



# SMART CONTRACT AUDIT REPORT

for

## Oyster Aggregator



Prepared By: Xiaomi Huang

PeckShield  
February 5, 2025

## Document Properties

Client	Oyster
Title	Smart Contract Audit Report
Target	Oyster
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 5, 2025	Xuxian Jiang	Final Release
1.0-rc1	January 26, 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

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About Oyster . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	7
<b>2</b>	<b>Findings</b>	<b>9</b>
2.1	Summary . . . . .	9
2.2	Key Findings . . . . .	10
<b>3</b>	<b>Detailed Results</b>	<b>11</b>
3.1	Improved Validation on Function Arguments . . . . .	11
3.2	Revisited getMidPriceAndBalance() Logic in DEX Adapters . . . . .	12
3.3	Possible Fee Tier Inconsistency in ALBPoolAddress . . . . .	13
3.4	Trust Issue of Admin Keys . . . . .	14
<b>4</b>	<b>Conclusion</b>	<b>16</b>
	References	17

# 1 | Introduction

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

`Oyster` is a cutting-edge on-chain DEX Aggregator solution. Different from existing approaches, `Oyster` does not calculate the pools and find-best-path logic off-chain. Instead, `Oyster` has an on-chain pricing structure that can source liquidity from multiple DEX engines (via respective adapters) and support multiple hops and splitted pools. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Oyster Aggregator

Item	Description
Issuer	Oyster
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 5, 2025

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

- <https://github.com/SynFutures/oyster-aggregator.git> (84a54f8)

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

- <https://github.com/SynFutures/oyster-aggregator.git> (be03ce6)

## 1.2 About PeckShield

PeckShield Inc. [9] 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](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

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

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
Deprecated Uses	
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
Following Other Best Practices	

---

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

## 1.4 Disclaimer

---

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit



Category	Summary
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	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.
<b>Error Conditions, Return Values, Status Codes</b>	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.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logics</b>	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.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	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 `Oyster` protocol implementation. 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 logics, 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	3	
Informational	0	
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 that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in [Section 3](#).

## 2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 3 low-severity vulnerabilities.

Table 2.1: Key Oyster Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Validation on Function Arguments	Coding Practices	Resolved
PVE-002	Low	Revisited <code>getMidPriceAndBalance()</code> Logic in DEX Adapters	Business Logic	Resolved
PVE-003	Low	Possible Fee Tier Inconsistency in <code>ALBPoolAddress</code>	Business Logic	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

## 3 | Detailed Results

### 3.1 Improved Validation on Function Arguments

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OysterAggregator`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

#### Description

The `Oyster` protocol has an on-chain pricing structure that core `OysterAggregator` contract that is the main entry for user interaction. In the process of examining the logic to return swapped tokens to the user, we notice an issue that can better validate the input fee parameters.

In the following, we show the code snippet of the `_routeWithdraw()` routine. As the name indicates, this routine is invoked after all swaps are completed and is used to transfer tokens to original receiver and distribute fees to the protocol and the broker, if any. We notice the broker portion is calculated via the `brokerFeeRate` parameter (line 300) and is validated to be smaller than `10 ** 18` (line 301). Note the validation can be improved with the following statement, i.e. `if (routeFeeRate + brokerFeeRate >= 10 ** 18) revert LibAggregatorErrors.BrokerFeeRateOverflowed();`.

```
292     function _routeWithdraw(address toToken, uint256 receiveAmount, bytes memory feeData
      , uint256 minReturnAmount)
293         internal
294         returns (uint256 userReceiveAmount)
295     {
296         address originToToken = toToken;
297         if (toToken == _ETH_ADDRESS_) {
298             toToken = _WETH_;
299         }
300         (address broker, uint256 brokerFeeRate) = abi.decode(feeData, (address, uint256)
      );
301         if (brokerFeeRate >= 10 ** 18) revert LibAggregatorErrors.
      BrokerFeeRateOverflowed();
```

```

303     ...
304 }

```

Listing 3.1: `OysterAggregator::_routeWithdraw()`

**Recommendation** Revise the above-mentioned routine to thoroughly validate the given broker fee.

**Status** The issue has been resolved in the following commit: `d336fae`.

## 3.2 Revisited `getMidPriceAndBalance()` Logic in DEX Adapters

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

### Description

As mentioned earlier, `Oyster` has an on-chain pricing structure that directly sources liquidity from multiple DEX engines via respective adapters. In the process of examining current adapters, we notice one common core function (to compute the intermediate price and pool balances) can be improved.

