Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2023.11.09, the SlowMist security team received the MYX team's security audit application for MYX Protocol, 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 primarily focuses on its core, oracle, pool, token and tools modules. The core module is the entry point for users and keepers, used to handle user liquidity and manage order operations. The oracle module fetches prices from Pyth and Chainlink to provide necessary price feeds for the protocol. The pool module manages protocol liquidity, with all user added liquidity kept in the pool. The token module implements the specific logic for the MYX token. The tools module implements the token swap and timelock contracts. Please note that this audit does not include the earn module.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Token decimal is not checked when adding pair	Others	Low	Fixed
N2	Missing event record	Others	Suggestion	Fixed
N3	Redundant lpFeeDistributeP parameter	Others	Low	Fixed
N4	Redundant functions	Others	Suggestion	Fixed
N5	Reduction of the total amount without checking whether the total amount is greater than the reserve amount	Design Logic Audit	Low	Fixed
N6	Reentrancy risk when add/remove liquidity	Reentrancy Vulnerability	Critical	Fixed
N7	Risk of bypassing price updates through Pool contracts	Design Logic Audit	Critical	Fixed
N8	Redundant addLiquidityForAccount Functions in Pool Contracts	Design Logic Audit	Suggestion	Fixed
N9	Add liquidity and deflationary tokens compatibility issue	Design Logic Audit	Information	Acknowledged
N10	Risk that Rebase-type tokens are not compatible with the protocol	Design Logic Audit	Medium	Acknowledged
N11	Potential Risks of Differences in Index and Stable Token Price Markers	Design Logic Audit	High	Acknowledged

NO	Title	Category	Level	Status
N12	Suggestions for a reasonable kOfSwap setup	Others	Suggestion	Acknowledged
N13	Redundant pair token decimal conversion	Arithmetic Accuracy Deviation Vulnerability	Low	Fixed
N14	Risk of slippage fees being locked in	Design Logic Audit	Medium	Acknowledged
N15	Potential problems with error event logging	Malicious Event Log Audit	Low	Fixed
N16	Discounts are not taken into account when calculating through the getDepositAmount function	Design Logic Audit	Suggestion	Acknowledged
N17	The getReceivedAmount calculation does not take into account the lack of available balance for both tokens	Design Logic Audit	Suggestion	Fixed
N18	Incorrect token swap when pool token balance is insufficient	Design Logic Audit	Critical	Fixed
N19	Incorrect way to get the token price when using the chainlink oracle	Design Logic Audit	Critical	Fixed
N20	Potential issues with using latestAnswer() to get prices	Design Logic Audit	Low	Fixed
N21	Lack of precision conversion for the calculation of ammountInMaximum	Arithmetic Accuracy Deviation Vulnerability	Critical	Fixed
N22	Incorrect processing of price decimal	Arithmetic Accuracy	Critical	Fixed

NO	Title	Category	Level	Status
	returned by chainlink	Deviation Vulnerability		
N23	Incomplete judgement in updateTradingFeeConfig function	Others	Low	Fixed
N24	Parameters are missing range checking	Others	Low	Fixed
N25	Potential risk of not being able to approve swap again	Design Logic Audit	Medium	Fixed
N26	Failure to check whether the profit is greater than totalAmount results in failure to successfully obtain the totalAmount value of the Pool	Integer Overflow and Underflow Vulnerability	High	Fixed
N27	Potential risks of phishing when using tx.origin for authentication	tx.origin Authentication Vulnerability	Suggestion	Fixed
N28	Incorrect getPriceNoOlderThan age	Design Logic Audit	High	Fixed
N29	Incorrect decimal handling of Pyth oracle prices	Arithmetic Accuracy Deviation Vulnerability	Critical	Fixed
N30	Potential risk that users can still withdraw their margin when the position is liquidated	Design Logic Audit	Medium	Fixed
N31	Incorrect slippage check	Design Logic Audit	Critical	Acknowledged
N32	Missing decimal conversion for	Arithmetic Accuracy	Critical	Fixed

NO	Title	Category	Level	Status
	`exposureAmountChecker` operation	Deviation Vulnerability		
N33	The user's order was not canceled as designed	Design Logic Audit	Suggestion	Fixed
N34	Potential risks of Keeper role doing evil	Authority Control Vulnerability Audit	Medium	Fixed
N35	Market orders that are not fully executed still use full collateral	Others	Information	Acknowledged
N36	Inaccurate comparison operator between executionSize and sizeAmount	Others	Suggestion	Acknowledged
N37	Cancelling all orders may lead to a denial-of-service (DoS) attack	Denial of Service Vulnerability	High	Fixed
N38	It is not possible to execute ADL for orders in liquidation status	Design Logic Audit	Critical	Fixed
N39	During ADL, the order's needADL status was not checked	Design Logic Audit	Low	Acknowledged
N40	Not checking the difference between needADLAmount and executeTotalAmount causes gas waste	Gas Optimization Audit	Low	Fixed
N41	Bypass price update and execute order	Design Logic Audit	Medium	Fixed
N42	Unsafe price fetching during `_swapInUni` operation	Design Logic Audit	Low	Acknowledged
N43	Redundant claim interface	Gas Optimization Audit	Suggestion	Fixed

NO	Title	Category	Level	Status
N44	Fee collection does not follow the Checks-Effects-Interactions (CEI) principle	Reentrancy Vulnerability	Low	Fixed
N45	Liquidity calculation in funding fees does not match expected design	Others	Information	Fixed
N46	Incorrect old addressPositionManager retrieval	Design Logic Audit	Low	Fixed
N47	Optimizable `msg.sender` checks	Gas Optimization Audit	Suggestion	Fixed
N48	Adding liquidity without checking if the pair is enabled	Design Logic Audit	Low	Fixed
N49	Redundant data type conversion	Gas Optimization Audit	Suggestion	Fixed
N50	Potential issues with long-short market reversals	Others	Information	Acknowledged
N51	The protocol cannot effectively conduct ADL due to defects in needADL	Design Logic Audit	Critical	Fixed
N52	Optimized collateral calculation when a user is liquidated	Design Logic Audit	Suggestion	Fixed
N53	Potential overflow risks caused by type conversion	Arithmetic Accuracy Deviation Vulnerability	Medium	Fixed
N54	Unvalidated TP/SL prices	Design Logic Audit	Suggestion	Acknowledged
N55	Risk of over-privilege	Authority Control Vulnerability Audit	Medium	Acknowledged

4 Code Overview

4.1 Contracts Description

Audit Version:

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

commit: e6b6514305e4da4c043e0c427721c8e1d343493b

Fixed Version:

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

commit: 07c0fc4636babd360580a8209781d8a9ef5c8454

Audit Scope:

- contracts/core
- contracts/helpers
- contracts/interfaces
- contracts/libraries
- contracts/oracle
- contracts/pool
- contracts/tools
- contracts/AddressesProvider.sol
- contracts/RoleManager.sol

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:

Executor			
Function Name	Visibility	Mutability	Modifiers

Executor			
<Constructor>	Public	Can Modify State	Roleable
setPaused	External	Can Modify State	onlyPoolAdmin
setUnPaused	External	Can Modify State	onlyPoolAdmin
setPricesAndExecuteIncreaseMarketOrders	External	Payable	whenNotPaused onlyPositionKeeper
setPricesAndExecuteDecreaseMarketOrders	External	Payable	whenNotPaused onlyPositionKeeper
setPricesAndExecuteIncreaseLimitOrders	External	Payable	whenNotPaused onlyPositionKeeper
setPricesAndExecuteDecreaseLimitOrders	External	Payable	whenNotPaused onlyPositionKeeper
setPricesAndExecuteADL	External	Payable	whenNotPaused onlyPositionKeeper
setPricesAndLiquidatePositions	External	Payable	whenNotPaused onlyPositionKeeper
setPrices	External	Payable	-
needADL	External	-	-

FeeCollector			
Function Name	Visibility	Mutability	Modifiers
initialize	Public	Can Modify State	initializer
getLevelDiscounts	External	-	-
updatePositionManagerAddress	External	Can Modify State	onlyPoolAdmin
updateStakingPoolAddress	External	Can Modify State	onlyPoolAdmin
updateLevelDiscountRatios	External	Can Modify State	onlyPoolAdmin
updateLevelDiscountRatio	External	Can Modify State	onlyPoolAdmin
updateMaxReferralsRatio	External	Can Modify State	onlyPoolAdmin

FeeCollector			
claimStakingTradingFee	External	Can Modify State	onlyStakingPool
claimTreasuryFee	External	Can Modify State	onlyPoolAdmin
claimKeeperTradingFee	External	Can Modify State	nonReentrant
claimUserTradingFee	External	Can Modify State	nonReentrant
distributeTradingFee	External	Can Modify State	onlyPositionManager
_claimUserTradingFee	Internal	Can Modify State	-
_updateLevelDiscountRatio	Internal	Can Modify State	-

FundingRate			
Function Name	Visibility	Mutability	Modifiers
initialize	Public	Can Modify State	initializer
updateFundingFeeConfig	External	Can Modify State	onlyPoolAdmin
getFundingInterval	Public	-	-
getFundingRate	Public	-	-

OrderManager			
Function Name	Visibility	Mutability	Modifiers
initialize	Public	Can Modify State	initializer
setRouter	External	Can Modify State	onlyPoolAdmin
createOrder	Public	Can Modify State	-
cancelOrder	External	Can Modify State	onlyExecutorAndRouter
_cancelOrder	Private	Can Modify State	-
cancelAllPositionOrders	External	Can Modify State	onlyExecutor

OrderManager			
increaseOrderExecutedSize	External	Can Modify State	onlyExecutor
removeOrderFromPosition	Public	Can Modify State	onlyExecutor
removeIncreaseMarketOrders	External	Can Modify State	onlyExecutor
removeIncreaseLimitOrders	External	Can Modify State	onlyExecutor
removeDecreaseMarketOrders	External	Can Modify State	onlyExecutor
removeDecreaseLimitOrders	External	Can Modify State	onlyExecutor
setOrderNeedADL	External	Can Modify State	onlyExecutor
saveOrderTpSl	External	Can Modify State	onlyRouter
removeOrderTpSl	External	Can Modify State	onlyExecutor
_transferOrderCollateral	Internal	Can Modify State	-
_saveIncreaseOrder	Internal	Can Modify State	-
_saveDecreaseOrder	Internal	Can Modify State	-
_cancelIncreaseOrder	Internal	Can Modify State	-
_cancelDecreaseOrder	Internal	Can Modify State	-
_removeOrderAndRefundCollateral	Internal	Can Modify State	-
_addOrderToPosition	Private	Can Modify State	-
_removeOrderFromPosition	Private	Can Modify State	-
getIncreaseOrder	Public	-	-
getDecreaseOrder	Public	-	-
getOrderTpSl	Public	-	-
getPositionOrders	Public	-	-

PositionManager			
Function Name	Visibility	Mutability	Modifiers
initialize	Public	Can Modify State	initializer
setRouter	External	Can Modify State	onlyPoolAdmin
increasePosition	External	Can Modify State	onlyExecutor
decreasePosition	External	Can Modify State	onlyExecutor
adjustCollateral	External	Can Modify State	-
updateFundingRate	External	Can Modify State	-
_takeFundingFeeAddTraderFee	Internal	Can Modify State	-
_currentLpProfit	Internal	-	-
_settleLPPosition	Internal	Can Modify State	-
_calLpProfit	Internal	Can Modify State	-
_handleCollateral	Internal	Can Modify State	-
_tradingFee	Internal	-	-
_updateFundingRate	Internal	Can Modify State	-
_nextFundingRate	Internal	-	-
lpProfit	External	-	-
getTradingFee	External	-	-
getFundingFee	Public	-	-
getCurrentFundingRate	External	-	-
getNextFundingRate	External	-	-
getNextFundingRateUpdateTime	External	-	-
needADL	External	-	-

PositionManager			
getExposedPositions	Public	-	-
getPosition	Public	-	-
getPositionByKey	Public	-	-
getPositionKey	Public	-	-

RiskReserve			
Function Name	Visibility	Mutability	Modifiers
initialize	Public	Can Modify State	initializer
updateDaoAddress	External	Can Modify State	onlyPoolAdmin
updatePositionManagerAddress	External	Can Modify State	onlyPoolAdmin
updatePoolAddress	External	Can Modify State	onlyPoolAdmin
increase	External	Can Modify State	onlyPositionManager
decrease	External	Can Modify State	onlyPositionManager
recharge	External	Can Modify State	-
withdraw	External	Can Modify State	onlyDao

Router			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
salvageToken	External	Can Modify State	onlyPoolAdmin
setPaused	External	Can Modify State	onlyPoolAdmin
setUnPaused	External	Can Modify State	onlyPoolAdmin
wrapWETH	External	Payable	-

Router			
setPriceAndAdjustCollateral	External	Payable	-
setPriceAndUpdateFundingRate	External	Payable	-
createIncreaseOrderWithTpSl	External	Can Modify State	whenNotPaused nonReentrant
createIncreaseOrder	External	Can Modify State	whenNotPaused nonReentrant
createDecreaseOrder	External	Can Modify State	whenNotPaused nonReentrant
createDecreaseOrders	External	Can Modify State	whenNotPaused nonReentrant
_checkOrderAccount	Private	-	-
cancelOrder	External	Can Modify State	whenNotPaused nonReentrant
cancelOrders	External	Can Modify State	whenNotPaused nonReentrant
cancelPositionOrders	External	Can Modify State	whenNotPaused nonReentrant
addOrderTpSl	External	Can Modify State	whenNotPaused nonReentrant
createTpSl	External	Can Modify State	whenNotPaused nonReentrant
addLiquidity	External	Payable	whenNotPaused nonReentrant
addLiquidityForAccount	External	Payable	whenNotPaused nonReentrant
removeLiquidity	External	Payable	whenNotPaused nonReentrant
removeLiquidityForAccount	External	Payable	whenNotPaused nonReentrant
removeLiquidityCallback	External	Can Modify State	onlyPool
createOrderCallback	External	Can Modify State	onlyOrderManager

