Security Assessment Report

Symphony Aggregator V1

December 3, 2025

Summary

The Sec3 team was engaged to conduct a thorough security analysis of the Symphony Aggregator V1.

The artifact of the audit was the source code of the following programs, excluding tests, in https://github.com/Symphony-Exchange/Symphony-Aggregator-Smart-Contract-v1.

The initial audit focused on the following versions and revealed 9 issues or questions.

Task	Type	Commit
Symphony Aggregator Smart Contract v1	Solidity	60281ac6abebbd67f874f4b1a019e6422709807a

This report provides a detailed description of the findings and their respective resolutions.

Table of Contents

Result Overview

Issue	Impact	Status
SYMPHONY AGGREGATOR SMART CONTRACT V1		
[L-01] Missing cap check on `feePercentage` configuration	Low	Resolved
[L-02] Fee bypass via unverified `feeData` input	Low	Resolved
[L-03] Ineffective hardcoded deadline in DEX integration	Low	Acknowledged
[L-04] Unconfigurable `stable` flag in `yakaSwap`	Low	Acknowledged
[I-01] Missing `tokenIn` consistency check	Info	Resolved
[I-02] Missing token path validation in `executeSwaps`	Info	Resolved
[I-03] Missing `tokenIn` validation when `msg.value > 0`	Info	Resolved
[I-04] Redundant approve call in `uniswapV3Swap` function	Info	Resolved
[I-05] Incorrect swap type emission in `uniswapV3ExactInputSingle`	Info	Acknowledged

Findings in Detail

SYMPHONY AGGREGATOR SMART CONTRACT V1

[L-01] Missing cap check on `feePercentage` configuration

> *Identified in commit* `60281ac`.

In the `_processFee` function, the swap fee is calculated using the following formula: `(finalTokenAmount * feePercentage) / 10000`, where the `finalTokenAmount` is the amount of tokens the user receives and the `feePercentage` is a fee ratio configured by the protocol administrator, scaled to a base of `10,000` (i.e., `10,000 = 100%`).

```
/* src/Symphony.sol */
586 | fee = (finalTokenAmount * feePercentage) / 10000;
587 | amountToTransfer = finalTokenAmount - fee;
588 | IERC20(finalTokenAddress).safeTransfer(fee_address, fee);
589 | emit SwapReceipt(
590 |     msg.sender,
591 |     tokenG,
592 |     finalTokenAddress,
593 |     totalAmountIn,
594 |     finalTokenAmount,
595 |     feePercentage,
596 |     fee,
597 |     fee_address
598 | );
```

However, the contract does not enforce an upper bound when setting or updating `feePercentage`.

Setting `feePercentage > 10000` may result in overcharging users or causing calculation errors.

It is recommended to add a cap check when updating `feePercentage`.

Resolution

Fixed by commit `4098702`.

SYMPHONY AGGREGATOR SMART CONTRACT V1

[L-02] Fee bypass via unverified `feeData` input

Identified in commit `60281ac`.

The `_processFee` function determines how to charge and distribute swap fees based on the `feeData` struct:

- If `feeData.feeAddress` is the zero address, the swap fee is calculated using `feePercentage`, and `100%` of the fee is sent to `fee_address`.
- If `feeData.feeAddress` is non-zero, the fee is calculated using `feeData.paramFee`. The `feeSharePercentage` of the fees goes to `feeData.feeAddress`, and the remainder goes to `fee_address`.

```
/* src/Symphony.sol */
560 | function _processFee(
561 |     uint totalAmountIn,
562 |     uint finalTokenAmount,
563 |     address tokenG,
564 |     address finalTokenAddress,
565 |     FeeParams memory feeData
566 | ) internal returns (uint) {
567 |     uint amountToTransfer;
568 |     uint fee;
569 |     if (feeData.feeAddress != address(0)){
570 |         fee = (finalTokenAmount * feeData.paramFee) / 10000;
571 |         uint feeShare = (fee * feeData.feeSharePercentage) / 10000;
572 |         amountToTransfer = finalTokenAmount - fee;
573 |         IERC20(finalTokenAddress).safeTransfer(fee_address, fee - feeShare);
574 |         IERC20(finalTokenAddress).safeTransfer(feeData.feeAddress, feeShare);
575 |         emit SwapReceipt(
576 |             msg.sender,
577 |             tokenG,
578 |             finalTokenAddress,
579 |             totalAmountIn,
580 |             finalTokenAmount,
581 |             feeData.paramFee,
582 |             fee,
583 |             feeData.feeAddress
584 |         );
585 |     }else {
586 |         fee = (finalTokenAmount * feePercentage) / 10000;
587 |         amountToTransfer = finalTokenAmount - fee;
588 |         IERC20(finalTokenAddress).safeTransfer(fee_address, fee);
589 |         emit SwapReceipt(
590 |             msg.sender,
591 |             tokenG,
592 |             finalTokenAddress,
593 |             totalAmountIn,
594 |             finalTokenAmount,
595 |             feePercentage,
596 |             fee,
597 |             fee_address
598 |         );
599 |     }
```

```
600 |     return amountToTransfer;
601 | }
```

