SMART CONTRACT AUDIT REPORT

for

MYX

Prepared By: Xiaomi Huang

PeckShield

February 17, 2024

Document Properties

Client	MYX
Title	Smart Contract Audit Report
Target	MYX
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 17, 2024	Xuxian Jiang	Final Release
1.0-rc1	February 2, 2024	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 MYX 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 audited protocol 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 MYX

MYX is a decentralized derivative exchange with an innovative matching pool mechanism (MPM). It matches long and short traders and takes on net exposure. In a balanced market, this engine supports unlimited open interest with limited capital, allowing MYX to match the capital efficiency of centralized order books, providing low transaction fees and higher LP yields. The protocol uses intelligent rates and exposure hedging mechanisms to ensure protocol stability and provide sustainable high returns. MYX aims to be the most efficient DEX in terms of fund usage. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of MYX

Item	Description
Name	MYX
Website	https://myx.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 17, 2024

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

- https://github.com/innsec/myx-contracts.git (0210164)

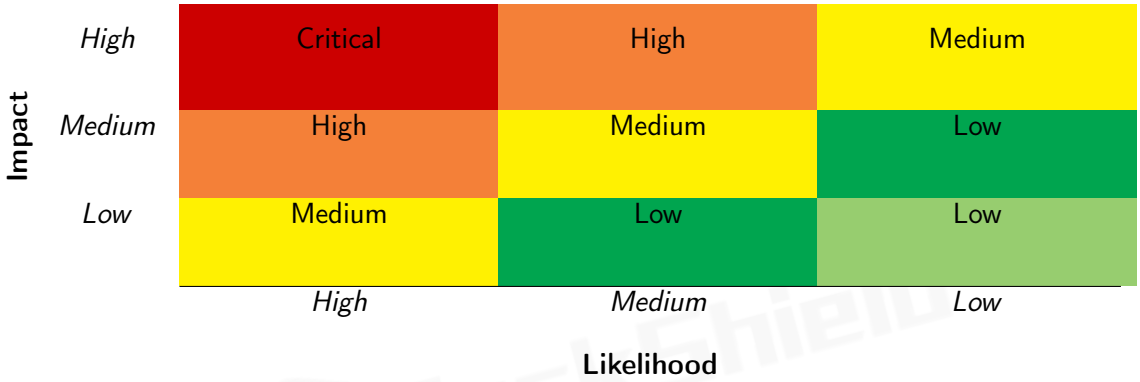
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/innsec/myx-contracts.git (62c8d92)

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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on 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.

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

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 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) [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.

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 Logics	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 MYX 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	1	■
Low	4	■■■■
Informational	1	■
Total	7	

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 high-severity vulnerability, 1 medium-severity vulnerability, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key MYX Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Revisited PaymentType Use in Router Order Creation	Business Logic	Resolved
PVE-002	High	Possible Bypass of maxLeverage Check in Position's Collateral Adjustment	Business Logic	Resolved
PVE-003	Low	Improved Trading Fee Distribution in FeeCollector	Coding Practices	Resolved
PVE-004	Informational	Revisited Average Price Update Logic in PositionManager	Business Logic	Resolved
PVE-005	Low	Incorrect Position Key Generation in PositionKey Library	Coding Practices	Resolved
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-007	Low	Improved Liquidity-Adding Logic in Router	Coding Practices	Resolved

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 Revisited PaymentType Use in Router Order Creation

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Router`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `MYX` protocol has a helper `Router` contract to facilitate the user interactions in creating, adjusting, and canceling user orders. In the process of examining the order-creating logic, we notice it does not use the correct `PaymentType` parameter.

