiency, allowing agents to perform external protocol operations without upfront capital. However, the function does not properly verify whether the expected amount of source tokens (`srcToken`) is received from the external protocol execution.

Without this validation, an under-returned amount from the external protocol could cause the contract to repay the Aave flashloan using existing fees or internal funds. This results in potential loss of funds from the contract and inconsistent accounting of asset flows.

_executeBuyOrderFlashLoan()

```
415     params.order.destToken.safeApprove(params.order.beneficiary, params.order.
            destAmount);

417     // Step 3: Execute external protocol contract
418     IExternalProtocolHandler(params.order.beneficiary).execute(
419         params.order.owner,
420         params.order.destToken, // tokenIn (e.g. old debt underlying)
421         params.order.destAmount, // amountIn (e.g. amount of old debt)
422         params.order.srcToken, // tokenOut (e.g. new debt underlying)
423         params.order.srcAmount - unusedSrcAmount, // amountOut (e.g. amount of new
                debt underlying to return)
424         params.order.data
425     );

427     // Step 4: Check sufficient srcToken for repayment and approve
428     if (params.order.srcToken.balanceOf(address(this)) < totalRepaymentAmount) {
429         revert InsufficientReturnAmount();
430     }

432     // Step 5: Approve Pool to pull repayment
433     params.order.srcToken.safeApprove(address(POOL), totalRepaymentAmount);
```

Remediation Before proceeding with flashloan repayment, the contract should explicitly calculate the amount of srcToken received from the external protocol execution and ensure it is no less than the expected debt amount.

3.3.10 [L-1] Potential Risks Associated with Centralization

Target	Category	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Security	Medium	Low	Acknowledged

In the `PortikusV2` protocol, the presence of a privileged `owner` accountintroduces risks of centralization, as it holds significant control and authority over critical operations governing the protocol. In the following, we highlight the representative functions that are potentially affected by the privileges associated with this privileged account.

Examples of Privileged Operations

```
111 function install(address module) external {
112   // Get adapter module storage
113   ModuleStorage storage ms = modulesStorage();
114   // Get module function selectors
115   bytes4[] memory selectors = IModule(module).selectors();
```

```
116     // Add module to modules
117     ms.modules.push(module);
118     // Set selectors in moduleToSelectors
119     ms.moduleToSelectors[module].selectors = selectors;

121     // Get selector position
122     uint32 selectorPosition = uint32(ms.moduleToSelectors[module].selectors.length
            );
123     // Set module in selectorToModule
124     for (uint256 i = 0; i < selectors.length; i++) {
125         address oldModule = ms.selectorToModule[selectors[i]].moduleAddress;
126         // If a selector is already set, revert as it would cause a conflict
127         if (oldModule != address(0)) {
128             // If a selector is already set the owner should uninstall the old
                    module first
129             revert SelectorAlreadySet(selectors[i], oldModule);
130         }
131         ms.selectorToModule[selectors[i]].functionSelectorPosition =
                selectorPosition;
132         ms.selectorToModule[selectors[i]].moduleAddress = module;
133         // Increase selectorPosition
134         selectorPosition++;
135     }
136 }

138 function setProtocolFeeClaimer(address protocolFeeClaimer) external onlyOwner {
139     protocolFeeClaimer.setFeeClaimer();
140 }

142 /// @inheritdoc IRegistry
143 function registerAgent(address[] calldata _agents) external onlyOwner {
144     // Loop through the agents and register them
145     for (uint256 i = 0; i < _agents.length; i++) {
146         address agent = _agents[i];
147         if (!isAgentRegistered[agent]) {
148             agents.push(agent);
149             isAgentRegistered[agent] = true;
150             emit AgentRegistered(agent);
151         }
152     }
153 }

155 /// @inheritdoc IRegistry
156 function registerModule(address[] calldata _modules) external onlyOwner {
157     // Loop through the modules and register them
158     for (uint256 i = 0; i < _modules.length; i++) {
159         address module = _modules[i];
160         if (!isModuleRegistered[module]) {
161             modules.push(module);
162             isModuleRegistered[module] = true;
```

```
163             emit ModuleRegistered(module);
164         }
165     }
166 }
```

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

3.3.11 [L-2] Improved Validation of Module in install()

Target	Category	IMPACT	LIKELIHOOD	STATUS
ModuleManagerLib.sol	Coding Practices	Low	Low	Acknowledged

The `install()` function in the `ModuleManagerLib` library is responsible for installing a new module by adding the module's function selectors and updating the mappings to link the module's address with the selectors. However, the current implementation lacks a validation check to ensure that the selectors array is not empty before proceeding with the installation. This omission can lead to issues in the `uninstall()` function.

As the code snippet shows, the `uninstall()` function retrieves the `selectors.length` and checks whether the module has been installed by ensuring that `selectors.length` is greater than zero (line 116). If the module's selectors length is zero, the `uninstall()` function fails to properly remove the module because the selectors array is empty. This can leave the protocol in an inconsistent state.