Router			
addLiquidityCallback	External	Can Modify State	onlyPool

ExecutionLogic			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
updateExecutor	External	Can Modify State	onlyPoolAdmin
updateMaxTimeDelay	External	Can Modify State	onlyPoolAdmin
executeIncreaseMarketOrders	External	Can Modify State	onlyExecutorOrKeeper
executeIncreaseLimitOrders	External	Can Modify State	onlyExecutorOrKeeper
executeIncreaseOrder	External	Can Modify State	onlyExecutorOrKeeper
executeDecreaseMarketOrders	External	Can Modify State	onlyExecutorOrKeeper
executeDecreaseLimitOrders	External	Can Modify State	onlyExecutorOrKeeper
executeDecreaseOrder	External	Can Modify State	onlyExecutorOrKeeper
_executeDecreaseOrder	Internal	Can Modify State	-
_createOrderTpSl	Internal	Can Modify State	-
needADL	Public	-	-
executeADLAndDecreaseOrder	External	Can Modify State	onlyExecutorOrKeeper

LiquidationLogic			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
updateExecutor	External	Can Modify State	onlyPoolAdmin
liquidatePositions	External	Can Modify State	onlyExecutorOrKeeper

LiquidationLogic			
liquidationPosition	External	Can Modify State	onlyExecutorOrKeeper
_executeLiquidationOrder	Private	Can Modify State	-
_needLiquidation	Private	-	-

ChainlinkPriceFeed			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Roleable
decimals	Public	-	-
setChainlinkFlags	External	Can Modify State	onlyPoolAdmin
setTokenConfig	External	Can Modify State	onlyTimelock
_setAssetPrices	Private	Can Modify State	-
getPrice	Public	-	-
getPriceSafely	External	-	-
getPrimaryPrice	Public	-	-

IndexPriceFeed			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
decimals	Public	-	-
updatePrice	External	Can Modify State	onlyKeeper
getPrice	External	-	-
getPriceSafely	External	-	-
_setAssetPrices	Private	Can Modify State	-

PythOraclePriceFeed			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
updatePythAddress	External	Can Modify State	onlyTimelock
setTokenPriceIds	External	Can Modify State	onlyTimelock
updatePrice	External	Payable	-
getPrice	External	-	-
getPriceSafely	External	-	-
_getPriceId	Internal	-	-
_returnPriceWithDecimals	Internal	-	-
_setTokenPriceIds	Internal	Can Modify State	-
decimals	Public	-	-

Pool			
Function Name	Visibility	Mutability	Modifiers
initialize	Public	Can Modify State	initializer
<Receive Ether>	External	Payable	-
_unwrapWETH	Private	Can Modify State	-
setSpotSwap	External	Can Modify State	onlyPoolAdmin
setRiskReserve	External	Can Modify State	onlyPoolAdmin
setFeeCollector	External	Can Modify State	onlyPoolAdmin
setPositionManager	External	Can Modify State	onlyPoolAdmin
setOrderManager	External	Can Modify State	onlyPoolAdmin
addStableToken	External	Can Modify State	onlyPoolAdmin

Pool			
removeStableToken	External	Can Modify State	onlyPoolAdmin
addPair	External	Can Modify State	onlyPoolAdmin
updatePair	External	Can Modify State	onlyPoolAdmin
updateTradingConfig	External	Can Modify State	onlyPoolAdmin
updateTradingFeeConfig	External	Can Modify State	onlyPoolAdmin
increaseTotalAmount	Public	Can Modify State	onlyPositionManager
_increaseTotalAmount	Internal	Can Modify State	-
decreaseTotalAmount	Public	Can Modify State	onlyPositionManager
_decreaseTotalAmount	Internal	Can Modify State	-
increaseReserveAmount	External	Can Modify State	onlyPositionManager
decreaseReserveAmoun t	External	Can Modify State	onlyPositionManager
updateAveragePrice	External	Can Modify State	onlyPositionManager
setLPStableProfit	External	Can Modify State	onlyPositionManagerOrFeeCollecto r
setLPIndexProfit	External	Can Modify State	onlyPositionManager
addLiquidity	External	Can Modify State	-
addLiquidityForAccount	External	Can Modify State	-
removeLiquidity	External	Can Modify State	-
_transferToken	Internal	Can Modify State	-
_swapInUni	Private	Can Modify State	-
getMintLpAmount	External	-	-
_getDiscount	Internal	-	-
_getStableTotalAmount	Internal	-	-

Pool			
_getIndexTotalAmount	Internal	-	-
_addLiquidity	Private	Can Modify State	-
_removeLiquidity	Private	Can Modify State	-
claimFee	External	Can Modify State	onlyTreasury
lpFairPrice	Public	-	-
getDepositAmount	External	-	-
getReceivedAmount	Public	-	-
transferTokenTo	External	Can Modify State	transferAllowed
transferTokenOrSwap	External	Can Modify State	transferAllowed
getProfit	Public	-	-
getVault	Public	-	-
getPair	Public	-	-
getTradingConfig	External	-	-
getTradingFeeConfig	External	-	-

PoolToken			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC20 Roleable
mint	External	Can Modify State	onlyMiner
burn	External	Can Modify State	-
setMiner	External	Can Modify State	-

PoolTokenFactory			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
createPoolToken	External	Can Modify State	-

4.3 Vulnerability Summary

[N1] [Low] Token decimal is not checked when adding pair

Category: Others

Content

In the Pool contract, the admin role can add index tokens and stable tokens to form a pair through the addPair function, and create LP tokens through the poolTokenFactory contract. There are a large number of decimal conversions in the Pool contract, which requires that the decimal of all tokens cannot exceed 18. Unfortunately, the decimal of the added token is not checked in the addPair function, which may cause the admin role to mistakenly add tokens of unexpected decimal, causing the pair to not work properly.

Code location: contracts/pool/Pool.sol#L138-L157

```
    function addPair(address _indexToken, address _stableToken) external
onlyPoolAdmin {
        require(_indexToken != address(0) && _stableToken != address(0), "!0");
        require(isStableToken[_stableToken], "!st");
        require(getPairIndex[_indexToken][_stableToken] == 0, "ex");

        ...
    }
```

Solution

It is recommended to check whether the decimal of the token is less than or equal to 18 when performing addPair operation.

Status

Fixed

[N2] [Suggestion] Missing event record

Category: Others

Content

In the Pool contract, the admin role can modify the sensitive parameters of the contract through the setSpotSwap, setRiskReserve, setFeeCollector, setPositionManager, setOrderManager, addStableToken, removeStableToken, updatePair, updateTradingConfig, updateTradingFeeConfig functions, but no events are recorded.

This will make it impossible for the project team and the community to quickly audit the operation records of privileged roles.

The updateExecutor function in the ExecutionLogic contract is used to modify the executor address, but it does not record events.

The updateExecutor function in the LiquidationLogic contract is the same.

The setRouter function in the OrderManager contract is also the same.

The setRouter function in the PositionManager contract also has this issue.

Code location:

contracts/pool/Pool.sol#L109-L135

contracts/pool/Pool.sol#L159-L219

```
    function setSpotSwap(address _spotSwap) external onlyPoolAdmin {
        spotSwap = _spotSwap;
    }

    function setRiskReserve(address _riskReserve) external onlyPoolAdmin {
        riskReserve = _riskReserve;
    }

    function setFeeCollector(address _feeCollector) external onlyPoolAdmin {
        feeCollector = _feeCollector;
    }

    function setPositionManager(address _positionManager) external onlyPoolAdmin {
        positionManager = _positionManager;
    }

    function setOrderManager(address _orderManager) external onlyPoolAdmin {
```

```
        orderManager = _orderManager;
    }

    function addStableToken(address _token) external onlyPoolAdmin {
        isStableToken[_token] = true;
    }

    function removeStableToken(address _token) external onlyPoolAdmin {
        delete isStableToken[_token];
    }
    ...
```

contracts/core/logic/ExecutionLogic.sol#L65

contracts/core/logic/LiquidationLogic.sol#L59

```
    function updateExecutor(address _executor) external override onlyPoolAdmin {
        executor = _executor;
    }
```

contracts/core/OrderManager.sol#L87

contracts/core/PositionManager.sol#L76

```
    function setRouter(address _router) external onlyPoolAdmin {
        router = _router;
    }
```

Solution

It is to record events when modifying sensitive parameters of the protocol for subsequent self-examination or community review.

Status

Fixed

[N3] [Low] Redundant lpFeeDistributeP parameter

Category: Others

Content

In the Pool contract, the admin role can update the pair configuration through the updatePair function. The lpFeeDistributeP parameter is defined in the Pair structure, which is used to determine the fee distribution ratio,

but it is not configured in the updatePair function. Instead, the lpFeeDistributeP parameter is also defined in the TradingFeeConfig structure, and the fee distribution ratio used by the protocol comes from TradingFeeConfig. Therefore the lpFeeDistributeP parameter in the Pair structure is redundant.

Code location:

contracts/interfaces/IPool.sol#L83

contracts/interfaces/IPool.sol#L102

```
struct Pair {
    ...
    uint256 lpFeeDistributeP;
}

struct TradingFeeConfig {
    ...
    // distribute
    uint256 lpFeeDistributeP;
   ...
}
```

Solution

It is recommended to clarify design expectations and remove redundant parameters.

Status

Fixed

[N4] [Suggestion] Redundant functions

Category: Others

Content

The increaseTotalAmount and decreaseTotalAmount functions are used to increase/decrease the total amount of a vault in a Pool contract, and are only allowed to be called by the PositionManager contract, but there is no call to the increaseTotalAmount and decreaseTotalAmount functions in the PositionManager contract. The increaseTotalAmount and decreaseTotalAmount functions are not called by the PositionManager contract. Therefore, the increaseTotalAmount and decreaseTotalAmount functions are redundant in the Pool contract and cannot be called in the current protocol.

The setLPIndexProfit function has the same problem described above.

Code location:

contracts/pool/Pool.sol#L221

contracts/pool/Pool.sol#L246

contracts/pool/Pool.sol#L339

```
    function increaseTotalAmount(
        uint256 _pairIndex,
        uint256 _indexAmount,
        uint256 _stableAmount
    ) public onlyPositionManager {
        _increaseTotalAmount(_pairIndex, _indexAmount, _stableAmount);
    }

    function decreaseTotalAmount(
        uint256 _pairIndex,
        uint256 _indexAmount,
        uint256 _stableAmount
    ) public onlyPositionManager {
        _decreaseTotalAmount(_pairIndex, _indexAmount, _stableAmount);
    }

    function setLPIndexProfit(uint256 _pairIndex, int256 _profit) external
onlyPositionManager {
        ...
    }
```

Solution

If this is not the intended design, it is recommended that redundant functions be removed. If this is an interface reserved for future iterations it is recommended that the visibility of the function be changed to external.

Status

Fixed

[N5] [Low] Reduction of the total amount without checking whether the total amount is greater than the reserve amount

Category: Design Logic Audit

Content

In a Pool contract, the increaseReserveAmount function is used to increase the reserve amount of the vault, but it must ensure that the total reserve amount does not exceed the total amount. The `_decreaseTotalAmount` function reduces the total amount of a vault, but does not check that the new total amount is still greater than the total reserve amount when it reduces the total amount. This will result in a situation where the total amount may be less than the reserve amount.

Consider the following fixes:

```
function _decreaseTotalAmount(
    ...
) internal {
    Vault storage vault = vaults[_pairIndex];
    vault.indexTotalAmount = vault.indexTotalAmount - _indexAmount;
    vault.stableTotalAmount = vault.stableTotalAmount - _stableAmount;
    require(vault.indexTotalAmount >= vault.indexReservedAmount, "iia");
    require(
        vault.stableTotalAmount >= vault.stableReservedAmount,
        "ist"
    );
    ...
}
```

Code location: contracts/pool/Pool.sol#L260-L261

```
function _decreaseTotalAmount(
    uint256 _pairIndex,
    uint256 _indexAmount,
    uint256 _stableAmount
) internal {
    Vault storage vault = vaults[_pairIndex];
    vault.indexTotalAmount = vault.indexTotalAmount - _indexAmount;
    vault.stableTotalAmount = vault.stableTotalAmount - _stableAmount;
    ...
}
```

Solution

It is recommended to check that the total amount must be greater than or equal to the reserve amount when performing the `_decreaseTotalAmount` operation.

Status

Fixed

[N6] [Critical] Reentrancy risk when add/remove liquidity

Category: Reentrancy Vulnerability

Content

In a Pool contract, any user can add/remove liquidity via addLiquidity/addLiquidityForAccount/removeLiquidity. This allows the user to pass in the account to be liquidated via the `data` parameter and transfer assets via a callback to `msg.sender`. This allows users to interact directly with Pool contracts without having to go through a Router contract, which is a great convenience for users.