In the following, we show the implementation from the related `createIncreaseOrderWithTpSl()` routine. As the name indicates, this routine is designed to create an increase order with the `take-profit` and `stop-loss` information. The routine assumes the payment is made with `Ether` (line 154), which suggests the need of using `TradingTypes.NetworkFeePaymentType.ETH` as the the payment at lines 169 and 184. Note the same issue is also applicable to `OrderManager`.

```
144     function createIncreaseOrderWithTpSl(
145         TradingTypes.IncreasePositionWithTpSlRequest memory request
146     ) external payable whenNotPaused nonReentrant returns (uint256 orderId) {
147         require(!operationStatus[request.pairIndex].increasePositionDisabled, "disabled"
                );
148         require(
149             request.tradeType != TradingTypes.TradeType.TP &&
150             request.tradeType != TradingTypes.TradeType.SL,
151             "not support"
152         );
153         require(
154             msg.value >= request.networkFeeAmount + request.tpNetworkFeeAmount + request
                    .slNetworkFeeAmount,
155             "insufficient network fees"
156         );
```

```
157         request.account = msg.sender;
158
159         orderId = orderManager.createOrder{value: request.networkFeeAmount}(
160             TradingTypes.CreateOrderRequest({
161                 account: request.account,
162                 pairIndex: request.pairIndex,
163                 tradeType: request.tradeType,
164                 collateral: request.collateral,
165                 openPrice: request.openPrice,
166                 isLong: request.isLong,
167                 sizeAmount: uint256(request.sizeAmount).safeConvertToInt256(),
168                 maxSlippage: request.maxSlippage,
169                 paymentType: TradingTypes.convertPaymentType(request.paymentType),
170                 networkFeeAmount: request.networkFeeAmount,
171                 data: abi.encode(request.account)
172             })
173         );
174
175         // tp/sl
176         this._createTpSl(
177             request.account,
178             request.pairIndex,
179             request.isLong,
180             request.tpPrice,
181             request.tp,
182             request.slPrice,
183             request.sl,
184             request.paymentType,
185             request.tpNetworkFeeAmount,
186             request.slNetworkFeeAmount
187         );
188         return orderId;
189     }
```

Listing 3.1: `Router::createIncreaseOrderWithTpSl()`

Recommendation Revisit the above-mentioned routines to use the correct payment type in the created orders.

Status This issue has been fixed by the following commit: `116de8c`.

3.2 Possible Bypass of maxLeverage Check in Position's Collateral Adjustment

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

Description

The `MYX` protocol has a core `PositionManager` contract that allows the user to create or adjust his/her trading positions. While examining the collateral-adjusting logic, we notice the validation of resulting state needs to be improved.

In the following, we show the collateral-adjusting logic from the related `adjustCollateral()` routine. We notice it calls `position.validLeverage()` (line 296) to validate the position is in a normal state. However, it always makes use of `_sizeAmount=0` argument, which completely bypasses the `maxLeverage` validation (line 144). To fix, there is a need to adjust the `if`-condition to be the following: `if (!simpleVerify && ((_increase && _sizeAmount > 0)|| _collateral<0))`.

```
280     function adjustCollateral(
281         uint256 pairIndex,
282         address account,
283         bool isLong,
284         int256 collateral
285     ) external override onlyRouter {
286         bytes32 positionKey = PositionKey.getPositionKey(account, pairIndex, isLong);
287         Position.Info storage position = positions[positionKey];
288         if (position.positionAmount == 0) {
289             revert("position not exists");
290         }
291 
292         IPool.Pair memory pair = pool.getPair(pairIndex);
293 
294         uint256 price = IPriceFeed(ADDRESS_PROVIDER.priceOracle()).getPriceSafely(pair.
                indexToken);
295         IPool.TradingConfig memory tradingConfig = pool.getTradingConfig(pairIndex);
296         position.validLeverage(
297             pair,
298             price,
299             collateral,
300             0,
301             true,
302             tradingConfig.maxLeverage,
303             tradingConfig.maxPositionAmount,
304             false,
```

```
305             getFundingFee(account, pairIndex, isLong)
306         );
307         ...
308     );
```

Listing 3.2: PositionManager::adjustCollateral()

```
144         if (!simpleVerify && _increase && _sizeAmount > 0) {
145             uint256 collateralDec = uint256(IERC20Metadata(pair.stableToken).decimals())
                    ;
146             uint256 tokenDec = uint256(IERC20Metadata(pair.indexToken).decimals());
147
148             uint256 tokenWad = 10 ** (PrecisionUtils.maxTokenDecimals() - tokenDec);
149             uint256 collateralWad = 10 ** (PrecisionUtils.maxTokenDecimals() -
                    collateralDec);
150
151             uint256 afterPositionD = afterPosition * tokenWad;
152             uint256 availableD = (availableCollateral.abs() * maxLeverage *
                    collateralWad).divPrice(price);
153             require(afterPositionD <= availableD, 'exceeds max leverage');
154         }
```

Listing 3.3: Position::validLeverage()

Recommendation Revise the above-mentioned routines to ensure maxLeverage is properly enforced. Similarly, there is a need to ensure any adjustment on a healthy user position should always result in a healthy state. In other words, there is a need to validate a healthy user position will not become liquidatable after the position adjustment.

Status This issue has been fixed in the following commit: a29b43f2.

3.3 Improved Trading Fee Distribution in FeeCollector

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: FeeCollector
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The MYX protocol has a FeeCollector contract to distribute the trading fee. Our analysis shows the trading fee logic may have an issue for possibly resulting in arithmetic underflow.