However, since `feeData` is a user supplied input that is not verified, a malicious user could manipulate fee parameters and bypass protocol fees. For example:

- Set `feeData.feeAddress` to a non-zero address but set `feeData.paramFee = 0` (zero fee)
- Set `feeData.feeSharePercentage = 10000 (100%)`, diverting all fees to the user controlled feeAddress

These manipulations would result in zero or misdirected fees, effectively bypassing the protocol's intended fee mechanism.

It is recommended to implement strict validation logic for the `feeData` input.

Resolution

Fixed by commit `4098702`.

SYMPHONY AGGREGATOR SMART CONTRACT V1

[L-03] Ineffective hardcoded deadline in DEX integration

Identified in commit `60281ac`.

During the DEX integration process, a `deadline` parameter is used to indicate the expiration time for a swap transaction.

```
/* src/Symphony.sol */
194 | function dragonSwap(
195 |     address tokenIn,
196 |     address tokenOut,
197 |     uint amountIn
198 | ) internal returns (IPool.TokenAmount memory) {
199 |     _ensureApproval(tokenIn, dragonRouterAddress, amountIn);
200 |
201 |     address[] memory path = new address[](2);
202 |     path[0] = tokenIn;
203 |     path[1] = tokenOut;
204 |     uint deadline = block.timestamp + 20 minutes;
205 |     uint[] memory amounts = dragonRouter.swapExactTokensForTokens(
206 |         amountIn,
207 |         0,
208 |         path,
209 |         address(this),
210 |         deadline
211 |     );
```

However, the current implementation sets the `deadline` inside the contract as `block.timestamp + 20 minutes`. Since `block.timestamp` reflects the current block time at the moment of transaction execution, the `deadline` will always be 20 minutes after execution, making it effectively useless as a safeguard against stale or delayed transactions.

It is recommended to accept the `deadline` as a function input parameter (e.g. provided by the user or frontend), rather than computing it within the contract.

Resolution

The team acknowledged this finding.

SYMPHONY AGGREGATOR SMART CONTRACT V1

[L-04] Unconfigurable `stable` flag in `yakaSwap`

> *Identified in commit* `60281ac`.

In the `yakaSwap` function, the constructed `route` sets `stable: false` unconditionally.

This prevents selecting stable pools when beneficial and ignores route preferences from the caller, which can result in failed routes for tokens that primarily trade on stable pools.

```
/* src/Symphony.sol */
242 | route[] memory path = new route[](1);
243 | path[0] = route({
244 |     from: tokenIn, // Replace with actual address
245 |     to: tokenOut, // Replace with actual address
246 |     stable: false  // Set the boolean value
247 | });
```

It is recommended to make the `stable` flag configurable via user input.

Resolution

The team acknowledged this finding.

SYMPHONY AGGREGATOR SMART CONTRACT V1

[I-01] Missing `tokenIn` consistency check

Identified in commit `60281ac`.

In the `executeSwaps` function, the `swapParams` parameter is defined as a two-dimensional array of `SwapParam`, and the function iterates over each item to perform a series of swaps.

At the beginning of execution, the contract sums all `swapParams[i][0].amountIn` values to determine the total input amount and charges the user that amount of tokens.

```
/* src/Symphony.sol */
433 | function executeSwaps(
434 |     Params.SwapParam[][] memory swapParams,
435 |     uint minTotalAmountOut,
436 |     bool conveth,
437 |     FeeParams memory feeData
438 | ) external payable nonReentrant whenNotPaused returns (uint) {
439 |     address tokenG = swapParams[0][0].tokenIn;
440 |     IERC20 token = IERC20(tokenG);
441 |     uint256 totalAmountIn = 0;
442 |     for (uint i = 0; i < swapParams.length; i++){
443 |         totalAmountIn += swapParams[i][0].amountIn;
444 |     }
445 |     if (msg.value > 0) {
446 |         weth.deposit{value: msg.value}();
447 |         require(totalAmountIn == msg.value, "Invalid Input Amount");
448 |     } else {
449 |         token.safeTransferFrom(msg.sender, address(this), totalAmountIn);
450 |     }
```

However, the function does not validate whether all `swapParams[i][0].tokenIn` values refer to the same token. This allows a user to pass in multiple different `tokenIn` tokens, transfer them directly to the aggregator to execute swap.