Unfortunately, the callback operation leads to the risk of reentry, where a malicious user can provide liquidity through a contract they created and reenter the addLiquidity function again in the callback operation when liquidity is removed, and the price of the LP will be lower than expected because the total amount of the vault has been reduced while the total supply of the LP has not been burned. This will result in more LP tokens being available to the user during the reentry.

Here is a simple example of an exploit:

1. the malicious user creates a contract containing the addLiquidityCallback and removeLiquidityCallback functions. 2. the malicious user performs the addLiquidity operation through this contract.
2. The malicious user calls the addLiquidity function of the Pool contract through the malicious contract to add liquidity, at which point the protocol will mint LP tokens to the user as expected.
3. the malicious user receives the LP tokens and calls the removeLiquidity function of the Pool contract, the removeLiquidity function will call back to the malicious contract via the removeLiquidityCallback function.
4. The malicious contract reenters the addLiquidity function of the Pool contract during the callback process. It should be noted that at the time of the callback, the total amount of the vault has already been reduced and the LPs with liquidity removed have not yet been burned.

   _decreaseTotalAmount(_pairIndex, receiveIndexTokenAmount, receiveStableTokenAmount);

ILiquidityCallback(msg.sender).removeLiquidityCallback(pair.pairToken, _amount, data);

IPoolToken(pair.pairToken).burn(_amount);

5. When reentering the addLiquidity function to calculate the amount of LPs to be minted from the LP price, since the LP price is calculated by dividing the total amount of the vault by the total supply of LPs. This makes the LP price smaller than expected during the reentry process, so more LP tokens will be minted during the addLiquidity operation than expected.
6. After the reentry is complete, a malicious user can acquire more LP tokens than expected to steal additional profit.

Code location:

contracts/pool/Pool.sol#L354-L399

contracts/pool/Pool.sol#L740-L743

contracts/pool/Pool.sol#L582

contracts/pool/Pool.sol#L745-L754

```
    function _removeLiquidity(
        address payable _receiver,
        uint256 _pairIndex,
        uint256 _amount,
        bool useETH,
        bytes calldata data
    )
        private
        returns (
            uint256 receiveIndexTokenAmount,
            uint256 receiveStableTokenAmount,
            uint256 feeAmount
        )
    {
        ...

        _decreaseTotalAmount(_pairIndex, receiveIndexTokenAmount,
receiveStableTokenAmount);

        ILiquidityCallback(msg.sender).removeLiquidityCallback(pair.pairToken,
_amount, data);
        IPoolToken(pair.pairToken).burn(_amount);

        if (useETH && pair.indexToken == ADDRESS_PROVIDER.WETH()) {
```

```
            _unwrapWETH(receiveIndexTokenAmount, _receiver);
        } else {
            IERC20(pair.indexToken).safeTransfer(_receiver, receiveIndexTokenAmount);
        }

        IERC20(pair.stableToken).safeTransfer(_receiver, receiveStableTokenAmount);

        feeTokenAmounts[pair.indexToken] += feeIndexTokenAmount;
        feeTokenAmounts[pair.stableToken] += feeStableTokenAmount;

        ...
    }
```

Solution

It is recommended to allow users to add/remove mobility only through Router contracts, which not only unifies the protocol entry but also facilitates the management of user behaviour.

It is also recommended to move the removeLiquidityCallback operation to the very beginning of the function to follow the Checks-Effects-Interactions principle and to reduce risk. It is also recommended to move the feeTokenAmounts modification to before the transfer operation to the user.

Status

Fixed

[N7] [Critical] Risk of bypassing price updates through Pool contracts

Category: Design Logic Audit

Content

In a perpetual contract protocol, the market is highly dependent on the accuracy and timeliness of the oracle's prices, so the protocol uses the updatePrice function of the oracle to first refresh the price to ensure that the price is fresh when adding/removing liquidity, executing orders, and so on.

In Pool contracts, however, users are allowed to add/remove liquidity directly without price refresh. This allows users to mint/burn LP tokens at older prices, and malicious users can mint more LP tokens than expected at a lower old price, and then burn LP tokens at a higher new price after refreshing the price to make a bigger profit.

Code location:

contracts/pool/Pool.sol#L354-L399

contracts/core/Router.sol#L380

contracts/core/Router.sol#L408

contracts/core/Router.sol#L434

contracts/core/Router.sol#L461

```
    function addLiquidity(
        ...
    )
        external
        payable
        whenNotPaused
        nonReentrant
        returns (uint256 mintAmount, address slipToken, uint256 slipAmount)
    {
        IPythOraclePriceFeed(ADDRESS_PROVIDER.priceOracle()).updatePrice{value:
msg.value}(tokens, updateData);
        ...
    }

    ...

    function addLiquidity(
        ...
    ) external returns (uint256 mintAmount, address slipToken, uint256 slipAmount) {
        ...
    }

    function addLiquidityForAccount(
        ...
    ) external returns (uint256 mintAmount, address slipToken, uint256 slipAmount) {
        ...
    }

    function removeLiquidity(
        ...
    )
        external
        returns (uint256 receivedIndexAmount, uint256 receivedStableAmount, uint256
feeAmount)
    {
        ...
    }
```

Solution

As mentioned earlier, it is recommended that users are only allowed to operate through the Router contract as

an entry point.

Status

Fixed

[N8] [Suggestion] Redundant addLiquidityForAccount Functions in Pool Contracts

Category: Design Logic Audit

Content

The addLiquidity and addLiquidityForAccount functions are both used in Pool contracts to add liquidity to a pool, and both functions take exactly the same parameters and return the same value, and the addLiquidity function can also be used to add liquidity to a specific account. Therefore the addLiquidityForAccount function is redundant.

Code location: contracts/pool/Pool.sol#L354-L376

```
function addLiquidity(
    address recipient,
    uint256 _pairIndex,
    uint256 _indexAmount,
    uint256 _stableAmount,
    bytes calldata data
) external returns (uint256 mintAmount, address slipToken, uint256 slipAmount) {
    ...
}

function addLiquidityForAccount(
    address recipient,
    uint256 _pairIndex,
    uint256 _indexAmount,
    uint256 _stableAmount,
    bytes calldata data
) external returns (uint256 mintAmount, address slipToken, uint256 slipAmount) {
    ...
}
```

Solution

It is recommended to keep only one function with the same functionality to save gas by reducing the contract size.

Status

Fixed

[N9] [Information] Add liquidity and deflationary tokens compatibility issue

Category: Design Logic Audit

Content

In a Pool contract, when a user adds liquidity, it transfers the user's collateral to the pool via the `_transferToken` function. The `_transferToken` function performs a transferFrom operation to transfer tokens via the callback addLiquidityCallback function, and after the tokens have been successfully transferred the token balance in the contract is checked to see if it is as expected. This will make deflationary tokens incompatible with the protocol, as the deflationary tokens will have a smaller balance in the recipient after the transfer than the amount passed in by the user. This makes it impossible to pass the balance check.

The `_transferOrderCollateral` function in the OrderManager contract also has this issue.

Code location: contracts/pool/Pool.sol#L420-L430

```
function _transferToken(
    ...
) internal {
    ...

    if (indexAmount > 0)
        require(
            balanceIndexBefore.add(indexAmount) <=
IERC20(indexToken).balanceOf(address(this)),
            "ti"
        );
    if (stableAmount > 0) {
        require(
            balanceStableBefore.add(stableAmount) <=
                IERC20(stableToken).balanceOf(address(this)),
            "ts"
        );
    }
}
```

contracts/core/OrderManager.sol#L355

```
    function _transferOrderCollateral(
        address collateral,
        uint256 collateralAmount,
        address to,
        bytes calldata data
    ) internal {
        uint256 balanceBefore = IERC20(collateral).balanceOf(to);

        if (collateralAmount > 0) {
            IOrderCallback(msg.sender).createOrderCallback(collateral,
collateralAmount, to, data);
        }
        require(balanceBefore.add(collateralAmount) <=
IERC20(collateral).balanceOf(to), "tc");
    }
```

Solution

If the protocol requires direct deflationary tokens then it is recommended to use the difference in balance of the contract before and after the token transfer as the actual number of receipts for liquidity addition.

Status

Acknowledged; After communicating with the project team, they've stated that the protocol will not support deflationary tokens.

[N10] [Medium] Risk that Rebase-type tokens are not compatible with the protocol

Category: Design Logic Audit

Content

The liquidity of individual tokens in a pool contract is recorded using a vault, and the calculation of the pool's balance and value relies on the TotalAmount/ReservedAmount property in the vault. However, for rebase type tokens (e.g. AMPL), due to their periodic rebase, the actual token balance of the pool is not the same as that recorded in the vault. Therefore, when the actual balance of the pool is less than the value recorded in the vault, it may not be possible to successfully honour the same amount of liquidity for the user as was deposited.

In fact, having rebase-type tokens on the protocol also creates more risk exposure.

Code location: contracts/interfaces/IPool.sol#L63-L69

```
struct Vault {
    uint256 indexTotalAmount; // total amount of tokens
    uint256 indexReservedAmount; // amount of tokens reserved for open positions
    uint256 stableTotalAmount;
    uint256 stableReservedAmount;
    uint256 averagePrice;
}
```

Solution

It is recommended not to allow rebase tokens to be listed, or to update the balance before rebase token operations.

Status

Acknowledged; After communicating with the project team, they've stated that the protocol will not support rebase tokens.

[N11] [High] Potential Risks of Differences in Index and Stable Token Price Markers

Category: Design Logic Audit

Content

In the Pool contract, when a user adds/removes liquidity, the expected number of index tokens and stable tokens in the pool is calculated using the expectIndexTokenP parameter. The calculation is: (total value of index tokens + total value of stable tokens)*expectIndexTokenP.

However, it should be noted that the total value of index tokens is calculated by obtaining the price of index tokens from the oracle, while the total value of stable tokens is calculated by simply converting its decimal value to 1e18. Therefore, the total value of index tokens is denominated in USD, while the total value of stable tokens is the stable tokens themselves. This means that the two tokens have different price tags.

Historically, there have been instances where mainstream stablecoins, including USDT and USDC tokens, have depegged, which means that the price of a stable coin will not be stable to $1, and therefore it does not make sense to add up the values of different price tags. When stablecoins are depegged in the market, malicious users can arbitrage and cause the protocol to incur large amounts of bad debt.

The same problem also exists in the LiquidationLogic contract.

It has not been confirmed whether this will also affect the calculation of user positions.

Code location:

contracts/pool/Pool.sol#L509

contracts/pool/Pool.sol#L832

contracts/pool/Pool.sol#L917

```
    function getMintLpAmount(
        ...
    )
        external
        view
        override
        returns (
            ...
        )
    {
        ...
        uint256 indexTotalDeltaWad = uint256(TokenHelper.convertTokenAmountWithPrice(
            pair.indexToken, int256(_getIndexTotalAmount(pair, vault, price)), 18,
price));
        uint256 stableTotalDeltaWad = uint256(TokenHelper.convertTokenAmountTo(
            pair.stableToken, int256(_getStableTotalAmount(pair, vault, price)), 18));

        uint256 indexDepositDeltaWad =
uint256(TokenHelper.convertTokenAmountWithPrice(
            pair.indexToken, int256(afterFeeIndexAmount), 18, price));
        uint256 stableDepositDeltaWad = uint256(TokenHelper.convertTokenAmountTo(
            pair.stableToken, int256(afterFeeStableAmount), 18));

        uint256 slipDeltaWad;
        uint256 discountRate;
        uint256 discountAmount;
        if (indexTotalDeltaWad + stableTotalDeltaWad > 0) {
            // after deposit
            uint256 totalIndexTotalDeltaWad = indexTotalDeltaWad +
indexDepositDeltaWad;
            uint256 totalStableTotalDeltaWad = stableTotalDeltaWad +
stableDepositDeltaWad;

            // expect delta
            uint256 totalDelta = totalIndexTotalDeltaWad + totalStableTotalDeltaWad;
            ...
            } else if (totalStableTotalDeltaWad > expectStableDeltaWad) {
                ...
            }
```

```
        }
        ...
    }
```

contracts/core/logic/LiquidationLogic.sol#L265-277

```
    function _needLiquidation(bytes32 positionKey, uint256 price) private view
returns (bool) {
        ...

        if (exposureAsset <= 0) {
            need = true;
        } else {
            uint256 maintainMarginWad = uint256(
                TokenHelper.convertTokenAmountWithPrice(
                    pair.indexToken,
                    int256(position.positionAmount),
                    18,
                    position.averagePrice
                )
            ) * tradingConfig.maintainMarginRate;
            uint256 netAssetWad = uint256(
                TokenHelper.convertTokenAmountTo(pair.stableToken, exposureAsset, 18)
            );

            uint256 riskRate = maintainMarginWad / netAssetWad;
            need = riskRate >= PrecisionUtils.percentage();
        }
        return need;
    }
```

Solution

It is recommended that when calculating the value of stable tokens, you also use the oracle to get the USD price of the tokens. Alternatively, it could be standardised to stable tokens.

Status

Acknowledged; After communicating with the project team, the project team indicated this is intended design and the stable token is currently pegged to USD for pricing. Note that there may be severe consequences if the stable token becomes unpegged from the anchor price

[N12] [Suggestion] Suggestions for a reasonable kOfSwap setup

Category: Others

Content

In the Pool contract, there is a vAMM function to virtually swap index tokens with stable tokens when liquidity is added. when a user adds a token that is not required by the pool, the pool will virtually swap it to the pool's required tokens via the vAMM, and the slippage loss incurred in the process will be charged as a fee.