Based on this, it is recommended to add a validation check in the `install()` function to ensure that the selectors array is not empty.

ModuleManagerLib.sol

```
91  function install(address module) external {
92    // Get adapter module storage
93    ModuleStorage storage ms = modulesStorage();
94    // Get module function selectors
95    bytes4[] memory selectors = IModule(module).selectors();
96    // Add module to modules
97    ms.modules.push(module);
98    // Set selectors in moduleToSelectors
99    ms.moduleToSelectors[module].selectors = selectors;
100   ...
101 }

103 /*////////////////////////////////////////////////////////////////
```

```
104                             UNINSTALL
105 //////////////////////////////////////////////////////////////*/

107 /// @notice Remove a module from the adapter, removing all of its function
        selectors
108 /// @param module The address of the module to uninstall
109 function uninstall(address module) external {
110   // Get adapter module storage
111   ModuleStorage storage ms = modulesStorage();
112   // Get module function selectors
113   bytes4[] memory selectors = ms.moduleToSelectors[module].selectors;

115   // Check if the module is actually installed
116   if (selectors.length == 0) {
117       revert ModuleNotInstalled(module);
118   }
119   ...
120 }
```

Remediation Add a validation check in the `install()` function to ensure that the module's selectors array is not empty.

Response By Team The team will make sure to only register valid modules.

3.3.12 [L-3] Celer Refund Failure Due to protocolData Inconsistency

Target	Category	IMPACT	LIKELIHOOD	STATUS
CelerBridgeModule.sol	Business Logic	Low	Low	Addressed

The `celerRefund()` function in `CelerBridgeModule` decodes `protocolData` from the original user order to determine the bridge type for refund execution. However, when a bridge operation is executed with a `bridgeOverride` via cosignature, the actual bridging uses different `protocolData` from the override, which can include a different `bridgeType`. During bridge execution, `BridgeLib.doBridge()` resolves the override and uses `bridgeOverride.protocolData` if a cosignature is present. This resolved `protocolData` is passed to the bridge module. However, during refund, `celerRefund()` reads `protocolData` directly from the original order, ignoring any override used during bridging. The `_executeRefund()` function then uses this `protocolData.bridgeType` to determine which Celer contract to call. If the original order specified `PegDepositV` but the bridge was executed with `PegDepositV2` from an override, the refund will attempt to call the wrong contract, causing the refund to fail. This prevents users from successfully refunding their bridged tokens when a bridge override was used, potentially leading to permanent fund locking.

ModuleManagerLib.sol

```
function celerRefund(
    OrderWithSig memory orderWithSig,
    bytes calldata _request,
    bytes[] calldata _sigs,
    address[] calldata _signers,
    uint256[] calldata _powers
)
    external
{
    bytes32 orderHash = _verifySignature(orderWithSig);

    CelerBridgeData memory protocolData = _validateAndDecodeProtocolData(
        orderWithSig.order);

    _validateRefundRequest(orderHash, _request);

    // Execute refund
    (uint256 refundAmount, bytes32 refundId) =
        _executeRefund(orderWithSig.order, protocolData, _request, _sigs,
            _signers, _powers);
  ...
}
```

Remediation Store resolved `protocolData` in storage under orderHash. On refund, read and reuse this stored value instead of re-decoding from the original order — ensuring bridge and refund always use identical protocol parameters. Additionally, allow users to directly specify the `bridgeType` when initiating a refund.

3.3.13 [L-4] Inflation of receivedAmount by Yield Fee in productiveSettle()

Target	Category	IMPACT	LIKELIHOOD	STATUS
ProductiveSettlementModule	Business Logic	Low	Low	Addressed

When a ProductiveOrder swaps `ETH` to `WETH`, the yield fee transferred as `WETH` to the adapter during `redeemFromStrategy()` (executed in `_pre()`) is incorrectly included in the `receivedAmount` calculation in `_post()`, inflating the swap output. The issue occurs because `balanceBefore` is captured before the yield fee transfer, and the subsequent `receivedAmount` calculation `(order.destToken.getBalance() - balanceBefore)` includes the `WETH` yield fee that was already deducted from the user's assets and transferred to the adapter.

ProductiveSettlementModule::productiveSettle()

```
function productiveSettle(
    ProductiveOrderWithSig calldata orderWithSig,
    bytes calldata executorData,
    address executor,
    bytes calldata bridgeData
)
    external
    payable
    nonReentrant
    onlyAuthorizedAgent
{
    // 1. Check balance of destToken before settlement
    uint256 balanceBefore = orderWithSig.order.destToken.getBalance();
    // 2. Verify the order and transfer the input assets to the executor
        contract
    (bytes32 orderHash, uint256 actualDestAmount, uint256 actualExpectedAmount)
        = _pre(orderWithSig, executor);
    // 3. Execute the order using the provided data on the executor contract
    _execute(executorData, executor);
    // 4. Transfer the output assets to the beneficiary (with bridge override
        support)
    _post(orderWithSig, orderHash, actualDestAmount, actualExpectedAmount,
        balanceBefore, bridgeData);
}
```

The inflated `receivedAmount` causes `processSellFees()` to overcharge protocol and partner fees (calculated from the inflated base amount) and results in users receiving an inflated `returnAmount` that includes a portion of the yield fee that should belong to the protocol.