It is recommended to add a validation in `executeSwaps` to ensure that all `swapParams[i][0].tokenIn` values refer to the same token.

Resolution

Fixed by commit `0f329b4`.

SYMPHONY AGGREGATOR SMART CONTRACT V1

[I-02] Missing token path validation in `executeSwaps`

Identified in commit `60281ac`.

In the `executeSwaps` function, the `swapParams` parameter is defined as a two-dimensional array of `SwapParam`, and the function iterates over `swapParams[i][j]` to perform a series of sequential swaps.

During this process, each swap is expected to use the output of the previous swap as the input for the next. Although the code correctly sets the input amount for the next swap based on the previous output, it does not verify that the input token (`tokenIn`) of the next swap matches the output token (`tokenOut`) of the previous swap.

```
/* src/Symphony.sol */
457 | for (uint j = 0; j < swapParams[i].length; j++) {
458 |     Params.SwapParam memory param = swapParams[i][j];
459 |     IPool.TokenAmount memory result;
460 |     if (param.swapType == 1) {
461 |         result = dragonSwap(
462 |             param.tokenIn,
463 |             param.tokenOut,
464 |             amountInCurrent
465 |         );
466 |     } else if (param.swapType == 2) {
467 |         result = yakaSwap(
468 |             param.tokenIn,
469 |             param.tokenOut,
470 |             amountInCurrent
471 |         );
472 |     } else if (param.swapType == 3) {
473 |         result = donkeSwap(
474 |             param.tokenIn,
475 |             param.tokenOut,
476 |             amountInCurrent
477 |         );
478 |     } else if (param.swapType == 4){
479 |         result = jellySwap(
480 |             param.tokenIn,
481 |             param.tokenOut,
482 |             param.poolAddress,
483 |             amountInCurrent
484 |         );
485 |     }else if(param.swapType == 5){
486 |         result = uniswapV3Swap(
487 |             param.tokenIn,
488 |             param.tokenOut,
489 |             amountInCurrent,
490 |             param.fee
491 |         );
492 |     }else if (checkRouter(param.swapType)) {
493 |         _ensureApproval(param.tokenIn, v3Routers[param.swapType], amountInCurrent);
494 |         IUniswapV3SwapRouter uniswapV3SwapRouter = IUniswapV3SwapRouter(v3Routers[param.swapType]);
```

```
495 |         result = uniswapV3ExactInputSingle(
496 |             uniswapV3SwapRouter,
497 |             param.tokenIn,
498 |             param.tokenOut,
499 |             amountInCurrent,
500 |             param.fee
501 |         );
502 |     } else {
503 |         revert("Invalid swap type");
504 |     }
505 |     amountInCurrent = result.amount; // Update for next swap in path
506 |     pathFinalTokenAddress = result.token;
507 |     pathFinalTokenAmount = result.amount;
508 | }
```

This missing check allows a user to forge arbitrary token paths in `swapParams[i][j]` and directly transfer mismatched tokens to the aggregator, potentially bypassing the expected routing logic or introducing inconsistencies in token flow.

It is recommended to add a validation to ensure that each `swapParams[i][j].tokenIn` matches the `swapParams[i][j - 1].tokenOut`.

Resolution

Fixed by commit `0f329b4`.

SYMPHONY AGGREGATOR SMART CONTRACT V1

[I-03] Missing `tokenIn` validation when `msg.value > 0`

Identified in commit `60281ac`.

In the `executeSwaps` function, if `msg.value > 0`, the contract wraps the native SEI into WSEI and uses it as the `totalAmountIn` for the swap operation. If `msg.value == 0`, the function sums all `swapParams[i][0].amountIn` values to determine the total input amount.

```
/* src/Symphony.sol */
442 | for (uint i = 0; i < swapParams.length; i++){
443 |     totalAmountIn += swapParams[i][0].amountIn;
444 | }
445 | if (msg.value > 0) {
446 |     weth.deposit{value: msg.value}();
447 |     require(totalAmountIn == msg.value, "Invalid Input Amount");
448 | } else {
449 |     token.safeTransferFrom(msg.sender, address(this), totalAmountIn);
450 | }
```

However, when `msg.value > 0`, the function does not validate whether `swapParams[i][0].tokenIn` is WSEI, which is the expected wrapped version of the deposited native token.