When doing the virtual swap, its using the kOfSwap parameter as the virtual reserve amount for the vAMM and using the price provided by the external oracle for the reserve calculation: `Math.sqrt(Math.mulDiv(k, price, pricePrecision))`. We all know that only when the depth of the vAMM virtual reserve is greater, the slippage loss suffered when making large virtual swaps is lower.

Note that the decimal for price is 1e30, so there is a risk of overflow if the kOfSwap parameter is set too large, and when the kOfSwap parameter is set small, this can lead to excessive slippage losses as the user does not have any control over the size of the acceptable slippage.

Code location:

contracts/libraries/AMMUtils.sol#L15

contracts/pool/Pool.sol#L532

```
    function getReserve(
        uint256 k,
        uint256 price,
        uint256 pricePrecision
    ) internal pure returns (uint256 reserveA, uint256 reserveB) {
        require(price > 0, "Invalid price");
        require(k > 0, "Invalid k");

        reserveB = Math.sqrt(Math.mulDiv(k, price, pricePrecision));
        reserveA = k / reserveB;
        return (reserveA, reserveB);
    }

    function getMintLpAmount(
        ...
    )
        external
        view
        override
        returns (
            ...
```

```
        )
    {
        ...
        if (indexTotalDeltaWad + stableTotalDeltaWad > 0) {
            ...
            (uint256 reserveA, uint256 reserveB) = AMMUtils.getReserve(
                pair.kOfSwap,
                price,
                AmountMath.PRICE_PRECISION
            );
            if (totalIndexTotalDeltaWad > expectIndexDeltaWad) {
                ...
                slipDeltaWad = swapIndexDeltaWad
                    - AMMUtils.getAmountOut(
                        AmountMath.getIndexAmount(swapIndexDeltaWad, price),
                        reserveA,
                        reserveB
                    );
                ...
            } else if (totalStableTotalDeltaWad > expectStableDeltaWad) {
                ...
                slipDeltaWad = swapStableDeltaWad -
AMMUtils.getAmountOut(swapStableDeltaWad, reserveB, reserveA);
                ...
            }
        }
        ...
    }
```

Solution

It is recommended to set a reasonable value of kOfSwap based on key information such as the price of index tokens, total supply, etc. This value is not as large as possible. Or provide users with a configurable slippage option to stop adding liquidity when the slippage loss exceeds the user's expectation.

Status

Acknowledged; After communication with the project team, they have indicated that they will calculate a reasonable value for kOfSwap and then set it in the contract.

[N13] [Low] Redundant pair token decimal conversion

Category: Arithmetic Accuracy Deviation Vulnerability

Content

In Pool contracts, the getMintLpAmount function calculates the amount of LP tokens to be minted, the amount

of fees, and the amount of money to be deposited after deducting the fees. This function finally converts the amount of LP tokens to be minted to a decimal of 1e18, but in reality, since the pair tokens are inherited from OpenZeppelin's ERC20 tokens, their decimal is fixed at 1e18. Therefore, another decimal would be redundant, and it would consume more gas.

Code location:

contracts/pool/Pool.sol#L582

contracts/pool/PoolToken.sol#L4

```
    function getMintLpAmount(
        ...
    )
        external
        view
        override
        returns (
            ...
        )
    {
        ...
        if (mintDeltaWad > 0) {
            uint8 pairTokenDec = IERC20Metadata(pair.pairToken).decimals();
            mintAmount += AmountMath.getIndexAmount(mintDeltaWad,
lpFairPrice(_pairIndex, price)) / (10 ** (18 - pairTokenDec));
        }
        ...
    }
```

Solution

It is recommended that the decimal conversion operation is no longer performed when performing LP casting amount calculations.

Status

Fixed

[N14] [Medium] Risk of slippage fees being locked in

Category: Design Logic Audit

Content

In a Pool contract, the getMintLpAmount function is used to calculate the amount of LP tokens that can be minted when a user adds liquidity, the fee, and the actual amount of liquidity added. When a user adds tokens that are not expected by the pool, a slippage fee is charged, but this fee is deducted from the user's liquidity amount and not added to indexFeeAmount/stableFeeAmount or recorded separately, which is not in line with the design expectation. This leaves the slippage fee in an unrecorded form in the pool, which means it is indirectly locked in the pool and cannot be withdrawn through normal claim fees.

Code location:

contracts/pool/Pool.sol#L543

contracts/pool/Pool.sol#L553

```
    function getMintLpAmount(
        ...
    )
        external
        view
        override
        returns (
            ...
        )
    {
        ...
            if (totalIndexTotalDeltaWad > expectIndexDeltaWad) {
                ...
                slipAmount = AmountMath.getIndexAmount(slipDeltaWad, price) / (10 **
(18 - indexTokenDec));
                if (slipAmount > 0) {
                    slipToken = pair.indexToken;
                }
                afterFeeIndexAmount -= slipAmount;
            } else if (totalStableTotalDeltaWad > expectStableDeltaWad) {
                ...
                slipDeltaWad = swapStableDeltaWad -
AMMUtils.getAmountOut(swapStableDeltaWad, reserveB, reserveA);
                slipAmount = slipDeltaWad / (10 ** (18 - stableTokenDec));
                if (slipAmount > 0) {
                    slipToken = pair.stableToken;
                }
                afterFeeStableAmount -= slipAmount;
            }
        }
```

```
        ...
    }
```

Solution

It is recommended to add the sliding fee to indexFeeAmount/stableFeeAmount or define a separate attribute in the Vault to record it.

Status

Acknowledged; After communication with the project team, they have stated that this is the intended design, where the slippage fee will ultimately be added directly to the pool.

[N15] [Low] Potential problems with error event logging

Category: Malicious Event Log Audit

Content

In a Pool contract, the AddLiquidity/RemoveLiquidity events are triggered when a user performs a liquidity add/remove; the funder parameter is present in the AddLiquidity event to record `tx.origin`, and the account parameter is present in the RemoveLiquidity event to record `tx.origin`. msg.sender `. However, it is important to note that users may provide liquidity to the pool through other protocols/contracts (e.g. multi-signature wallets), where the funding provider may not be` tx.origin`, and therefore the events logged by AddLiquidity may be incorrect. When a user removes liquidity through a Router contract, the account recorded in the RemoveLiquidity event will always be the Router contract, which is clearly incorrect.

Code location:

contracts/pool/Pool.sol#L678

contracts/pool/Pool.sol#L757

```
    function _addLiquidity(
        ...
    ) private returns (uint256 mintAmount, address slipToken, uint256 slipAmount) {
        ...
        emit AddLiquidity(
            tx.origin,
            recipient,
            _pairIndex,
```

```
            _indexAmount,
            _stableAmount,
            mintAmount,
            indexFeeAmount,
            stableFeeAmount,
            slipToken,
            slipAmount
        );
        ...
    }

    function _removeLiquidity(
        ...
    )
        private
        returns (
            ...
        )
    {
        ...
        emit RemoveLiquidity(
            msg.sender,
            _receiver,
            _pairIndex,
            receiveIndexTokenAmount,
            receiveStableTokenAmount,
            _amount,
            feeAmount
        );
        ...
    }
```

Solution

It is recommended that Pools only allow liquidity add/remove operations through Router contracts. Also pass msg.sender from the Router contract to the Pool as the required parameter for the AddLiquidity/RemoveLiquidity event.

Status

Fixed

[N16] [Suggestion] Discounts are not taken into account when calculating through the getDepositAmount function

Category: Design Logic Audit

Content

In a Pool contract, the getDepositAmount function is used to calculate the amount of index/stable tokens the user needs to deposit based on the amount of LPs the user needs to receive. When it comes to actual deposits, there is a discount for adding liquidity in the direction of helping the pool to rebalance, but no discount is calculated in the getDepositAmount function. Therefore, when adding liquidity using the index/stable token amount calculated by the getDepositAmount function, the amount of LP obtained will be more than expected.

Code location: contracts/pool/Pool.sol#L800

```
    function getDepositAmount(
        uint256 _pairIndex,
        uint256 _lpAmount,
        uint256 price
    ) external view returns (uint256 depositIndexAmount, uint256 depositStableAmount)
{
        ...
    }
```

Solution

It is recommended to consider the presence of discounts in the calculation of the getDepositAmount function.

Status

Acknowledged; After communication with the project team, they have stated that this is the intended design.

[N17] [Suggestion] The getReceivedAmount calculation does not take into account the lack of available balance for both tokens

Category: Design Logic Audit

Content

In the Pool contract, the getReceivedAmount function is used to calculate the amount of index/stable tokens that can be obtained after burning LP. It will check the available amount of index and stable tokens in the pool separately, but will not report an error when the available amount of both tokens is insufficient to pay the user's needs.

Code location: contracts/pool/Pool.sol#L956-L972

```
function getReceivedAmount(
    ...
)
    public
    view
    returns (
        ...
    )
{
    ...
    if (availableIndexToken < receiveIndexTokenAmount) {
        stableTokenAdd = uint256(TokenHelper.convertIndexAmountToStableWithPrice(
            pair,
            int256(receiveIndexTokenAmount - availableIndexToken),
            price));
        receiveIndexTokenAmount = availableIndexToken;
    }

    if (availableStableToken < receiveStableTokenAmount) {
        indexTokenAdd = uint256(TokenHelper.convertStableAmountToIndex(
            pair,
            int256(receiveStableTokenAmount - availableStableToken)
        )).divPrice(price);
        receiveStableTokenAmount = availableStableToken;
    }
    receiveIndexTokenAmount += indexTokenAdd;
    receiveStableTokenAmount += stableTokenAdd;

    return (
        receiveIndexTokenAmount,
        receiveStableTokenAmount,
        feeAmount,
        feeIndexTokenAmount,
        feeStableTokenAmount
    );
}
```

Solution

An error should be returned to alert the user when the available amount of both tokens is insufficient.

Status

Fixed

[N18] [Critical] Incorrect token swap when pool token balance is insufficient

Category: Design Logic Audit

Content

In the Pool contract, the transferTokenOrSwap function is used to transfer tokens in the pool to the specified address. When the balance of one token in the pool is insufficient for payment, another token will be swapped into the required token through the `_swapInUni` function to complete the payment.

Unfortunately, when performing the `_swapInUni` operation, the token with insufficient balance is passed into the `_swapInUni` function as tokenIn, which will make the situation even worse, causing the transferTokenOrSwap function to always fail when the token balance is insufficient. Unable to successfully execute as expected.

Code location:

contracts/pool/Pool.sol#L999

contracts/pool/Pool.sol#L434

```
    function transferTokenOrSwap(
        uint256 pairIndex,
        address token,
        address to,
        uint256 amount
    ) external transferAllowed {
        if (amount == 0) {
            return;
        }
        uint256 bal = IERC20(token).balanceOf(address(this));
        if (bal < amount) {
            _swapInUni(pairIndex, token, amount);
        }
        IERC20(token).safeTransfer(to, amount);
    }

    function _swapInUni(uint256 _pairIndex, address tokenIn, uint256 expectAmountOut)
private {
        ...
    }
```

Solution

It is recommended to pass another token into the `_swapInUni` function as tokenIn to successfully complete the token swap and payment.

Status

Fixed

[N19] [Critical] Incorrect way to get the token price when using the chainlink oracle

Category: Design Logic Audit

Content

In the ChainlinkPriceFeed contract, the getPrimaryPrice function is used to obtain a valid price from Chainlink. Before obtaining the price, it will call the chainlinkFlags contract to check whether the Chainlink service on L2 is available. However, it should be noted that the MYX protocol will be deployed to the Linea and Scroll chains, so it is not reasonable to use `chainlink.flags.arbitrum-seq-offline` for flag acquisition. And currently, the chainlink team has only deployed the chainlinkFlags service in the Arbitrum testnet, and has used the L2 sequencer feeds service in other L2 mainnets instead.

But it should be noted that in the Linea and Scroll chains, chainlink does not seem to have deployed the L2 sequencer feeds service.

Code location: contracts/oracle/ChainlinkPriceFeed.sol#L84-L90

```
    function getPrimaryPrice(address _token) public view returns (uint256) {
        ...

        if (chainlinkFlags != address(0)) {
            bool isRaised =
IChainlinkFlags(chainlinkFlags).getFlag(FLAG_ARBITRUM_SEQ_OFFLINE);
            if (isRaised) {
                // If flag is raised we shouldn't perform any critical operations
                revert("Chainlink feeds are not being updated");
            }
        }
        ...
    }
```

Solution

It is recommended to deprecate the obsolete chainlinkFlags service and use it instead in chains where the L2 sequencer feeds service is available.

Status

Fixed; But it should be noted that the project team uses the same priceAge to check oracle update time for all tokens. For tokens with different update intervals, this may pose potential risks.

[N20] [Low] Potential issues with using latestAnswer() to get prices

Category: Design Logic Audit

Content

In the ChainlinkPriceFeed contract, the getPrimaryPrice function will obtain the price from chainlink's price feed contract through the latestAnswer interface. But the latestAnswer interface only returns the price parameter, without any other parameters to check whether the price is valid. For this reason, chainlink has deprecated this interface in v3.

Reference:

https://docs.chain.link/data-feeds/api-reference/#latestanswer

https://docs.chain.link/data-feeds/#check-the-timestamp-of-the-latest-answer

Code location: contracts/oracle/ChainlinkPriceFeed.sol#L95

```
function getPrimaryPrice(address _token) public view returns (uint256) {
    ...
    int256 _p = priceFeed.latestAnswer();
    ...
}
```

Solution

It is recommended to use the latestRoundData interface instead of latestAnswer, and check the updatedAt parameter to ensure that the price obtained is fresh.