Remediation Exclude the yield fee from `receivedAmount` by capturing `balanceBefore` after the yield fee transfer but before the swap execution.

3.3.14 [L-5] Possible Cancel of Already-Filled Orders

Target	Category	IMPACT	LIKELIHOOD	STATUS
DeltaTokenModule.sol	Business Logic	Low	Low	Addressed

The `cancelAndWithdrawWithSignature()` function does not check whether the order nonce has already been used before calling `setNonceUsed()`. Since `setNonceUsed()` uses the `|=` operator, it will not revert even if the nonce was already consumed during order settlement. Combined with the fact that this function can be called by anyone with a valid cancellation signature, an attacker can repeatedly call this function with the same signature to repeatedly withdraw assets from the user's DeltaToken balance via `_withdrawNative()`. This can completely deplete the user's DeltaToken

balance, preventing legitimate orders from being settled due to insufficient balance.

The same issue also exists in other functions, like `cancelAndWithdraw()`, `cancelProductiveAndWithdraw()`, and `cancelProductiveAndWithdrawWithSignature()`, etc.

cancelAndWithdrawWithSignature()

```
283 function cancelAndWithdrawWithSignature(...)
284     external
285 {
286     Order calldata order = orderWithSig.order;
287     // Validate and verify order, get order hash
288     bytes32 orderHash = _validateAndVerifyOrder(orderWithSig, isFillable);
289     // Build EIP-712 hash for order cancellation authorization
290     bytes32 structHash = keccak256(abi.encode(ORDER_CANCELLATION_TYPEHASH,
            orderHash, isFillable));
291     bytes32 cancellationHash = _hashTypedDataV4(structHash);
292     // Verify the owner signed the cancellation authorization
293     cancellationSignature.verify(cancellationHash, order.owner);
294     // Order is cancelled by invalidating the nonce
295     NonceManagerLib.setNonceUsed(order.owner, order.nonce);
296     // Unwrap tokens if owner has sufficient balance
297     uint256 withdrawAmount = _withdrawNative(order, orderHash, isFillable);
298     emit OrderCancelled(order.owner, orderHash, withdrawAmount);
299 }
```

Remediation Add a nonce usage check before invalidating the nonce in all cancellation functions. Verify that the nonce has not been used by calling `isNonceUsed()` and revert if it has already been consumed. Alternatively, use `useNonce()` instead of `setNonceUsed()` or `invalidateNonce()`, as `useNonce()` will revert if the nonce is already used, preventing repeated cancellation and asset withdrawal.

3.3.15 [L-6] Enhanced Slice maxSrcAmount Validation in _validateSlice()

Target	Category	IMPACT	LIKELIHOOD	STATUS
TWAPBuySettlementModule	Business Logic	Low	Low	Addressed

The `_validateSlice()` function validates slice execution parameters before processing a TWAP Buy order slice. It checks execution timing (order deadline, slice deadline, interval), slice index sequencing, destination token amounts, and source token budget constraints. At line 402, the function validates that a slice's maxSrcAmount does not exceed the remaining source budget by checking `if (slice.maxSrcAmount > remainingSrcBudget)`. However, this validation permits a slice to consume the entire remaining budget. If that slice spends its full maxSrcAmount, subsequent slices have no budget and cannot execute, leaving the order partially completed.

_validateSlice()

```
397     if (slice.destAmount != expectedDestAmount) revert InvalidDestAmount();
398     if (slice.destAmount == 0) revert InvalidField("destAmount");

400     // 4. Validate maxSrcAmount doesn't exceed remaining budget
401     uint256 remainingSrcBudget = order.maxSrcAmount - state.totalSrcSpent;
402     if (slice.maxSrcAmount > remainingSrcBudget) revert ExceedsTotalSrcBudget();

404     // 5. Validate expectedSrcAmount <= maxSrcAmount (can't expect to spend more
            than max)
405     if (slice.expectedSrcAmount > slice.maxSrcAmount) revert InvalidField("
            expectedSrcAmount");
```

Remediation Enforce proportional budget allocation by validating that `slice.maxSrcAmount <= slice.destAmount * order.maxSrcAmount / order.totalDestAmount`. This ensures each slice's maximum source spending is proportional to its destination amount, preventing any slice from exhausting the budget and preserving funds for subsequent slices.

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