SMART CONTRACT AUDIT REPORT

for

Symphony Exchange

Prepared By: Xiaomi Huang

PeckShield

February 24, 2025

Document Properties

Client	Symphony Exchange
Title	Smart Contract Audit Report
Target	Symphony Exchange
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 24, 2025	Xuxian Jiang	Final Release
1.0-rc1	February 13, 2025	Xuxian Jiang	Release Candidate #1

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 `Symphony Exchange` 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 Symphony Exchange

`Symphony Exchange` is a high-performance `DEX` aggregator built natively on the `Sei Network`. It seamlessly routes trades through all major `Sei`-based decentralized exchanges, executing swaps at actual quoted prices while minimizing market impact through optimized liquidity pool balancing. The basic information of Symphony Exchange is as follows:

Table 1.1: Basic Information of Symphony Exchange

Item	Description
Issuer	Symphony Exchange
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 24, 2025

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

- https://github.com/Symphony-Exchange/Symphony-Aggregator-Smart-Contract-v1.git (2e6a636)

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

- https://github.com/Symphony-Exchange/Symphony-Aggregator-Smart-Contract-v1.git (eb59c92)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [9]:

- 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 checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

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) [8], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	2	■ ■
Informational	1	■
Total	4	

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 need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and and 1 informational recommendation.

Table 2.1: Key Symphony Exchange Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-002	Medium	Improved executeSwaps() Logic in Symphony	Business Logic	Resolved
PVE-003	Informational	Revisited SwapReceipt Event in _processFee()	Business Logic	Resolved
PVE-003	Low	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 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Symphony
- Category: Coding Practices [6]
- CWE subcategory: CWE-1099 [1]

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 `Symphony::withdrawTokens()` routine that is designed to transfer the funds out of the exchange contract. To accommodate the specific idiosyncrasy, there is a need to user `safeTransfer()`, instead of `transfer()` (line 178).

```
172     function withdrawTokens(address _token, address _to) external onlyOwner {
173         require(_to != address(0), "Invalid address");

175         IERC20 token = IERC20(_token);
176         uint256 balance = token.balanceOf(address(this));

178         require(token.transfer(_to, balance), "Transfer failed");
179     }
```

Listing 3.2: `Symphony::withdrawTokens()`

In the meantime, we also suggest to use the safe-version of `transfer()` in other related routines, including `executeSwaps()`, `_processFee()`, and `uniswapV3Swap()`.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`/`transfer()`/`transferFrom()`. Note this issue affects a number of functions, including `maxApprovals()`, `revokeApprovals()`, and `executeSwaps()`.

Status This issue has been fixed in the following commit: `eb59c92`.

3.2 Improved executeSwaps() Logic in Symphony

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Symphony`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In `Symphony`, there is a core function named `executeSwaps()` that is designed to execute a series of swap operations based on the provided swap parameters. Our analysis shows its logic may be improved, including enhanced user input validation.

In the following, we show the code snippet from the related `executeSwaps()` routine. When the routine is called with accompany non-zero `msg.value`, we need to ensure the local variable `totalAmountIn` matches msg.value, instead of current `totalAmountIn = msg.value` (line 447). Also, for another variable `pathFinalTokenAddress` that represents the output token address, we only need to assign it at the first iteration (when `i=0`) and validate its consistency for remaining iterations (when `i!=0`).

```
433     function executeSwaps(
434         Params.SwapParam[][] memory swapParams,
435         uint minTotalAmountOut,
436         bool conveth,
437         FeeParams memory feeData
438     ) external payable nonReentrant 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             totalAmountIn = msg.value;
448         } else {
449             if (!token.transferFrom(msg.sender, address(this), totalAmountIn))
450                 revert TransferFromFailedError(
451                     msg.sender,
452                     address(this),
453                     totalAmountIn
454                 );
455         }
456         ...
457     }
```

Listing 3.3: `Symphony::executeSwaps()`

Recommendation Improve the above routine to ensure (1) msg.value, if positive, is consistent with the local variable `totalAmountIn` and (2) the output token is always consistent in different iterations.

Status This issue has been fixed in the following commit: `eb59c92`.

3.3 Revisited SwapReceipt Event in _processFee()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Symphony
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

Description

In EVM-compliant chains, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the Symphony contract as an example. This contract has a public function that is used to perform token swaps. While examining the SwapReceipt event, we notice it may be emitted without inaccurate feePercentage information (line 589). The correct feePercentage being used in this specific case should be feeData.paramFee.

```
566     function _processFee(
567         uint totalAmountIn,
568         uint finalTokenAmount,
569         address tokenG,
570         address finalTokenAddress,
571         FeeParams memory feeData
572     ) internal returns (uint) {
573         uint amountToTransfer;
574         uint fee;
575         if (feeData.feeAddress != address(0)){
576             fee = (finalTokenAmount * feeData.paramFee) / 1000;
577             uint feeShare = (fee * feeData.feeSharePercentage) / 1000;
578             amountToTransfer = finalTokenAmount - fee;
579             if (!IERC20(finalTokenAddress).transfer(fee_address, fee - feeShare))
580                 revert TransferFailedError(finalTokenAddress, fee_address, fee -
                        feeShare );
581             if (!IERC20(finalTokenAddress).transfer(feeData.feeAddress, feeShare))
582                 revert TransferFailedError(finalTokenAddress, feeData.feeAddress,
                        feeShare);
583             emit SwapReceipt(
584                 msg.sender,
585                 tokenG,
586                 finalTokenAddress,
587                 totalAmountIn,
588                 finalTokenAmount,
589                 feePercentage,
```

```
                fee,
                feeData.feeAddress
            );
        }
        ...
    }
```

Listing 3.4: `Symphony::_processFee()`

Recommendation Accurately emit the `SwapReceipt` event with correct `feePercentage` information.

Status This issue has been fixed in the following commit: `eb59c92`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Symphony`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the `Symphony Exchange` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., manage token approvals, configure fees, and recover funds). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function maxApprovals(address[] calldata tokens) external onlyOwner {...}
    ...
    function revokeApprovals(address[] calldata tokens) external onlyOwner {...}
    ...
    function setFeePercentage(uint _feePercentage) external onlyOwner {...}
    ...
    function setFeeAddress(address _newFeeAddress) external onlyOwner {...}
    ...
    function withdrawTokens(address _token, address _to) external onlyOwner {...}
    ...
    function withdrawEther(address payable _to) external onlyOwner {...}
```

Listing 3.5: Example Privileged Function(s) in `Symphony`

Note that if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

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

Status This issue has been mitigated as a multisig account will be used to hold the `owner` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Symphony Exchange` protocol, which is a high-performance `DEX` aggregator built natively on the `Sei Network`. It seamlessly routes trades through all major `Sei`-based decentralized exchanges, executing swaps at actual quoted prices while minimizing market impact through optimized liquidity pool balancing. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that `Solidity-based` 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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

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