Status

Fixed

[N21] [Critical] Lack of precision conversion for the calculation of ammountInMaximum

Category: Arithmetic Accuracy Deviation Vulnerability

Content

In the Pool contract, The _swapInUni function is used for token exchange, where the required

amountInMaximum parameter is calculated from the incoming expectAmountOut parameter and the price obtained from the price oracle.

The precision of the passed expectAmountOut parameter is 1e18, but the precision of the price obtained from the prediction machine is 1e30. Calculating the price without converting it to the same precision may lead to unintended results (e.g., `(expectAmountOut * 12) / (price * 10)` may be calculated as 0 due to rounding issues).

Code location:

contracts/pool/Pool.sol#443-450

```
    function _swapInUni(uint256 _pairIndex, address tokenIn, uint256 expectAmountOut)
private {
        Pair memory pair = pairs[_pairIndex];
        uint256 price =
IPythOraclePriceFeed(ADDRESS_PROVIDER.priceOracle()).getPrice(pair.indexToken);
        uint256 amountInMaximum;
        address tokenOut;
        if (tokenIn == pair.indexToken) {
            tokenOut = pair.stableToken;
            amountInMaximum = (expectAmountOut * 12) / (price * 10);
        } else if (tokenIn == pair.stableToken) {
            tokenOut = pair.indexToken;
            amountInMaximum = (expectAmountOut * price * 12) / 10;
        }
        ...
    }
```

Solution

It is recommended that the TokenHelper library can be used to unify the precision of expectAmountOut and price before performing the calculation.

Status

Fixed

[N22] [Critical] Incorrect processing of price decimal returned by chainlink

Category: Arithmetic Accuracy Deviation Vulnerability

Content

In the ChainlinkPriceFeed contract, the getPrimaryPrice function is used to get the price from chainlink and convert its decimal to 1e30 via `price * (10 ** (PRICE_DECIMALS - 8))`. Unfortunately, the decimal of chainlink oracles is not all 8. For example, the decimal of the AMPL oracle is 18. Therefore, using the `PRICE_DECIMALS - 8` method for price conversion will result in a return value much larger than expected for an oracle with a decimal of 8.

Code location: contracts/oracle/ChainlinkPriceFeed.sol#L102

```
function getPrimaryPrice(address _token) public view returns (uint256) {
    ...
    return price * (10 ** (PRICE_DECIMALS - 8));
}
```

Solution

It is recommended to obtain the decimal from the oracle and then process it instead of subtracting 8 directly.

Status

Fixed

[N23] [Low] Incomplete judgement in updateTradingFeeConfig function

Category: Others

Content

In the Pool contract, the updateTradingFeeConfig function is used to update some of the configurations for charging fees on trading. The distributorAmount is calculated by subtracting referralsAmount, lpAmount, keeperAmount and stakingAmount from the surplusFee at the time of charging. However, only the value of lpFeeDistributeP + keeperFeeDistributeP + stakingFeeDistributeP is determined to be less than 1e8 when updating the configuration of the fees, and maxReferralsRatio is missing.

This could lead to a situation: when lpFeeDistributeP + keeperFeeDistributeP + stakingFeeDistributeP <= 1e8, but maxReferralsRatio (or referralRate) + lpFeeDistributeP + keeperFeeDistributeP + stakingFeeDistributeP >= 1e8, which results in the charge operation failing due to a computational overflow.

Code Location:

contracts/pool/Pool.sol#211-217

```
    function updateTradingFeeConfig(
        uint256 _pairIndex,
        TradingFeeConfig calldata _tradingFeeConfig
    ) external onlyPoolAdmin {
       ...
        require(
            _tradingFeeConfig.lpFeeDistributeP +
                _tradingFeeConfig.keeperFeeDistributeP +
                _tradingFeeConfig.stakingFeeDistributeP <=
                PrecisionUtils.percentage(),
            "ex"
        );
        tradingFeeConfigs[_pairIndex] = _tradingFeeConfig;
    }
```

contracts/core/FeeCollector.sol#160-164

```
    function distributeTradingFee(
        IPool.Pair memory pair,
        address account,
        address keeper,
        uint256 sizeDelta,
        uint256 tradingFee,
        uint256 vipRate,
        uint256 referralRate
    ) external override onlyPositionManager returns (uint256 lpAmount) {
        ...

        uint256 referralsAmount = surplusFee.mulPercentage(
            Math.min(referralRate, maxReferralsRatio)
        );

        lpAmount = surplusFee.mulPercentage(tradingFeeConfig.lpFeeDistributeP);
        pool.setLPStableProfit(pair.pairIndex, int256(lpAmount));

        uint256 keeperAmount =
surplusFee.mulPercentage(tradingFeeConfig.keeperFeeDistributeP);
        userTradingFee[keeper] += keeperAmount;

        uint256 stakingAmount =
surplusFee.mulPercentage(tradingFeeConfig.stakingFeeDistributeP);
        stakingTradingFee += stakingAmount;

        uint256 distributorAmount = surplusFee -
            referralsAmount -
            lpAmount -
```

```
            keeperAmount -
            stakingAmount;
        ...
    }
```

Solution

It is recommended to add a check for maxReferralsRatio when updating the fee configuration.

Status

Fixed

[N24] [Low] Parameters are missing range checking

Category: Others

Content

1. In the Pool contract, the updatePair function is used to update the parameters of the trading pair. However, in this function only the range of the expectIndexTokenP and addLpFeeP parameters are checked, while the removeLpFeeP, unbalancedDiscountRate and kOfSwap parameters lack range checking. If the admin role is set too high due to a mistake (e.g. removeLpFeeP exceeds 1e8), it may result in unexpected results or errors when trading or changing liquidity.
2. In the FundingRate contract, the updateFundingFeeConfig function is used to update the values of the parameters required to calculate the funding rate. However, in this function only the range of the growthRate, the baseRate and the maxRate parameters are checked, while the fundingInterval parameter lacks the range checking. If the admin role is set too high due to a mistake (e.g. fundingInterval exceeds 86400), it may result in unexpected results or errors when calculating the funding rate.

Code Location: contracts/pool/Pool.sol#159-178

```
    function updatePair(uint256 _pairIndex, Pair calldata _pair) external
onlyPoolAdmin {
        ...
        require(
            _pair.expectIndexTokenP <= PrecisionUtils.percentage() &&
                _pair.addLpFeeP <= PrecisionUtils.percentage(),
            "ex"
```

```
        );

        pair.enable = _pair.enable;
        pair.kOfSwap = _pair.kOfSwap;
        pair.expectIndexTokenP = _pair.expectIndexTokenP;
        pair.maxUnbalancedP = _pair.maxUnbalancedP;
        pair.unbalancedDiscountRate = _pair.unbalancedDiscountRate;
        pair.addLpFeeP = _pair.addLpFeeP;
        pair.removeLpFeeP = _pair.removeLpFeeP;
    }
```

contracts/core/FundingRate.sol#25-37

```
    function updateFundingFeeConfig(
        uint256 _pairIndex,
        FundingFeeConfig calldata _fundingFeeConfig
    ) external onlyPoolAdmin {
        require(
            _fundingFeeConfig.growthRate <= PrecisionUtils.percentage() &&
                _fundingFeeConfig.baseRate <= PrecisionUtils.percentage() &&
                _fundingFeeConfig.maxRate <= PrecisionUtils.percentage(),
            "exceed 100%"
        );

        fundingFeeConfigs[_pairIndex] = _fundingFeeConfig;
    }
```

Solution

1. It is recommended to limit the scope when updating the core parameters of the trading pair (removeLpFeeP, unbalancedDiscountRate, kOfSwap, etc.) to prevent unnecessary errors.
2. It is recommended to limit the scope when updating the fundingInterval parameter of the fundingFeeConfigs to prevent unnecessary errors.

Status

Fixed

[N25] [Medium] Potential risk of not being able to approve swap again

Category: Design Logic Audit

Content

In the Pool contract, the `_swapInUni` function is used to convert index and stable tokens to and from each other. When the subsidy approved by its Pool contract to the spotSwap contract is 0, the Pool will first approve the maximum value to the spotSwap contract, and then perform the swap operation. But when the allowance is consumed to less than the amount required to exchange tokens, the swap operation will not be successful. Unfortunately, since the subsidy amount has not reached 0 at this time, the approval operation cannot be performed again. This results in a swap operation that may never be used again.

The same is true for the swap function in the SpotSwap contract.

Code location:

contracts/pool/Pool.sol#L446-L449

```
    function _swapInUni(uint256 _pairIndex, address tokenIn, uint256 expectAmountOut)
private {
        ...
        if (IERC20(tokenIn).allowance(address(this), spotSwap) == 0) {
            IERC20(tokenIn).safeApprove(spotSwap, type(uint256).max);
        }
        ISpotSwap(spotSwap).swap(tokenIn, tokenOut, amountInMaximum, expectAmountOut);
    }
```

contracts/tools/SpotSwap.sol#L39

```
    function swap(address tokenIn, address tokenOut, uint256 amountIn, uint256
amountOut) external {
        bytes memory path = tokenPath[tokenIn][tokenOut];
        IERC20(tokenIn).safeTransferFrom(msg.sender, address(this), amountIn);
        if (IERC20(tokenIn).allowance(address(this), swapRouter) == 0) {
            IERC20(tokenIn).safeApprove(swapRouter, type(uint256).max);
        }
        ...
    }
```

Solution

It is recommended to trigger the approval operation when checking that the allowance is less than the amountInMaximum.

Status

Fixed

[N26] [High] Failure to check whether the profit is greater than totalAmount results in failure to successfully obtain the totalAmount value of the Pool

Category: Integer Overflow and Underflow Vulnerability

Content

In the Pool contract, the `_getIndexTotalAmount` and `_getStableTotalAmount` functions are used to obtain the value of the index/stable total amount recorded in the pool. They will first obtain the profit of the current pool through getProfit. When profit is less than 0, the index/stable total amount minus profit.abs() will be returned. However, it does not check whether the absolute value of profit is less than the index/stable total amount. When the absolute value of profit is greater than the index/stable total amount, the value of the index/stable total amount cannot be successfully obtained. This will affect the availability of core modules.

Code location:

contracts/pool/Pool.sol#L621

contracts/pool/Pool.sol#L634

```
function _getStableTotalAmount(
    ...
) internal view returns (uint256) {
    int256 profit = getProfit(pair.pairIndex, pair.stableToken, price);
    if (profit < 0) {
        return vault.stableTotalAmount.sub(profit.abs());
    } ...
}

function _getIndexTotalAmount(
    ...
) internal view returns (uint256) {
    int256 profit = getProfit(pair.pairIndex, pair.indexToken, price);
    if (profit < 0) {
        return vault.indexTotalAmount.sub(profit.abs());
    } ...
}
```

Solution

It is recommended that when profit is less than 0, check whether profit.abs() is greater than index/stableTotalAmount. If it is greater, 0 should be returned.

Status

Fixed

[N27] [Suggestion] Potential risks of phishing when using tx.origin for authentication

Category: tx.origin Authentication Vulnerability

Content

In IndexPriceFeed contracts, the onlyKeeper modifier is used to check if `tx.origin` is the keeper role. Note, however, that if the keeper's private key is used for more than just the MYX protocol, this puts it at risk of being phished. An attacker could manipulate the index token price in this way, putting the protocol at risk.

Code location: contracts/oracle/IndexPriceFeed.sol#L24

```
    modifier onlyKeeper() {
        require(IRoleManager(ADDRESS_PROVIDER.roleManager()).isKeeper(tx.origin),
"opk");
        _;
    }
```

Solution

It is recommended to check whether msg.sender is a keeper or Executor contract. Instead of checking tx.origin.

Status

Fixed

[N28] [High] Incorrect getPriceNoOlderThan age

Category: Design Logic Audit

Content

In PythOraclePriceFeed contracts, the getPriceSafely function gets the valid price for age 0 through the getPriceNoOlderThan interface of the Pyth Oracle. This means that the time between the user fetching the price from off-chain and submitting it to the chain to update the price cannot be greater than the average block time.

This is extremely difficult to achieve in L2, where the blocks are very fast, and publishTimes will always be less than block.timestamp after the user fetches a valid price and updates the price, so calling the getPriceNoOlderThan function with an age of 0 will be very difficult to achieve success.

Code location: contracts/oracle/PythOraclePriceFeed.sol#L73

```
    function getPriceSafely(address token) external view override returns (uint256) {
        bytes32 priceId = _getPriceId(token);
        PythStructs.Price memory pythPrice;
        try pyth.getPriceNoOlderThan(priceId, 0) returns (PythStructs.Price memory
_pythPrice) {
            pythPrice = _pythPrice;
        } catch {
            revert("get price failed");
        }
        return _returnPriceWithDecimals(pythPrice);
    }
```

Solution

It is recommended to set an adjustable age to accommodate different market conditions and different blockchain environments.

Status

Fixed

[N29] [Critical] Incorrect decimal handling of Pyth oracle prices

Category: Arithmetic Accuracy Deviation Vulnerability

Content

In the PythOraclePriceFeed contract, the `_returnPriceWithDecimals` function is used to convert the Pyth prophet's price to decimal. it defaults to a decimal of 8 returned by the pyth prophet, so it is processed using `price * (10 ** (PRICE_DECIMALS - 8 ))` for conversion. Unfortunately, the Pyth oracle's decimal is not always 8, e.g. AAPL's decimal is 5, which will cause decimal processing to deviate from the expected result.