To elaborate, we show below the implementation of the distributeTradingFee() routine. This routine depends on a number of fee parameters, including vipFeeRate, referralUserRatio, referralsRatio,

and `maxReferralsRatio`. However, our analysis shows that the fee distribution logic does not consider possible arithmetic underflows. For example, if the given `vipFeeRate` is larger than `PERCENTAGE = 1e8`, the calculation of `vipDiscountAmount = tradingFee - vipTradingFee` (line 202) will be underflowed. Also, if `referralUserRatio > referralsRatio`, the calculation of `referralsAmount - referralUserAmount` (line 218) may be underflowed as well.

```
187     function distributeTradingFee(
188         IPool.Pair memory pair,
189         address account,
190         address keeper,
191         uint256 sizeDelta,
192         uint256 tradingFee,
193         uint256 vipFeeRate,
194         uint256 referralsRatio,
195         uint256 referralUserRatio,
196         address referralOwner
197     ) external override onlyPositionManagerOrLogic returns (uint256 lpAmount, uint256
        vipDiscountAmount) {
198         IPool.TradingFeeConfig memory tradingFeeConfig = pool.getTradingFeeConfig(pair.
            pairIndex);
199
200         // vip discount
201         uint256 vipTradingFee = sizeDelta.mulPercentage(vipFeeRate);
202         vipDiscountAmount = tradingFee - vipTradingFee;
203         userTradingFee[account] += vipDiscountAmount;
204
205         uint256 surplusFee = tradingFee - vipDiscountAmount;
206
207         // referrals amount
208         uint256 referralsAmount;
209         uint256 referralUserAmount;
210         if (referralOwner != address(0)) {
211             referralsAmount = surplusFee.mulPercentage(
212                 Math.min(referralsRatio, maxReferralsRatio)
213             );
214             referralUserAmount = surplusFee.mulPercentage(
215                 Math.min(referralUserRatio, referralsRatio)
216             );
217             userTradingFee[account] += referralUserAmount;
218             referralFee[referralOwner] += referralsAmount - referralUserAmount;
219
220             surplusFee = surplusFee - referralsAmount;
221         }
222         ...
223     }
```

Listing 3.4: `FeeCollector::distributeTradingFee()`

Recommendation Revise the above trade fee distribution logic to avoid unexpected arithmetic underflow.

Status This issue has been fixed by the following commit: `116de8c`.

3.4 Revisited Average Price Update Logic in PositionManager

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PositionManager`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `MYX` protocol has a key `PositionManager` contract that allows the user to create or adjust his/her trading positions. While examining the current position-related logic, we notice the price adjustment of an increased position can be improved.

To elaborate, we show below the code snippet from the related routine. As the name indicates, this routine computes the next average price when a position is increased with `sizeAmount` (line 115). Specifically, for current position of `positionAmount` with its `averagePrice`, if it is increased by `sizeAmount` with the latest mark price `oraclePrice`, the next average price is currently computed as `(positionAmount * averagePrice + sizeAmount * oraclePrice)/(positionAmount + sizeAmount)`, which needs to be revised as `(positionAmount + sizeAmount)/(positionAmount / averagePrice + sizeAmount / oraclePrice)`.

```
106     ...
107     if (position.positionAmount == 0) {
108         position.init(pairIndex, account, isLong, oraclePrice);
109     }
110
111     if (position.positionAmount > 0 && sizeDelta > 0) {
112         position.averagePrice = (position.positionAmount.mulPrice(position.averagePrice)
                +
113             sizeDelta).mulDiv(
114                 PrecisionUtils.pricePrecision(),
115                 (position.positionAmount + sizeAmount)
116             );
117     }
118
119     // update funding fee
120     _updateFundingRate(pairIndex, oraclePrice);
121     _handleCollateral(pairIndex, position, collateral);
122     ...
```

Listing 3.5: `PositionManager::increasePosition()`

Recommendation Revise the above routine to properly compute the next average price when a position is increased.

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

3.5 Incorrect Position Key Generation in PositionKey Library

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PositionKey`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The `MYX` protocol has a `PositionKey` library that is used to compute `position key` based on the given three parameters, i.e., `paccount`, `pairIndex`, and `isLong`. The `position key` is generated by concatenating these three fields into an unsigned integer (`uint256`) and then to a `byte32` string. With that, there is a need to ensure the given `pairIndex` is less than `2 ** (96-32)`, not `2 ** (224)` (line 6).

```
4  library PositionKey {
5      function getPositionKey(address account, uint256 pairIndex, bool isLong) internal
           pure returns (bytes32) {
6          require(pairIndex < 2 ** 224, "ptl");
7          return bytes32(
8              uint256(uint160(account)) << 96  pairIndex « 32  (isLong ? 1 : 0)
9          );
10     }
11 }
```

Listing 3.6: PositionKey::getPositionKey()

Recommendation Revise the above logic to properly validate the given `pairIndex` input.

