Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

2 Audit Methodology

3 Project Overview

3.1 Project Introduction

3.2 Vulnerability Information

4 Code Overview

4.1 Contracts Description

4.2 Visibility Description

4.3 Vulnerability Summary

5 Audit Result

6 Statement

1 Executive Summary

On 2024.01.26, the SlowMist security team received the MYX team's security audit application for MYX Protocol Phase2, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "white box lead, black, grey box assists" to conduct a complete security test on the project in the way closest to the real attack.

The test method information:

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open source code, non-open source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project team should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.
Suggestion	There are better practices for coding or architecture.

2 Audit Methodology

The security audit process of SlowMist security team for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using automated analysis tools.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

Serial Number	Audit Class	Audit Subclass
1	Overflow Audit	-
2	Reentrancy Attack Audit	-
3	Replay Attack Audit	-
4	Flashloan Attack Audit	-
5	Race Conditions Audit	Reordering Attack Audit
6	Permission Vulnerability Audit	Access Control Audit
		Excessive Authority Audit
7	Security Design Audit	External Module Safe Use Audit
		Compiler Version Security Audit
		Hard-coded Address Security Audit
		Fallback Function Safe Use Audit
		Show Coding Security Audit
		Function Return Value Security Audit
		External Call Function Security Audit

Serial Number	Audit Class	Audit Subclass
7	Security Design Audit	Block data Dependence Security Audit
		tx.origin Authentication Security Audit
8	Denial of Service Audit	-
9	Gas Optimization Audit	-
10	Design Logic Audit	-
11	Variable Coverage Vulnerability Audit	-
12	"False Top-up" Vulnerability Audit	-
13	Scoping and Declarations Audit	-
14	Malicious Event Log Audit	-
15	Arithmetic Accuracy Deviation Audit	-
16	Uninitialized Storage Pointer Audit	-

3 Project Overview

3.1 Project Introduction

MYX Finance is a P2Pool2P decentralized perpetual contract protocol. MYX offers USDC-margined perpetual future contracts up to 50x leverage. This audit is based on the iterations since the previous audit.

3.2 Vulnerability Information

The following is the status of the vulnerabilities found in this audit:

NO	Title	Category	Level	Status
N1	Risk of lacking access control for	Authority Control Vulnerability Audit	Critical	Fixed

NO	Title	Category	Level	Status
	backtracking operations			
N2	Optimizable function visibility	Others	Suggestion	Fixed
N3	Bypassable network fee collection method	Design Logic Audit	Information	Acknowledged
N4	Unsafe int type casting	Others	Low	Fixed
N5	Risk of extra `msg.value` being locked	Design Logic Audit	Low	Fixed
N6	Forgeable network fee	Design Logic Audit	Critical	Fixed
N7	Missing check for `msg.value`	Design Logic Audit	Low	Fixed
N8	Potential risk of operators transferring arbitrary token amounts	Authority Control Vulnerability Audit	Medium	Acknowledged
N9	Potential risk of USDT tokens being non-transferrable	Design Logic Audit	Critical	Fixed
N10	Duplicate fee deduction issues	Design Logic Audit	High	Fixed
N11	Not reverting when failing to update historical price via pyth	Design Logic Audit	Medium	Fixed

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/innsec/myx-contracts

commit: 0210164bdb33246eedfdd096201d9a55338b6bff

Fixed Version:

https://github.com/innsec/myx-contracts

commit: a525ad3bcd56f8cb11299030a2798b8af7e43ddb

Audit Scope:

The audit only covers the iterative portions since the last audit up to the current audited version.

The main network address of the contract is as follows:

The code was not deployed to the mainnet.

4.2 Visibility Description

The SlowMist Security team analyzed the visibility of major contracts during the audit, the result as follows:

Backtracker			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
enterBacktracking	External	Can Modify State	whenNotBacktracking
quitBacktracking	External	Can Modify State	whenBacktracking
_requireNotBacktracking	Internal	-	-
_requireBacktracking	Internal	-	-

MultipleTransfer			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
<Receive Ether>	External	Payable	-
addOperator	External	Can Modify State	onlyRole
addMerkleManager	External	Can Modify State	onlyRole

MultipleTransfer			
updateMerkleRoot	External	Can Modify State	onlyMerkleManager
pause	Public	Can Modify State	onlyOwner
unpause	Public	Can Modify State	onlyOwner
batchTransferETH	Public	Can Modify State	onlyOperator whenNotPaused
batchTransferERC20	Public	Can Modify State	onlyOperator whenNotPaused
emergencyWithdrawETH	Public	Can Modify State	onlyOwner
emergencyWithdrawERC20	Public	Can Modify State	onlyOwner