Code location: contracts/oracle/PythOraclePriceFeed.sol#L95

```
    function _returnPriceWithDecimals(
        PythStructs.Price memory pythPrice
    ) internal pure returns (uint256) {
```

```
        if (pythPrice.price < 0) {
            return 0;
        }
        return uint256(uint64(pythPrice.price)) * (10 ** (PRICE_DECIMALS - 8));
    }
```

Solution

It is recommended to process the expo parameter returned by the Pyth oracle as decimal.

Status

Fixed

[N30] [Medium] Potential risk that users can still withdraw their margin when the position is liquidated

Category: Design Logic Audit

Content

In the Position library, the validLeverage function is used to check whether the user's position is healthy. When the user closes the position, afterPosition will be reduced to 0, at which point the validLeverage function will return directly without further checking the user's Pnl and leverage status. This allows the user to successfully close the position and get back the order margin when the user is in a liquidable state.

In fact, the user's operation of closing the position will also be performed by the keeper. And when the keeper monitors that the user is in a liquidable state, it will perform the liquidation operation and deduct the user's margin. But judging from the contract, there is no guarantee that the keeper will actually perform the liquidation operation first and ignore the user's closing order. If the keeper executes the user's closing order first, it will cause the user in the liquidated state to unexpectedly successfully withdraw the margin.

Code location: contracts/libraries/Position.sol#L118-L120

```
    function validLeverage(
        ...
    ) internal view returns (uint256, uint256) {
        ...

        // close position
        if (afterPosition == 0) {
            return (0, 0);
        }
```

```
        ...
    }
```

Solution

It is recommended that during the validLeverage check process, when the afterPosition is 0, still perform subsequent checks instead of returning directly.

Status

Fixed

[N31] [Critical] Incorrect slippage check

Category: Design Logic Audit

Content

In the ValidationHelper library, the validatePriceTriggered function is used to check whether the current price and the user's order price are reasonable. When the order type is MARKET, it will also be checked whether the price deviation is within the allowed range of slippage. Theoretically, when the maxSlippage passed in by the user is 0, it means that the user does not accept any price deviation. But in the validatePriceTriggered function, when maxSlippage is 0, regardless of whether the price meets expectations, the check will be passed directly. This may cause users to suffer unnecessary losses.

Code location: contracts/helpers/ValidationHelper.sol#L40

```
    function validatePriceTriggered(
        ...
        uint256 maxSlippage
    ) internal pure {
        if (tradeType == TradingTypes.TradeType.MARKET) {
            bool valid = currentPrice >=
                orderPrice.mulPercentage(PrecisionUtils.percentage() - maxSlippage) &&
                currentPrice <= orderPrice.mulPercentage(PrecisionUtils.percentage() +
maxSlippage);
            require(maxSlippage == 0 || valid, "exceeds max slippage");
        } else if (tradeType == TradingTypes.TradeType.LIMIT) {
            ...
    }
```

Solution

It is recommended to use percentage instead of 0 check. When the maxSlippage passed by the user is a percentage, it means that it accepts any price deviation.

Status

Acknowledged; After communicating with the project team, the team has indicated that this is the expected design. The protocol front end does not allow users to set a 0 slippage. If the maxSlippage parameter value passed in by the user is 0, it means that no slippage check is performed.

[N32] [Critical] Missing decimal conversion for `exposureAmountChecker` operation

Category: Arithmetic Accuracy Deviation Vulnerability

Content

In the TradingHelper library, the exposureAmountChecker function is used to calculate the position size available in the current pool, and the decimal of its return value is unified to the decimal of index. But when exposedPositions is less than 0 and isLong is false, it does not perform decimal conversion and directly uses availableStable to participate in the calculation. This will result in calculation results that are not as expected.

Code location: contracts/helpers/TradingHelper.sol#L63-L66

```
    function exposureAmountChecker(
        ...
    ) internal view returns (uint256 executionSize) {
        ...
            } else {
                uint256 availableStable = lpVault.stableTotalAmount -
lpVault.stableReservedAmount;
                if (executionSize > availableStable.divPrice(executionPrice)) {
                    executionSize = availableStable.divPrice(executionPrice);
                }
            }
        ...
    }
```

Solution

It is recommended to use convertStableAmountToIndex to convert the decimal of availableStable first.

Status

Fixed

[N33] [Suggestion] The user's order was not canceled as designed

Category: Design Logic Audit

Content

In the ExecutionLogic contract, the keeper role can execute orders in batches through executeIncreaseLimitOrders/executeDecreaseLimitOrders. According to design expectations, when the order execution fails, the protocol will cancel the user's abnormal order. However, in the executeIncreaseLimitOrders and executeDecreaseLimitOrders functions, if the order execution fails, it will only return a predetermined error and will not cancel the order. This is different from design expectations.

And the keeper role can directly call the executeIncreaseOrder and executeDecreaseOrder functions to execute the specified order. These functions do not implement the logic of canceling the order in case of execution failure.

Code location:

contracts/core/logic/ExecutionLogic.sol#L114

contracts/core/logic/ExecutionLogic.sol#L120

contracts/core/logic/ExecutionLogic.sol#L334

contracts/core/logic/ExecutionLogic.sol#L340

```
function executeIncreaseLimitOrders(
    address keeper,
    ExecuteOrder[] memory orders
) external override onlyExecutorOrKeeper {
    for (uint256 i = 0; i < orders.length; i++) {
        ExecuteOrder memory order = orders[i];
        try
            ...
        {} catch Error(string memory reason) {
            emit ExecuteOrderError(order.orderId, reason);
        }
    }
}

function executeIncreaseOrder(
```

```
    ...
) external override onlyExecutorOrKeeper {
    ...
}

function executeDecreaseLimitOrders(
    address keeper,
    ExecuteOrder[] memory orders
) external override onlyExecutorOrKeeper {
    for (uint256 i = 0; i < orders.length; i++) {
        ExecuteOrder memory order = orders[i];
        try
            ...
        {} catch Error(string memory reason) {
            emit ExecuteOrderError(order.orderId, reason);
        }
    }
}

function executeDecreaseOrder(
    ...
) external override onlyExecutorOrKeeper {
    ...
}
```

Solution

It is recommended to clarify the design expectations and handle them in accordance with the design expectations if the execution of the order fails.

Status

Fixed

[N34] [Medium] Potential risks of Keeper role doing evil

Category: Authority Control Vulnerability Audit

Content

1. Keepers can execute user orders by calling the executeIncreaseMarketOrders, executeIncreaseLimitOrders, executeIncreaseOrder, executeDecreaseMarketOrders, executeDecreaseLimitOrders, executeDecreaseOrder, or executeADLAndDecreaseOrder functions in the ExecutionLogic contract, either through the Executor contract or directly. Keepers can pass in any value for the discount level and commission ratio, meaning that any keepers can use different discount

levels and commission ratios for different orders as they see fit. While there is a maxReferralsRatio limit, it is clear that keepers will always execute orders in the way that is most beneficial to them.

2. In the Executor contract, the keeper role updates the oracle price before executing an order through the setPrices function to ensure that the price obtained when executing the order is the latest valid price. The setPrices function separately updates the Pyth oracle price and the index token oracle price and checks whether the price updated by the keeper is reliable through the getValidPrice function when executing the order.

   In theory, although the keeper role can manipulate the index token oracle price, it cannot manipulate the Pyth oracle price, and the getValidPrice ultimately returns the on-chain oracle price. This will minimize the risk of keeper malicious behavior to the greatest extent. Unfortunately, the Pyth oracle accepts the update of old prices. In other words, the keeper updating the Pyth oracle with the old price will not be reverted. Although the price cannot be successfully updated, the transaction will not fail. This also leaves room for the keeper to do evil. The keeper can pass in the old price to prevent the Pyth oracle from updating to the latest price and use this old price to update the index token oracle. This will allow it to pass the getValidPrice check, but the price is not the latest.

   Although the price of the Pyth oracle needs to be checked by age, it can still cause large-scale arbitrage and cause the pool to suffer losses in the case of market volatility.

3. In the executeDecreaseOrder() function of the ExecutionLogic contract, keepers can pass arbitrary values for the `isSystem` and `onlyOnce` parameters. However, theoretically, these parameters are used when the contract calls executeDecreaseOrder() itself. In other words, these parameters should not be allowed to be passed by keepers, but should be passed by the contract itself according to the current order status. This could lead to the risk of keeper misconduct. For example, a keeper could pass `isSystem` as true, which would allow the executionSize to exceed the maxTradeAmount. This would expose LPs to unnecessary risk.

4. In the Executor contract, all functions do not have reentrancy checks. In fact, there is a reentrancy risk when users call functions. For example, when a keeper executes the executeIncreaseOrder operation, it may reenter the `_handleCollateral` function in `PositionManager::increasePosition` to

repeatedly execute the same market order. Malicious keepers can exploit this method to profit to cover their losses from being penalized.

Code location:

contracts/core/logic/ExecutionLogic.sol#L86-L87

contracts/core/logic/ExecutionLogic.sol#L111-L112

contracts/core/logic/ExecutionLogic.sol#L124-L125

contracts/core/logic/ExecutionLogic.sol#L300-L301

contracts/core/logic/ExecutionLogic.sol#L328-L329

contracts/core/logic/ExecutionLogic.sol#L344-L345

contracts/core/logic/ExecutionLogic.sol#L629-L630

```
function executeIncreaseOrder(
    ...
    uint8 level,
    uint256 commissionRatio
) external override onlyExecutorOrKeeper {
    ...
}

function executeDecreaseOrder(
    ...
    uint8 level,
    uint256 commissionRatio,
    ...
) external override onlyExecutorOrKeeper {
    ...
}

function executeADLAndDecreaseOrder(
    ...
    uint8 _level,
    uint256 _commissionRatio
) external override onlyExecutorOrKeeper {
    ...
}
```

contracts/core/Executor.sol#L135-L148

contracts/helpers/TradingHelper.sol#L26-L27

```
    function setPrices(
        address[] memory _tokens,
        uint256[] memory _prices,
        bytes[] memory updateData
    ) external payable {
        require(msg.sender == address(this), "internal");

        IIndexPriceFeed(ADDRESS_PROVIDER.indexPriceOracle()).updatePrice(_tokens,
_prices);

        IPythOraclePriceFeed(ADDRESS_PROVIDER.priceOracle()).updatePrice{value:
msg.value}(
            _tokens,
            updateData
        );
    }

    function getValidPrice(
        IAddressesProvider addressesProvider,
        address token,
        IPool.TradingConfig memory tradingConfig
    ) internal view returns (uint256) {
        ...
        require(diffP <= tradingConfig.maxPriceDeviationP, "exceed max price
deviation");
        return oraclePrice;
    }
```

contracts/core/logic/ExecutionLogic.sol#L346-L348

```
    function executeDecreaseOrder(
        ...
        bool isSystem,
        uint256 executionSize,
        bool onlyOnce
    ) external override onlyExecutorOrKeeper {
        ...
    }
```

Solution

The protocol mitigates the risk of keeper misconduct by requiring keepers to stake assets and impose penalties for misconduct. However, it is important to note that if the keeper's profit exceeds the cost of the penalty, they will be more likely to choose to misconduct. Therefore, it is necessary to require keepers to undergo KYC

verification off-chain, and to implement strict checks in smart contracts, in order to mitigate the risk of keeper misconduct to the greatest extent possible.

1. Unless otherwise intended, it is recommended to divide keepers into fixed levels based on their stake amount, rather than allowing keepers to choose freely.
2. On the one hand, the arbitrage space can be limited by setting a smaller Pyth age. On the other hand, the project team can sign the index token price and verify the signature in the index token oracle to ensure that the index token price updated by the keeper is expected.
3. It is recommended to restrict the executeDecreaseOrder() function to only allow the current contract to call it.
4. It is recommended to add the `nonReentrant` modifier to the Executor contract to maximize the impact of curbing keeper misconduct.

Status

Fixed

[N35] [Information] Market orders that are not fully executed still use full collateral

Category: Others

Content

In the ExecutionLogic contract, the executeIncreaseOrder function is used to execute a user's increase order. When executing a MARKET order, it will use the full collateral provided by the user. However, it is important to note that the execution size is not fully used, and this depends on the return value of the exposureAmountChecker function. When the return value of the exposureAmountChecker function is less than the execution size of the user's order, the user's order will be partially executed. Even if it is partially executed, the collateral used is still the full amount.

Code location: contracts/core/logic/ExecutionLogic.sol#L197-L206

```
    function executeIncreaseOrder(
        ...
    ) external override onlyExecutorOrKeeper {
        ...
        if (order.collateral > 0) {
```

```
            collateral = order.executedSize == 0 || order.tradeType ==
TradingTypes.TradeType.MARKET
                ? order.collateral
                : int256(0);
        } else {
            collateral = order.executedSize + executionSize >= order.sizeAmount ||
                order.tradeType == TradingTypes.TradeType.MARKET
                ? order.collateral
                : int256(0);
        }
        ...
    }
```

Solution

If it is not the expected design, it is recommended to return the remaining margin in proportion to the execution size.

Status

Acknowledged

[N36] [Suggestion] Inaccurate comparison operator between executionSize and sizeAmount

Category: Others

Content

In the ExecutionLogic contract, the executed size is used to compare with the order's sizeAmount to check if the order is fully executed. However, it is important to note that the executed size will never be greater than the sizeAmount. Therefore, the use of the `>=` operator in the contract is not appropriate.