Status The issue has been fixed by following the above suggestion.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `MYX` protocol, there is a privileged `admin` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., parameter configuration and sensitive operation execution). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `Pool` contract as an example and show the representative functions potentially affected by the privileges of the `admin` account.

```
126     function setRiskReserve(address _riskReserve) external onlyPoolAdmin {
127         address oldAddress = riskReserve;
128         riskReserve = _riskReserve;
129         emit UpdateRiskReserve(msg.sender, oldAddress, _riskReserve);
130     }

132     function setFeeCollector(address _feeCollector) external onlyPoolAdmin {
133         address oldAddress = feeCollector;
134         feeCollector = _feeCollector;
135         emit UpdateFeeCollector(msg.sender, oldAddress, _feeCollector);
136     }

138     function setPositionManager(address _positionManager) external onlyPoolAdmin {
139         address oldAddress = positionManager;
140         positionManager = _positionManager;
141         emit UpdatePositionManager(msg.sender, oldAddress, _positionManager);
142     }

144     function setOrderManager(address _orderManager) external onlyPoolAdmin {
145         address oldAddress = orderManager;
146         orderManager = _orderManager;
147         emit UpdateOrderManager(msg.sender, oldAddress, _orderManager);
148     }

150     function setRouter(address _router) external onlyPoolAdmin {
151         address oldAddress = router;
152         router = _router;
153         emit UpdateRouter(msg.sender, oldAddress, _router);
154     }

156     function addStableToken(address _token) external onlyPoolAdmin {
157         isStableToken[_token] = true;
```

```
158         emit AddStableToken(msg.sender, _token);
159     }

161     function removeStableToken(address _token) external onlyPoolAdmin {
162         delete isStableToken[_token];
163         emit RemoveStableToken(msg.sender, _token);
164     }
```

Listing 3.7: Example Privileged Operations in `Pool`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the privileged account may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the administrative privileges to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been mitigated as the team confirmed the `owner` will be transferred to a multi-sig account.

3.7 Improved Liquidity-Adding Logic in Router

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Router`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, `MYX` has a `Router` contract to facilitate user interactions, including liquidity addition and removal. Liquidity providers may receive profit for the provided liquidity. In the process of examining the liquidity-adding logic, we notice the implementation may be improved.

To elaborate, we show below the implementation of the related `addLiquidityCallback()` routine. As the name indicates, this routine behaves as a callback to add the requested liquidity amount into the pool. When the requested `indexToken` is `WETH`, we notice the use of `safeTransferFrom()` to transfer `indexToken` from `Router` to pool. This `WETH` transfer can be improved as `IERC20(indexToken).safeTransfer(msg.sender, uint256(amountIndex))` (line 652).

```
641     function addLiquidityCallback(
642         address indexToken,
```

```
643         address stableToken,
644         uint256 amountIndex,
645         uint256 amountStable,
646         bytes calldata data
647     ) external override onlyPool {
648         address sender = abi.decode(data, (address));
649 
650         if (amountIndex > 0) {
651             if (indexToken == ADDRESS_PROVIDER.WETH()) {
652                 IERC20(indexToken).safeTransferFrom(address(this), msg.sender, uint256(
                        amountIndex));
653             } else {
654                 IERC20(indexToken).safeTransferFrom(sender, msg.sender, uint256(
                        amountIndex));
655             }
656         }
657         if (amountStable > 0) {
658             IERC20(stableToken).safeTransferFrom(sender, msg.sender, uint256(
                    amountStable));
659         }
660     }
```

Listing 3.8: `Router::addLiquidityCallback()`

In addition, the `Router` contract has another helper `addLiquidityETH()` to facilitate the support of `Ether`. In this helper routine, we can improve it by enforcing `require(indexToken == ADDRESS_PROVIDER.WETH())`.

Recommendation Revisit the above routine to properly add the liquidity.

Status This issue has been fixed by the following commit: `116de8c`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the MYX protocol, which is a decentralized derivative exchange with an innovative matching pool mechanism (MPM). It matches long and short traders and takes on net exposure. In a balanced market, this engine supports unlimited open interest with limited capital, allowing MYX to match the capital efficiency of centralized order books, providing low transaction fees and higher LP yields. The protocol uses intelligent rates and exposure hedging mechanisms to ensure protocol stability and provide sustainable high returns. MYX aims to be the most efficient DEX in terms of fund usage. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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.