In the following, we use the `UniV2Adapter` as an example and show its implementation of the related `getMidPriceAndBalance()` routine. This routine has a rather straightforward logic in computing the swap price from the respective `Uniswap` pools and their balances. However, we notice the fee adjustment logic can be improved. Specifically, when it is a buy operation, the fee adjustment can be improved as  $price = price * 1e18 / (1e18 - feeAdjustment)$ , not  $current\ price * (1e18 + feeAdjustment) / 1e18$  (line 131).

```

106     function getMidPriceAndBalances(address pool, bool isBuy)
107         external
108         view
109         override
110         returns (uint256 price, uint256 token0bal, uint256 token1bal)
111     {
112         (uint112 reserve0, uint112 reserve1,) = IUniswapV2Pair(pool).getReserves();
113         price = uint256(reserve1) * 1e18 / uint256(reserve0);
114
115         // Get token decimals
116         address token0 = IUniswapV2Pair(pool).token0();
117         address token1 = IUniswapV2Pair(pool).token1();
118         uint8 decimals0 = IERC20Metadata(token0).decimals();

```

```
119     uint8 decimals1 = IERC20Metadata(token1).decimals();
121     // Adjust for decimals
122     if (decimals0 > decimals1) {
123         price = price * (10 ** (decimals0 - decimals1));
124     } else if (decimals1 > decimals0) {
125         price = price / (10 ** (decimals1 - decimals0));
126     }
128     // Adjust price for 0.3% fee
129     uint256 feeAdjustment = 0.003e18; // 0.3% in 1e18 format
130     if (isBuy) {
131         price = price * (1e18 + feeAdjustment) / 1e18;
132     } else {
133         price = price * (1e18 - feeAdjustment) / 1e18;
134     }
136     token0bal = IERC20(token0).balanceOf(pool);
137     token1bal = IERC20(token1).balanceOf(pool);
138 }
```

Listing 3.2: UniV2Adapter::getMidPriceAndBalance()

**Recommendation** Revise the above-mentioned routine to properly factor in the fee parameter for the intermediate price calculation. Note other adapters can be similarly improved, including UniV3Adapter, AeroV2Adapter, AeroV3Adapter, PancakeV3Adapter, and ALbV3Adapter.

**Status** The issue has been resolved as the team confirms it is part of the design.

### 3.3 Possible Fee Tier Inconsistency in ALBPoolAddress

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: ALBPoolAddress
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

#### Description

As mentioned earlier, Oyster may source liquidity from multiple DEX engines via respective adapters. In the process of examining the ALBV3Adapter support, we notice an inconsistency in the supported fee tiers with the external AlienBase DEX protocol.

In the following, we show the code snippet from the `getPools()` routine, which supports four fee tiers, i.e., 100, 400, 300, and 750. However, our examination on the AlienBase DEX code base indicates the following supported fee tiers: 750, 3000, and 10000.

```

52     function getPools(address factory, address tokenA, address tokenB) internal pure
       returns (address[] memory pools) {
53         // Standard Uniswap V3 fee tiers
54         uint24[4] memory fees = [uint24(100), uint24(400), uint24(300), uint24(750)];
55         pools = new address[](fees.length);

56         for (uint256 i = 0; i < fees.length; i++) {
57             pools[i] = computeAddress(factory, getPoolKey(tokenA, tokenB, fees[i]));
58         }
59     }
60 }

```

Listing 3.3: ALBPoolAddress::getPools()

**Recommendation** Revise the above-mentioned routine to ensure the fee tiers in ALBV3Adapter are consistent with the external AlienBase DEX.

**Status** The issue has been resolved in the following commit: 721db5b.

### 3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

#### Description

In the Oyster protocol, there is a privileged account, i.e., `owner`. This account plays a critical role in governing and regulating the protocol-wide operations (e.g., collect contract funds, configure tokens/pools, and manage bots). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `OysterAggregator` contract as an example and show the representative functions potentially affected by the privileged account.

```

79     function changeRouteFeeRate(uint256 newFeeRate) public onlyOwner {
80         if (newFeeRate >= 10 ** 18) revert LibAggregatorErrors.NewFeeRateOverflowed();
81         routeFeeRate = newFeeRate;
82     }
83
84     function changeRouteFeeReceiver(address newFeeReceiver) public onlyOwner {
85         if (newFeeReceiver == address(0)) revert LibAggregatorErrors.FeeReceiverInvalid();
86         routeFeeReceiver = newFeeReceiver;
87     }
88
89     /// @notice used for emergency, generally there wouldn't be tokens left
90     function superWithdraw(address token) public onlyOwner {

```

```
91     if (token != _ETH_ADDRESS_) {
92         uint256 restAmount = IERC20(token).universalBalanceOf(address(this));
93         IERC20(token).universalTransfer(payable(routeFeeReceiver), restAmount);
94     } else {
95         uint256 restAmount = address(this).balance;
96         IERC20(_ETH_ADDRESS_).universalTransfer(payable(routeFeeReceiver),
97             restAmount);
98     }
99 }
```

Listing 3.4: Example Privileged Functions in `OysterAggregator`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. Note that 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, the `Config` contract makes use of the proxy contract to allow for future upgrades. The upgrade is a privileged operation and the management of the related admin key also falls in this trust issue.

**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 the team confirms that multi-sig will be adopted for the privileged account.

---

## 4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Oyster` protocol, which is a cutting-edge on-chain DEX Aggregator solution. Different from existing approaches, `Oyster` does not calculate the pools and find-best-path logic off-chain. Instead, `Oyster` has an on-chain pricing structure that can source liquidity from multiple sources (via respective adapters) and support multiple hops and splitted pools. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [2] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [4] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [5] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [6] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [7] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [8] OWASP. Risk Rating Methodology. [https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [9] PeckShield. PeckShield Inc. <https://www.peckshield.com>.