Code location:

contracts/core/logic/ExecutionLogic.sol#L202

contracts/core/logic/ExecutionLogic.sol#L250

contracts/core/logic/ExecutionLogic.sol#L485

contracts/core/logic/ExecutionLogic.sol#L516

```
    function executeIncreaseOrder(
        ...
    ) external override onlyExecutorOrKeeper {
        ...
        if (order.collateral > 0) {
```

```
            ...
        } else {
            collateral = order.executedSize + executionSize >= order.sizeAmount ||
                order.tradeType == TradingTypes.TradeType.MARKET
                ? order.collateral
                : int256(0);
        }
        ...
        if (
            order.tradeType == TradingTypes.TradeType.MARKET ||
            order.executedSize >= order.sizeAmount
        ) {
            ...
          }
    }

    function _executeDecreaseOrder(
        ...
    ) internal {
        ...
        if (order.collateral > 0) {
            collateral = order.executedSize == 0 || onlyOnce ? order.collateral :
int256(0);
        } else {
            collateral = order.executedSize + executionSize >= order.sizeAmount ||
onlyOnce
                ? order.collateral
                : int256(0);
        }
        ...
        if (onlyOnce || order.executedSize >= order.sizeAmount ||
position.positionAmount == 0) {
            ...
        }
        ...
    }
```

Solution

It is recommended to use the `==` operator instead of the `>=` operator to compare executedSize and sizeAmount.

Status

Acknowledged

[N37] [High] Cancelling all orders may lead to a denial-of-service (DoS) attack

Category: Denial of Service Vulnerability

Content

In the OrderManager contract, the cancelAllPositionOrders() function is used to cancel all orders for a specified user. It iterates through the user's orders using a while loop and then cancels them one by one using the _cancelOrder() function. However, the gas limit of a block on a blockchain is finite. If the gas used to execute the while loop exceeds the block gas limit, the transaction will fail.

Although the keeper can cancel a specified order using the cancelOrder() function, the _executeDecreaseOrder() function in the ExecutionLogic contract will call the cancelAllPositionOrders() function to cancel all orders for a user when the user's remaining position is 0. The liquidationPosition() function in the LiquidationLogic contract will also cancel all orders for the liquidated user using the cancelAllPositionOrders() function. Therefore, if a user is aware that they are about to be liquidated, they can create a large number of orders in advance to prevent the keeper from successfully liquidating them.

Code location:

contracts/core/OrderManager.sol#L264

contracts/core/logic/ExecutionLogic.sol#L400

contracts/core/logic/LiquidationLogic.sol#L102

```
    function cancelAllPositionOrders(
        address account,
        uint256 pairIndex,
        bool isLong
    ) external onlyExecutor {
        ValidationHelper.validateAccountBlacklist(ADDRESS_PROVIDER, account);

        bytes32 key = PositionKey.getPositionKey(account, pairIndex, isLong);

        while (positionOrders[key].length > 0) {
            ...
        }
    }

    function _executeDecreaseOrder(
        ...
    ) internal {
        ...
        if (position.positionAmount == 0) {
```

```
            orderManager.cancelAllPositionOrders(order.account, order.pairIndex,
order.isLong);
            return;
        }
        ...
    }

    function liquidationPosition(
        ...
    ) external override onlyExecutorOrKeeper {
        ...
        orderManager.cancelAllPositionOrders(position.account, position.pairIndex,
position.isLong);
        ...
    }
```

Solution

It is recommended to limit the number of orders that users can create to mitigate this risk.

Status

Fixed

[N38] [Critical] It is not possible to execute ADL for orders in liquidation status

Category: Design Logic Audit

Content

In the `_executeDecreaseOrder()` function of the ExecutionLogic contract, it checks the health of the user's position using the validLeverage() function when the user decreases their position. If the user's position is in liquidation status, the user will not be able to successfully decrease their position. This is a good way to ensure the health of the protocol. Unfortunately, when the protocol believes that a liquidated user needs to execute an ADL operation, the protocol will not be able to successfully execute the decrease position operation due to the validLeverage() check. This is not in line with the expected design and could put the protocol at risk of bankruptcy.

Code location: contracts/core/logic/ExecutionLogic.sol#L449

```
    function _executeDecreaseOrder(
        ...
    ) internal {
        ...
```

```
        // check position and leverage
        position.validLeverage(
            pair,
            executionPrice,
            order.collateral,
            executionSize,
            false,
            tradingConfig.maxLeverage,
            tradingConfig.maxPositionAmount,
            false
        );
        ...
    }
```

Solution

It is recommended to change the simpleVerify parameter passed to the validLeverage() function in the `_executeDecreaseOrder()` function to isSystem. This will allow for more flexible control of the validLeverage() check during ADL. It is important to note that this will increase the risk of keeper misconduct. It is recommended to implement measures to mitigate this risk as proposed in N34.

Status

Fixed

[N39] [Low] During ADL, the order's needADL status was not checked

Category: Design Logic Audit

Content

The executeADLAndDecreaseOrder function in the ExecutionLogic contract is used to execute ADL and decrease order. However, it does not check the order's needADL status when executing the order. This could lead to the keeper erroneously executing orders that do not need ADL.

Code location: contracts/core/logic/ExecutionLogic.sol#L632-L635

```
    function executeADLAndDecreaseOrder(
        address keeper,
        ExecutePosition[] memory executePositions,
        uint256 _orderId,
        TradingTypes.TradeType _tradeType,
        uint8 _level,
        uint256 _commissionRatio
    ) external override onlyExecutorOrKeeper {
```

```
        TradingTypes.DecreasePositionOrder memory order =
orderManager.getDecreaseOrder(
            _orderId,
            _tradeType
        );
        ...
    }
```

Solution

It is recommended to check the order's needADL status before executing the executeADLAndDecreaseOrder operation.

Status

Acknowledged; After communicating with the project team, the project team stated that this was the expected design. If no ADL operation is performed, the protocol will perform normal position reduction operations.

[N40] [Low] Not checking the difference between needADLAmount and executeTotalAmount causes gas waste

Category: Gas Optimization Audit

Content

In the executeADLAndDecreaseOrder function of the ExecutionLogic contract, when it executes the ADL operation for the executePositions list, it does not check whether `needADLAmount - executeTotalAmount` is 0. This will cause the for loop to continue operating even when needADLAmount is already equal to executeTotalAmount, which will consume unnecessary gas.

Code location: contracts/core/logic/ExecutionLogic.sol#L698

```
    function executeADLAndDecreaseOrder(
        ...
    ) external override onlyExecutorOrKeeper {
        ...
        for (uint256 i = 0; i < adlPositions.length; i++) {
            ...
            if (position.positionAmount >= needADLAmount - executeTotalAmount) {
                adlExecutionSize = needADLAmount - executeTotalAmount;
            } else {
                adlExecutionSize = position.positionAmount;
            }
            ...
```

```
        }
        ...
    }
```

Solution

It is recommended to check whether `needADLAmount - executeTotalAmount` is 0 in the for loop. When the difference is 0, the loop can be stopped to save gas.

Status

Fixed

[N41] [Medium] Bypass price update and execute order

Category: Design Logic Audit

Content

Users interact with the protocol through the Router contract, and Keepers execute orders through the Executor contract. When these operations depend on fresh prices, the contract will first perform the updatePrice operation to ensure that the price obtained from the oracle is the latest. However, it is important to note that since the underlying contracts also allow users or Keepers to call directly, users and Keepers can bypass price updates and interact with the protocol directly, which is not in line with the design intent. Users or Keepers can use this method to arbitrage.

For example, if a user is in a liquidation state according to the latest price, but the price has not yet been updated and the user is not yet in a liquidation state according to the current price. Therefore, the user can directly call the PositionManager.adjustCollateral function to modify the collateral, instead of the expected Router.setPriceAndAdjustCollateral function. The same is true for Keepers. The Executor contract needs to perform the setPrices operation, but Keepers can also directly call the ExecutionLogic and LiquidationLogic contracts to bypass price updates.

For users, the risk of bypassing price updates has been listed in N7. For Keepers, although they do not intentionally commit malicious acts, their direct interaction with logical contracts without going through the Executor contract is also not in line with market expectations. Their use of stale prices to execute orders may cause users unexpected losses.

Solution

It is recommended that the protocol be redesigned to have a single entry point, with all users interacting with the protocol through the Router contract and all keepers interacting with the protocol through the Executor contract.

Status

Fixed

[N42] [Low] Unsafe price fetching during `_swapInUni` operation

Category: Design Logic Audit

Content

In the Pool contract, the `_swapInUni` function is used to swap stable/index tokens. It retrieves the price from the oracle through the getPrice interface and uses it to calculate slippage. However, the price retrieved by this interface may not be the latest, which may cause slippage calculation to be biased.

Code location: contracts/pool/Pool.sol#L436

```
    function _swapInUni(uint256 _pairIndex, address tokenIn, uint256 expectAmountOut)
private {
        Pair memory pair = pairs[_pairIndex];
        uint256 price =
IPythOraclePriceFeed(ADDRESS_PROVIDER.priceOracle()).getPrice(pair.indexToken);
        ...
    }
```

Solution

It is recommended to use the getPriceSafely interface to obtain safe prices.

Status

Acknowledged; After communicating with the project team, the team has indicated that this is the expected design.

[N43] [Suggestion] Redundant claim interface

Category: Gas Optimization Audit

Content

In the FeeCollector contract, keepers can claim trading fees through the claimKeeperTradingFee function, while users can claim trading fees through the claimUserTradingFee function. However, the claimKeeperTradingFee function and the claimUserTradingFee function both internally call the same _claimUserTradingFee function to claim fees. Therefore, having two functions in the interface consumes more gas.

Code location: contracts/core/FeeCollector.sol#L123-L129

```
    function claimKeeperTradingFee() external override nonReentrant returns (uint256)
{
        return _claimUserTradingFee();
    }

    function claimUserTradingFee() external override nonReentrant returns (uint256) {
        return _claimUserTradingFee();
    }
```

Solution

It is recommended to merge the claimUserTradingFee and claimKeeperTradingFee functions into a single function to save gas.

Status

Fixed

[N44] [Low] Fee collection does not follow the Checks-Effects-Interactions (CEI) principle

Category: Reentrancy Vulnerability

Content

Users can claim stakingTradingFee through the claimStakingTradingFee function of the FeeCollector contract from the StakingPool contract's claimReward function. PoolAdmin can claim treasuryFee through the claimTreasuryFee function. However, it is important to note that in the fee collection process, the transfer operation is performed first and then the fee record is set to 0, which does not comply with the Checks-Effects-Interactions (CEI) principle. If the transfer function of pledgeAddress has a callback function, it will lead to a re-entry risk, and users can re-enter the claimReward function of the StakingPool to deplete the funds of the Pool. For pledgeAddress on Linea/scroll, this may not pose a risk, because mainstream stablecoins generally do not

support callbacks. However, for some chains (such as xDAI), transfer callbacks are supported by default.

Therefore, when the protocol is deployed to multiple chains, it may be subject to this risk.

Code location:

contracts/core/FeeCollector.sol#L106-L107

contracts/core/FeeCollector.sol#L116-L117

```
    function claimStakingTradingFee() external override onlyStakingPool returns
(uint256) {
        ...
        if (claimableStakingTradingFee > 0) {
            pool.transferTokenTo(pledgeAddress, msg.sender,
claimableStakingTradingFee);
            stakingTradingFee = 0;
        }
        ...
    }

    function claimTreasuryFee() external override onlyPoolAdmin returns (uint256) {
        uint256 claimableTreasuryFee = treasuryFee;
        if (claimableTreasuryFee > 0) {
            pool.transferTokenTo(pledgeAddress, msg.sender, claimableTreasuryFee);
            treasuryFee = 0;
        }
        ...
    }
```

Solution

It is recommended to follow the CEI principle and set the fee record to 0 before performing the transfer operation.

Status

Fixed

[N45] [Information] Liquidity calculation in funding fees does not match expected design

Category: Others

Content

In the FundingRate contract, the getFundingRate function is used to calculate the current funding rate of the

protocol. The l parameter represents the total liquidity of the current pool, denominated in index tokens. This is different from the expected design, which specifies that l should be denominated in USD.

Code location: contracts/core/FundingRate.sol#L59-L62

```
    function getFundingRate(
        ...
    ) public view override returns (int256 fundingRate) {
        ...
        uint256 l = vault.indexTotalAmount
            + uint256(TokenHelper.convertStableAmountToIndex(
                pair, int256(vault.stableTotalAmount))
            ).divPrice(price);
        ...
    }
```

Solution

It is recommended to clarify the design expectations.

Status

Fixed

[N46] [Low] Incorrect old addressPositionManager retrieval

Category: Design Logic Audit

Content

In the RiskReserve contract, the updatePositionManagerAddress function is used to update the addressPositionManager parameter. Before updating, it gets the old addressPositionManager parameter in order to use an event to record the change of the new and old parameter values. However, it incorrectly gets the value of addressDao when getting the old value.

Code location: contracts/core/RiskReserve.sol#L46

```
    function updatePositionManagerAddress(address newAddress) external override
onlyPoolAdmin {
        address oldAddress = addressDao;
        addressPositionManager = newAddress;
        emit UpdatedPositionManagerAddress(msg.sender, oldAddress,
```

```
addressPositionManager);
    }
```

Solution

It is recommended to use the addressPositionManager parameter instead of addressDao.