UiPoolDataProvider			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
getPairsData	Public	-	-
getUserTokenData	External	-	-

UiPositionDataProvider			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
getPositionsData	Public	-	-

PositionCaller			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
getFundingFees	Public	-	-
getTradingFees	Public	-	-

4.3 Vulnerability Summary

[N1] [Critical] Risk of lacking access control for backtracking operations

Category: Authority Control Vulnerability Audit

Content

In the Backtracker contract, the backtracking parameter controls whether the protocol oracle uses historical prices. When the keeper executes liquidation via the Executor's setPricesAndLiquidatePositions, it will first call the Backtracker's enterBacktracking to enable backtracking, then update the historical price via updateHistoricalPrice to use the correct historical price for liquidation. After liquidation is complete, the historical price is removed and backtracking mode is exited to avoid other business logic getting incorrect prices. Unfortunately, any user can arbitrarily modify the backtracking state by calling enterBacktracking/quitBacktracking. This allows users to set backtracking to true before facing liquidation. Then when the keeper tries to execute liquidation, due to the whenNotBacktracking restriction, it will be unable to successfully call enterBacktracking leading to a DoS risk. More severely, when backtracking is set to true without any valid historical prices set for the oracle, getHistoricalPrice will return 0. Malicious users can use this incorrect price to operate positions and profit unfairly.

Code location: contracts/core/Backtracker.sol#L29-L38

```
function enterBacktracking(uint64 _backtrackRound) external whenNotBacktracking {
    backtracking = true;
    backtrackRound = _backtrackRound;
    emit Backtracking(msg.sender, _backtrackRound);
}

function quitBacktracking() external whenBacktracking {
    backtracking = false;
    emit UnBacktracking(msg.sender);
}
```

Solution

It is recommended to add access control to `enterBacktracking/quitBacktracking`, allowing only the Executor contract to call them to mitigate this risk.

Status

Fixed

[N2] [Suggestion] Optimizable function visibility

Category: Others

Content

In the Executor contract, the setPricesHistorical function has external visibility, with `require(msg.sender == address(this), "internal")` to check the caller can only be itself. This adds unnecessary gas costs.

The same is true for the `_createTpSl` function in the Router contract.

Code location:

contracts/core/Executor.sol#L227

```
    function setPricesHistorical(
        ...
    ) external payable {
        require(msg.sender == address(this), "internal");
        ...
    }

contracts/core/Router.sol#L203
    function _createTpSl(
        ...
    ) public payable returns (uint256 tpOrderId, uint256 slOrderId) {
        require(msg.sender == address(this), "internal");
        ...
    }
```

Solution

It is recommended to make setPricesHistorical internal visibility to optimize gas usage.

Status

Fixed

[N3] [Information] Bypassable network fee collection method

Category: Design Logic Audit

Content

In the protocol, when users create orders via the Router contract, they need to pay a network fee. The network fee is processed according to InnerPaymentType in OrderManager and stored in orderNetworkFees for fee distribution by FeeCollector. However, the network fee amount is specified by users themselves when creating orders, rather than mandatorily collected by the protocol. This allows users to pass in a 0 network fee and still successfully create orders. So users can bypass the fee by providing a custom amount.

Code location:

contracts/core/Router.sol#L170

contracts/core/Router.sol#L265

contracts/core/Router.sol#L290

contracts/core/Router.sol#L316

Solution

If not intended, it is recommended to set a fixed fee variable in the protocol to enforce network fee collection, instead of relying on user input.

Status

Acknowledged; After communicating with the project team, the project team stated that this was the expected design.

[N4] [Low] Unsafe int type casting

Category: Others

Content

In the PositionManager contract, the needADL function checks whether the protocol needs to perform ADL. When calculating afterExposedPositions, it forcibly casts the uint256 executionSize parameter to int256. If executionSize is very large, this force casting may cause overflow risks.

Code location: contracts/core/PositionManager.sol#L761-L765

```
    function needADL(
        ...
    ) external view returns (bool need, uint256 needADLAmount) {
        ...
        int256 afterExposedPositions = exposedPositions;
        if (isLong) {
```

```
            afterExposedPositions -= int256(executionSize);
        } else {
            afterExposedPositions += int256(executionSize);
        }
        ...
    }
```

Solution

It is recommended to use Int256Utils library's safeConvertToInt256 function for safe conversion.