This missing check allows a malicious user to forge arbitrary tokenIn values in `swapParams[i][0]`, and bypass the intended input token control by manually transferring mismatched tokens to the aggregator contract before calling `executeSwaps`.

It is recommended to add a validation to ensure that all `swapParams[i][0].tokenIn == WSEI` when `msg.value > 0` to ensure consistency and prevent misuse.

Resolution

Fixed by commit `0f329b4`.

SYMPHONY AGGREGATOR SMART CONTRACT V1

[I-04] Redundant approve call in `uniswapV3Swap` function

Identified in commit `60281ac`.

In the `uniswapV3Swap` function, the contract performs a token swap using the `universalRouter`. The function first calls `_ensureApproval` to grant the `universalRouter` token allowance. Then transfers the input tokens directly to the router before executing the swap.

```
/* src/Symphony.sol */
353 | function uniswapV3Swap(
354 |         address tokenIn,
355 |         address tokenOut,
356 |         uint256 amountIn,
357 |         uint24 fee
358 |     ) internal returns (IPool.TokenAmount memory) {
359 |         _ensureApproval(tokenIn, universalRouterAddress, amountIn);
360 |
361 |         // Prepare the commands for the Universal Router
362 |         IERC20(tokenIn).safeTransfer(address(universalRouter), amountIn);
363 |         bytes memory commands = abi.encodePacked(
364 |             uint8(0x00) // V3_SWAP_EXACT_IN command
365 |         );
366 |
367 |         // Prepare the swap data
368 |         bytes memory swapData = abi.encode(
369 |             address(this), // recipient
370 |             amountIn, // amountIn
371 |             0, // amountOutMinimum
372 |             abi.encodePacked(tokenIn, fee, tokenOut), // path
373 |             false
374 |         );
375 |
376 |         // Prepare the inputs array
377 |         bytes[] memory inputs = new bytes[](1);
378 |         inputs[0] = swapData;
379 |
380 |         // Record the balance before the swap
381 |         uint256 balanceBefore = IERC20(tokenOut).balanceOf(address(this));
382 |
383 |         // Execute the swap using the Universal Router interface
384 |         universalRouter.execute(commands, inputs);
385 |
615 | function _ensureApproval(address token, address spender, uint256 amount) internal {
616 |     uint256 currentAllowance = IERC20(token).allowance(address(this), spender);
617 |
618 |     if (currentAllowance < amount) {
619 |         IERC20(token).forceApprove(spender, amount);
620 |     }
621 | }
```

However, since the `payerIsUser` flag in the `swapData` is set to false, the router will not rely on the allowance. Instead, it will use the tokens that have already been transferred to execute the swap.

It is recommended to remove the initial `_ensureApproval` call, as it is redundant and can be safely omitted.

Resolution

Fixed by commit `0f329b4`.

SYMPHONY AGGREGATOR SMART CONTRACT V1

[I-05] Incorrect swap type emission in `uniswapV3ExactInputSingle`

Identified in commit `60281ac`.

In the `uniswapV3ExactInputSingle` function, the `SwapExecuted` event uses a hardcoded `swapType` of `6`, while the router keys are configured as `[6, 7, 8, 9, 10]`. The dynamic branch routes through different V3 routers based on `param.swapType`. This mislabels swaps in logs and hinders the ability to accurately analyze which router handled the execution.

```
/* src/Symphony.sol */
399 | function uniswapV3ExactInputSingle(
...
422 |     emit SwapExecuted(msg.sender, tokenIn, tokenOut, amountIn, amountOut, 6);
```

It is recommended to pass the actual `swapType` used for the swap into the emit function, ensuring events accurately reflect the executed path.

Resolution

The team acknowledged this finding.

Appendix: Methodology and Scope of Work

Assisted by the Sec3 Scanner developed in-house, the manual audit particularly focused on the following work items:

- Check common security issues.
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work

DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a Sec3 (the "Company") and Symphony (the "Client''). This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

This Report is considered null and void if the Report (or any portion thereof) is altered in any manner.

ABOUT

The Sec3 audit team comprises a group of computer science professors, researchers, and industry veterans with extensive experience in smart contract security, program analysis, testing, and formal verification. We are also building automated security tools that incorporate static analysis, penetration testing, and formal verification.

At Sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.