Status

Fixed

[N47] [Suggestion] Optimizable `msg.sender` checks

Category: Gas Optimization Audit

Content

In the OrderManager contract, the createOrder function is used to create an order. It first checks the caller to ensure that only trusted contracts are allowed to call it. However, the check is the same as the onlyExecutorAndRouter modifier, so it can be simplified by using the modifier instead.

Code location:

contracts/core/OrderManager.sol#L76

contracts/core/OrderManager.sol#L93-L98

```
    modifier onlyExecutorAndRouter() {
        require(
            msg.sender == router ||
                msg.sender == ADDRESS_PROVIDER.executionLogic() ||
                msg.sender == ADDRESS_PROVIDER.liquidationLogic(),
            "no access"
        );
        _;
    }

    function createOrder(
        TradingTypes.CreateOrderRequest calldata request
    ) public returns (uint256 orderId) {
        require(
            msg.sender == ADDRESS_PROVIDER.executionLogic() ||
                msg.sender == ADDRESS_PROVIDER.liquidationLogic() ||
                msg.sender == router,
            "onlyExecutor&Router"
        );
```

```
        ...
    }
```

Solution

It is recommended to use the `onlyExecutorAndRouter` modifier to check `msg.sender` to simplify the createOrder function.

Status

Fixed

[N48] [Low] Adding liquidity without checking if the pair is enabled

Category: Design Logic Audit

Content

In the Pool contract, the admin role can modify the enable state of a pair using the updatePair function. The protocol checks the `pair.enable` state when users create orders, but users do not check it when adding liquidity. If a trading pair experiences an emergency, the admin can set its enable to false to disable it, but users can still add liquidity to the affected pair.

Code location: contracts/pool/Pool.sol#L171

```
    function updatePair(uint256 _pairIndex, Pair calldata _pair) external
onlyPoolAdmin {
        ...
        pair.enable = _pair.enable;
        ...
    }
```

Solution

It is recommended to check the enable state of a pair when users add liquidity.

Status

Fixed

[N49] [Suggestion] Redundant data type conversion

Category: Gas Optimization Audit

Content

In the OrderManager contract's createOrder function, users can create user orders. When creating a decrease order for users through the `_saveDecreaseOrder` function, the function first uses `abs()` to convert `request.sizeAmount` to a positive number, with a data type of uint256. However, the function then uses `uint256()` to convert again, which is redundant.

Code location: contracts/core/OrderManager.sol#L201

```
function createOrder(
    TradingTypes.CreateOrderRequest calldata request
) public returns (uint256 orderId) {
    ...
    if (request.sizeAmount > 0) {
        ...
    } else if (request.sizeAmount < 0) {
        return
            _saveDecreaseOrder(
                TradingTypes.DecreasePositionRequest({
                    account: account,
                    pairIndex: request.pairIndex,
                    tradeType: request.tradeType,
                    collateral: request.collateral,
                    triggerPrice: request.openPrice,
                    sizeAmount: uint256(request.sizeAmount.abs()),
                    isLong: request.isLong,
                    maxSlippage: request.maxSlippage
                })
            );
    ...
}
```

Solution

It is recommended to remove the uint256().

Status

Fixed

[N50] [Information] Potential issues with long-short market reversals

Category: Others

Content

In the PositionManager contract, the `_settleLPPosition` function is used to settle LP profits. When the

market reverses from a long position to a short position, the pool subtracts the long position (decreaseLong). However, when the market reverses from a short position to a long position, it does not subtract decreaseShort but all stable reserves in the vault (lpVault.stableReservedAmount). This means that there are two different logics for decrease the reserve amount in a long-short reversal. If this is not an intentional design, it could lead to the accidental depletion of the stableReservedAmount in the vault.

Code location:

contracts/core/PositionManager.sol#L457

contracts/core/PositionManager.sol#L496-L508

```
function _settleLPPosition(
    ...
) internal {
    ...
    if (currentPositionStatus == PositionStatus.NetLong) {
        if (isAddPosition) {
            ...
        } else {
            ...
            pool.decreaseReserveAmount(_pairIndex, decreaseLong, 0);
            if (increaseShort > 0) {
                pool.increaseReserveAmount(
                    _pairIndex,
                    0,
                    uint256(
                        TokenHelper.convertIndexAmountToStableWithPrice(
                            pair,
                            int256(increaseShort),
                            _price
                        )
                    )
                );
                pool.updateAveragePrice(_pairIndex, _price);
            }

            _calLpProfit(pair, false, decreaseLong, _price);
        }
    } else if (currentPositionStatus == PositionStatus.NetShort) {
        if (isAddPosition) {
            ...
        } else {
            ...
```

```
                pool.decreaseReserveAmount(
                    _pairIndex,
                    0,
                    nextExposureAmountChecker >= 0
                        ? lpVault.stableReservedAmount
                        : uint256(
                            TokenHelper.convertIndexAmountToStableWithPrice(
                                pair,
                                int256(decreaseShort),
                                lpVault.averagePrice
                            )
                        )
                );
            ...
        ...
    }
```

Solution

If this is not an intentional design, it is recommended that the market adopt the same decreaseReserveAmount logic when the market reverses from a long position to a short position and when it reverses from a short position to a long position.

Status

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

[N51] [Critical] The protocol cannot effectively conduct ADL due to defects in needADL

Category: Design Logic Audit

Content

In the PositionManager contract, the `needADL` function is used to check whether the current protocol needs ADL. It calculates the ADL state by checking the total amount of stable/index tokens in the pool and the current exposed position. However, it does not consider the current available liquidity of the protocol, which may lead to the situation where the protocol cannot trigger the ADL state even if the current liquidity is insufficient. This could ultimately lead to the bankruptcy of the protocol.

For example, the current protocol has far more long positions than short positions, and the difference between the total amount and the reserve amount is already equal to 0. However, when a user closes a long position, the available calculated by `needADL` is still greater than 0, which prevents the protocol from entering the ADL state.

Code location: contracts/core/PositionManager.sol#L729-L762

```
    function needADL(
        uint256 pairIndex,
        bool isLong,
        uint256 executionSize,
        uint256 executionPrice
    ) external view returns (bool need, uint256 needADLAmount) {
        ...
    }
```

Solution

It is recommended to redesign the `needADL` function to perfectly fulfill the function of checking the ADL state.

Status

Fixed

[N52] [Suggestion] Optimized collateral calculation when a user is liquidated

Category: Design Logic Audit

Content

In the decreasePosition function of the PositionManager contract, when a user's collateral cannot cover their losses, the protocol will deduct the corresponding funds from the risk reserve. At the same time, the user's collateral will be updated through the following way:

`position.collateral.sub(int256(totalSettlementAmount + int256(subsidy)).abs())`. This is redundant, and the user's collateral can be directly set to 0 at this time to save gas and avoid precision errors caused by multiple calculations.

Code location: contracts/core/PositionManager.sol#L221-L223

```
    function decreasePosition(
        ...
    ) external onlyExecutor returns (uint256 tradingFee, int256 fundingFee, int256
pnl) {
        ...
        if (totalSettlementAmount >= 0) {
            position.collateral =
position.collateral.add(totalSettlementAmount.abs());
        } else {
```

```
            ...
            } else {
                if (position.positionAmount == 0) {
                    uint256 subsidy = totalSettlementAmount.abs() -
position.collateral;
                    riskReserve.decrease(pair.stableToken, subsidy);
                    position.collateral = position.collateral.sub(
                        int256(totalSettlementAmount + int256(subsidy)).abs()
                    );
                } else {
                    ...
                }
            }
        }
        ...
    }
```

Solution

It is recommended to directly set the user's `position.collateral` to 0 when the user is liquidated, in order to save gas and avoid precision errors caused by multiple calculations.

Status

Fixed

[N53] [Medium] Potential overflow risks caused by type conversion

Category: Arithmetic Accuracy Deviation Vulnerability

Content

In the Router contract, users can create orders through the `createIncreaseOrderWithTpSl`, `createIncreaseOrder`, and `createDecreaseOrder` functions. These functions convert the `request.sizeAmount` parameter to the int256 type. However, it is important to note that the original data type of `request.sizeAmount` is uint256. If the user passes in a sizeAmount that is greater than uint128, this will cause the protocol to overflow when converting the data to `int256(request.sizeAmount)`. This prevents the order from being created with the expected sizeAmount.

Although the sizeAmount of mainstream tokens generally does not exceed uint128, if the protocol lists some tokens with a huge total supply, then this risk may occur.

Code location:

contracts/core/Router.sol#L128

contracts/core/Router.sol#L169

contracts/core/Router.sol#L190

contracts/core/Router.sol#L212

contracts/libraries/TradingTypes.sol#L38

contracts/libraries/TradingTypes.sol#L49

contracts/libraries/TradingTypes.sol#L63

```
struct IncreasePositionRequest {
    ...
    uint256 sizeAmount; // size
    ...
}

struct IncreasePositionWithTpSlRequest {
    ...
    uint256 sizeAmount; // size
    ...
}

struct DecreasePositionRequest {
    ...
    uint256 sizeAmount; // size
    ...
}

function createIncreaseOrderWithTpSl(
    TradingTypes.IncreasePositionWithTpSlRequest memory request
) external whenNotPaused nonReentrant returns (uint256 orderId) {
    ...
    orderId = orderManager.createOrder(
        TradingTypes.CreateOrderRequest({
            ...
            sizeAmount: int256(request.sizeAmount),
            ...
        })
    );
    ...
}
```

Solution

It is recommended to modify the sizeAmount field in the `IncreasePositionWithTpSlRequest`, `IncreasePositionRequest`, and `DecreasePositionRequest` structs to the `uint128` type. Alternatively, the protocol can check for overflows when creating orders.

Status

Fixed

[N54] [Suggestion] Unvalidated TP/SL prices

Category: Design Logic Audit

Content

In the Router contract, users can add TP/SL to their orders through the addOrderTpSl function, or quickly create a TP/SL order through the createTpSl function. However, it does not check whether the TP/SL matches the order price. For long positions, the TP price should be greater than the order price, and the SL price should be less than the order price. The opposite is true for short positions. If a TP/SL order is created with an unexpected price, it may prevent the order from executing normally.

Code location:

contracts/core/Router.sol#L138

contracts/core/Router.sol#L317-L325

contracts/core/Router.sol#L330

```
function createIncreaseOrderWithTpSl(
    TradingTypes.IncreasePositionWithTpSlRequest memory request
) external whenNotPaused nonReentrant returns (uint256 orderId) {
    ...
    if (request.tp > 0 || request.sl > 0) {
        orderManager.saveOrderTpSl(
            orderId,
            TradingTypes.OrderWithTpSl({
                tpPrice: request.tpPrice,
                tp: request.tp,
                slPrice: request.slPrice,
                sl: request.sl
            })
        );
    }
```

```
        return orderId;
    }

    function addOrderTpSl(
        AddOrderTpSlRequest memory request
    ) external whenNotPaused nonReentrant returns (uint256 tpOrderId, uint256
slOrderId) {
        ...
    }

    function createTpSl(
        TradingTypes.CreateTpSlRequest memory request
    ) external whenNotPaused nonReentrant returns (uint256 tpOrderId, uint256
slOrderId) {
        ...
    }
```

Solution

It is recommended to check whether the price is reasonable when creating a TP/SL order.

Status

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

[N55] [Medium] Risk of over-privilege

Category: Authority Control Vulnerability Audit

Content

The RoleManager contract is used to manage the privileged roles of the protocol, including DEFAULT_ADMIN_ROLE, POOL_ADMIN_ROLE, OPERATOR_ROLE, TREASURER_ROLE, and KEEPER_ROLE. Among them, POOL_ADMIN_ROLE is used to set the necessary addresses in the protocol, some necessary parameter configurations, and manage the Pair in the Pool, and can pause the contract in an emergency. It is reasonable for EOA addresses to control the protocol's pause permission, as it can respond quickly in an emergency. However, it can add any pair and arbitrarily modify the necessary address parameters in the protocol, which will pose a risk of excessive privileges. If the PoolAdmin private key is lost, attackers can steal protocol funds through methods such as adding malicious pairs.

The protocol also designed a timelock contract to manage the permissions involved in sensitive parameter

modification, which will effectively reduce the risk of excessive privileges. Therefore, it is possible to transfer the operations involving excessive privilege risk of the PoolAdmin role to the timelock contract for management.

Solution

In the short term, in order to cope with the scenario that the protocol needs to frequently set parameters in the early stage, the PoolAdmin can be divided into two roles, one is an EOA address, which is used to manage the protocol's emergency pause permission, and the other is a multisign address, which is used to manage necessary parameter configuration and modification. This can solve the single-point risk without losing too much flexibility, but it cannot effectively mitigate the risk of excessive privileges. In the long run, it is more reasonable to entrust the protocol's parameter configuration and modification permissions to the timelock contract, and to entrust the timelock contract to community governance can effectively mitigate the risk of excessive privileges. This can also improve the trust of community users in the protocol.

Status

Acknowledged; After communicating with the project team, the project team stated that the poolAdmin role is a multi-signature wallet, and some privileges of the protocol will be transferred to the timelock contract, and will gradually transition to community governance in the future.

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002312040001	SlowMist Security Team	2023.11.09 - 2023.12.04	Medium Risk

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 11 critical risks, 4 high risks, 8 medium risks, 14 low risks, 14 suggestions, and 4 informations. All the findings were fixed or acknowledged. The code was not deployed to the mainnet. The excessive privilege risks in the protocol have not been fully resolved yet, thus there is still medium risk.

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