Status

Fixed

[N5] [Low] Risk of extra `msg.value` being locked

Category: Design Logic Audit

Content

In the Router contract, when users create orders via createIncreaseOrderWithTpSl, it checks that `msg.value` must be greater than the network fee paid by the user. However, note that the excess is not refunded to the user, which will cause native tokens exceeding the fee to be locked in the contract.

Code location: contracts/core/Router.sol#L154

```
    function createIncreaseOrderWithTpSl(
        TradingTypes.IncreasePositionWithTpSlRequest memory request
    ) external payable whenNotPaused nonReentrant returns (uint256 orderId) {
        ...
        require(
            msg.value >= request.networkFeeAmount + request.tpNetworkFeeAmount +
request.slNetworkFeeAmount,
            "insufficient network fees"
        );
        ...
    }
```

Solution

It is recommended to implement refund logic, so that when `msg.value` is greater than the network fee, the excess should be refunded to the user.

Status

Fixed

[N6] [Critical] Forgeable network fee

Category: Design Logic Audit

Content

In the Router contract, users can create orders via createIncreaseOrder/createDecreaseOrder functions. When creating orders, network fees are paid to the protocol via `msg.value`. The network fee amount is specified by the user externally, then the Router contract calls the orderManager's createOrder function to pass the user's `msg.value` and networkFeeAmount into orderManager for processing. Unfortunately, the protocol does not check if the networkFeeAmount matches `msg.value`. This allows users to forge a large networkFeeAmount when creating orders, but only pay a small `msg.value`. Malicious users can exploit this to assign an incorrect value to the `orderNetworkFees[ordersIndex]` variable, ultimately allowing the keeper to steal extra assets from the pool when claiming fees.

Code location:

contracts/core/Router.sol#L254-L267

contracts/core/Router.sol#L279-L292

```
    function createIncreaseOrder(
        TradingTypes.IncreasePositionRequest memory request
    ) external payable whenNotPaused nonReentrant returns (uint256 orderId) {
        ...

        return
            orderManager.createOrder{value: msg.value}(
                TradingTypes.CreateOrderRequest({
                    ...
                    networkFeeAmount: request.networkFeeAmount,
                    data: abi.encode(request.account)
                })
            );
    }

    function createDecreaseOrder(
        TradingTypes.DecreasePositionRequest memory request
    ) external payable whenNotPaused nonReentrant returns (uint256) {
```

```
        ...
        return
            orderManager.createOrder{value: msg.value}(
                TradingTypes.CreateOrderRequest({
                    ...
                    networkFeeAmount: request.networkFeeAmount,
                    data: abi.encode(request.account)
                })
            );
    }
```

Solution

It is recommended to enforce checking network fee equals `msg.value` in the Router contract.

Status

Fixed

[N7] [Low] Missing check for `msg.value`

Category: Design Logic Audit

Content

In the Router contract, users can batch create decrease orders via the createDecreaseOrders function. In the for loop, it directly passes in `request.networkFeeAmount` as `msg.value` when calling createOrder. It does not check if the remaining `msg.value` from the user is enough to pay the networkFeeAmount. This could allow users to maliciously drain native tokens incorrectly sent to the Router contract.

Code location: contracts/core/Router.sol#L305

```
    function createDecreaseOrders(
        TradingTypes.DecreasePositionRequest[] memory requests
    ) external payable whenNotPaused nonReentrant returns (uint256[] memory orderIds)
{
        orderIds = new uint256[](requests.length);
        for (uint256 i = 0; i < requests.length; i++) {
            TradingTypes.DecreasePositionRequest memory request = requests[i];

            require(!operationStatus[request.pairIndex].decreasePositionDisabled,
"disabled");

            orderIds[i] = orderManager.createOrder{value: request.networkFeeAmount}(
                TradingTypes.CreateOrderRequest({
                    ...
```

```
                })
            );
        }
        return orderIds;
    }
```

Solution

It is recommended to check inside the for loop if the remaining user `msg.value` is sufficient to pay the networkFeeAmount.

Status

Fixed

[N8] [Medium] Potential risk of operators transferring arbitrary token amounts

Category: Authority Control Vulnerability Audit

Content

In the MultipleTransfer contract, operators can batch transfer tokens in the contract to recipient addresses verified via Merkle proofs using the batchTransferETH/batchTransferERC20 functions. However, note that the Merkle proof only verifies legitimacy of the recipients, it does not contain the transfer amounts. Thus operators can transfer arbitrary token amounts to any recipient addresses that pass Merkle verification.

Code location:

contracts/periphery/MultipleTransfer.sol#L61

contracts/periphery/MultipleTransfer.sol#L77

```
    function batchTransferETH(
        ...
    ) public onlyOperator whenNotPaused {
        require(recipients.length == amounts.length, "Recipients and amounts length
mismatch");

        for (uint i = 0; i < recipients.length; i++) {
            bytes32 leaf = keccak256(abi.encodePacked(recipients[i]));
            require(MerkleProof.verify(proofs[i], merkleRoot, leaf), "Invalid Merkle
Proof");

            payable(recipients[i]).transfer(amounts[i]);
        }
    }
```

```
    function batchTransferERC20(
        ...
    ) public onlyOperator whenNotPaused {
        require(recipients.length == amounts.length, "Recipients and amounts length
mismatch");

        for (uint i = 0; i < recipients.length; i++) {
            bytes32 leaf = keccak256(abi.encodePacked(recipients[i]));
            require(MerkleProof.verify(proofs[i], merkleRoot, leaf), "Invalid Merkle
Proof");
            require(token.transfer(recipients[i], amounts[i]), "Transfer failed");
        }
    }
```

Solution

If not intended, it is recommended to also check the token transfer amounts during Merkle verification.

Status

Acknowledged; After communicating with the project team, the project team stated that this was the expected design.

[N9] [Critical] Potential risk of USDT tokens being non-transferrable

Category: Design Logic Audit

Content

In the MultipleTransfer contract, operators can batch transfer any ERC20 tokens via the batchTransferERC20 function. It checks the return value of the token transfers via `require(token.transfer(recipients[i], amounts[i]))`. However, note that for tokens without a return value, they can never pass this check as the return is always 0. For example, USDT tokens on Ethereum mainnet have a transfer function without a return value. This results in these non-ERC20 standard tokens being non-transferrable through this method.

Code location: contracts/periphery/MultipleTransfer.sol#L53C1-L81C6

```
    function batchTransferERC20(
        ...
    ) public onlyOperator whenNotPaused {
        require(recipients.length == amounts.length, "Recipients and amounts length
mismatch");

        for (uint i = 0; i < recipients.length; i++) {
```

```
            ...
            require(token.transfer(recipients[i], amounts[i]), "Transfer failed");
        }
    }
```

Solution

It is recommended to use OpenZeppelin's SafeERC20 library for token transfers.

Status

Fixed

[N10] [High] Duplicate fee deduction issues

Category: Design Logic Audit

Content

In the OrderManager contract, when performing the createOrder operation, the user's paid network fee is processed first. When the user pays with native tokens, the protocol directly transfers `msg.value` to the pool. But when the user pays with COLLATERAL, the protocol deducts collateral from the user's position, and also directly transfers network fees from the user's wallet to the pool contract via `_transferOrderCollateral`. This equates to deducting fees twice, causing users to unnecessarily pay duplicate fees.

Code location: contracts/core/OrderManager.sol#L148-L154

```
    function createOrder(
        TradingTypes.CreateOrderRequest calldata request
    ) public payable onlyExecutorAndRouter returns (uint256 orderId) {
        ...
        } else if (request.paymentType == TradingTypes.InnerPaymentType.COLLATERAL) {
            ...
                collateral -= request.networkFeeAmount.safeConvertToInt256();
                _transferOrderCollateral(
                    pair.stableToken,
                    request.networkFeeAmount,
                    address(pool),
                    request.data
                );
            }
        }
        ...
    }
```

Solution

When the user selects COLLATERAL payment, it is recommended to keep only one fee collection method.

Status

Fixed

[N11] [Medium] Not reverting when failing to update historical price via pyth

Category: Design Logic Audit

Content

As pointed out in the previous audit, although pyth oracle updates can fail, the transaction does not revert. Thus when updating historical prices via the parsePriceFeedUpdates function, if failure cases are not handled, it may store incorrect historical prices in the oracle. This can impact business logic.

Code location: contracts/oracle/PythOraclePriceFeed.sol#L124

```
    function updateHistoricalPrice(
        ...
    ) external payable onlyExecutor onlyBacktracking override {
        ...
        PythStructs.PriceFeed[] memory priceFeeds =
pyth.parsePriceFeedUpdates(updateData, priceIds, publishTime, publishTime);
        ...
    }
```

Solution

It is recommended to check if parsePriceFeedUpdates succeeds, and revert the transaction when it fails.

Status

Fixed

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002401310004	SlowMist Security Team	2024.01.26 - 2024.01.31	Passed

Summary conclusion: The SlowMist security team uses a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 3 critical risks, 1 high risk, 2 medium risk, 3 low risk, 1 suggestion, and 1 Information. All the findings were fixed or acknowledged. The code was not deployed to the mainnet.

6 Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist