GA GUARDIAN

MUX

MUX 3 Protocol

Security Assessment

February 6th, 2025

Summary

Audit Firm Guardian

Prepared By Roberto Reigada, Osman Ozdemir, Nicholas Chew,

Zdravko Hristov, Mark Jonathas, Michael Lett

Client Firm MUX

Final Report Date February 6, 2025

Audit Summary

MUX engaged Guardian to review the security of their margin trading protocol. From the 2nd of December to the 6th of January, a team of 6 auditors reviewed the source code in scope. All findings have been recorded in the following report.

Issues Detected Throughout the engagement 12 High/Critical issues were uncovered and promptly addressed by the MUX team.

Security Recommendation Given the number of High and Critical issues detected Guardian recommends that an independent security review of the protocol at a finalized frozen commit is conducted before deployment.

Notice Further changes were made to the codebase after Guardian's remediation review which changes existing functionality and introduces new functionality. Guardian cannot attest to the security of these latest changes and how that might affect the final deployment.

For a detailed understanding of risk severity, source code vulnerability, and potential attack vectors, refer to the complete audit report below.

🔗 Blockchain network: **Arbitrum**

✅ Verify the authenticity of this report on Guardian's GitHub: https://github.com/guardianaudits

📊 Code coverage & PoC test suite: https://github.com/GuardianAudits/mux-fuzzing

Table of Contents

Project Overview

Project Summary

Project Name	MUX
Language	Solidity
Codebase	https://github.com/mux-world/mux3-protocol
Commit(s)	Initial: 16d65cccdedd43922458508cc5b282b02032ca1b Final: 8e7c05612cbdfabae44d14ebc690bb0a3447c69c

Audit Summary

Delivery Date	February 6, 2025
Audit Methodology	Static Analysis, Manual Review, Test Suite, Contract Fuzzing

Vulnerability Summary

Vulnerability Level	Total	Pending	Declined	Acknowledged	Partially Resolved	Resolved
Critical	1	0	0	0	0	1
High	9	0	0	0	0	9
Medium	24	0	0	16	1	7
Low	42	0	0	17	1	24

Audit Scope & Methodology

Vulnerability Classifications

Severity	Impact: *High*	Impact: *Medium*	Impact: *Low*
Likelihood: *High*	🔴 Critical	🟠 High	🟡 Medium
Likelihood: *Medium*	🟠 High	🟡 Medium	🟢 Low
Likelihood: *Low*	🟡 Medium	🟢 Low	🟢 Low

Impact

High Significant loss of assets in the protocol, significant harm to a group of users, or a core functionality of the protocol is disrupted.

Medium A small amount of funds can be lost or ancillary functionality of the protocol is affected. The user or protocol may experience reduced or delayed receipt of intended funds.

Low Can lead to any unexpected behavior with some of the protocol's functionalities that is notable but does not meet the criteria for a higher severity.

Likelihood

High The attack is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount gained or the disruption to the protocol.

Medium An attack vector that is only possible in uncommon cases or requires a large amount of capital to exercise relative to the amount gained or the disruption to the protocol.

Low Unlikely to ever occur in production.

Audit Scope & Methodology

Methodology

Guardian is the ultimate standard for Smart Contract security. An engagement with Guardian entails the following:

- Two competing teams of Guardian security researchers performing an independent review.
- A dedicated fuzzing engineer to construct a comprehensive stateful fuzzing suite for the project.
- An engagement lead security researcher coordinating the 2 teams, performing their own analysis, relaying findings to the client, and orchestrating the testing/verification efforts.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross-referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts. Comprehensive written tests as a part of a code coverage testing suite.
- Contract fuzzing for increased attack resilience.

Invariants Assessed

During Guardian's review of MUX, fuzz-testing with Echidna was performed on the protocol's main functionalities. Given the dynamic interactions and the potential for unforeseen edge cases in the protocol, fuzz-testing was imperative to verify the integrity of several system invariants.

Throughout the engagement the following invariants were assessed for a total of 10,000,000+ runs with a prepared Echidna fuzzing suite.

ID	Description	Tested	Passed	Remediation	Run Count
MUX-01	Filling a position should not fail due to rounding	✅	❌	✅	10M+
MUX-02	Collateral pool's aum is always greater or equal to reservedUsd	✅	✅	✅	10M+
MUX-03	aumUsd should never be 0 in a collateral pool when adl_max_pnl_rate < 1e18	✅	✅	✅	10M+
MUX-04	The sum of the reserved USD of each pool backing a given market should be greater than or equal to the total size in USD of the market multiplied by its reserve rate	✅	❌	❌	10M+
MUX-05	Since nextFundingTime is adjusted according to the interval, the timeSpan which is nextFundingTime - market.lastBorrowingUpdateTime should always be a multiple of the interval	✅	✅	✅	10M+
MUX-06	If a position account is fully closed, it should also be closed for any given market	✅	✅	✅	10M+
MUX-07	After opening a position, increase in market size should be equal to order's size	✅	✅	✅	10M+
MUX-08	withdrawUsd function should never revert with InsufficientCollateralBalance error when user's total collateral balance is sufficient (multiple active collaterals).	✅	✅	✅	10M+
MUX-09	withdrawUsd function should never revert with Invalid Amount error when user's total collateral balance is sufficient (multiple active collaterals).	✅	❌	✅	10M+

Invariants Assessed

ID	Description	Tested	Passed	Remediation	Run Count
MUX-10	A Liquidity order cannot be filled before the lock period has elapsed	✅	✅	✅	10M+
MUX-11	PositionAccount._positionPnlUsd function should never revert with AllocationPositionMismatch when closing a position	✅	✅	✅	10M+
MUX-12	Reserved USD in a pool should never exceed the pool's collateral value	✅	✅	✅	10M+
MUX-13	Market size should remain unchanged after reallocation operations	✅	N/A	✅	10M+
MUX-14	Total open interest in USD for a market must not exceed its configured cap	✅	N/A	✅	10M+
MUX-15	Position PnL must not exceed the maximum PnL rate cap when positive	✅	✅	✅	10M+
MUX-16	Changes in pool token supply must be reflected in liquidity changes	✅	✅	✅	10M+
MUX-17	Market total size must always be aligned with the configured lot size	✅	✅	✅	10M+
MUX-18	Pool parameters (size, k, reserve rate) must remain positive and allocations must not exceed pool size	✅	✅	✅	10M+
MUX-19	Pool allocations must stay within capacity limits and deallocation must not exceed previous allocation	✅	✅	✅	10M+
MUX-20	Borrowing rate parameters must be positive and utilization must not exceed 100%	✅	✅	✅	10M+

Invariants Assessed

ID	Description	Tested	Passed	Remediation	Run Count
MUX-21	Total pool allocations must approximately match the market's total size in USD	✅	❌	❌	10M+
MUX-22	Market lot sizes must remain constant during operations	✅	✅	✅	10M+
MUX-23	Market trade enabled status must remain constant during operations	✅	✅	✅	10M+
MUX-24	Market total allocations must match the sum of individual pool allocations	✅	✅	✅	10M+
MUX-25	Market size and pool allocations must be properly updated when positions are closed	✅	✅	✅	10M+
MUX-26	Position PnL must not exceed the pool's Assets Under Management (AUM)	✅	✅	✅	10M+
MUX-27	Position leverage must not exceed the maximum allowed leverage based on initial margin rate	✅	✅	✅	10M+
MUX-28	Position collateral must meet or exceed the required maintenance margin	✅	✅	✅	10M+

Findings & Resolutions

ID	Title	Category	Severity	Status
C-02	Inflation Attack Steals First Deposit	Inflation Attack	Critical	Resolved
H-01	Possible DoS Vector In The Delegate Function	DoS	High	Resolved
H-02	Liquidity Providers Can Abuse Unrealized Gains To Push Pool Utilization Over One Hundred	Configuration	High	Resolved
H-03	Liquidity Providers Can Withdraw Right Away	Configuration	High	Resolved
H-04	Price Query Failure Due To Transient Storage In PricingManager	Configuration	High	Resolved
H-05	Unauthorized Position Creation	Configuration	High	Resolved
H-06	Incorrect reservedUsd Logic For Shorts	Logical Error	High	Resolved
H-07	Delegators Cannot Cancel All Orders	Logical Error	High	Resolved
H-08	Multi-market Liquidations May Fail	Logical Error	High	Resolved
H-09	oneRound Function Potentially Allows Negative Allocations	Logical Error	High	Resolved
M-01	ClosePosition Only Updates The Current Market's Borrowing Fee	Configuration	Medium	Acknowledged
M-02	Inconsistent Borrowing Fee Calculation	Configuration	Medium	Acknowledged
M-03	setPrices Function Does Not Accurately Retrieve The Asset Prices	Configuration	Medium	Acknowledged

Findings & Resolutions

ID	Title	Category	Severity	Status
M-04	Avoid Using _strictStableIds	Configuration	Medium	Acknowledged
M-05	Lack Of A Grace Period For Configuration Changes	Configuration	Medium	Acknowledged
M-06	Add And RemoveLiquidity Orders Are Missing Slippage Checks	Configuration	Medium	Acknowledged
M-07	No Post-Deposit Safety Check In depositCollateral Function	Configuration	Medium	Acknowledged
M-08	Position Fee Not Factored Into Margin	Logical Error	Medium	Acknowledged
M-09	Empty Collateral Can Be Activated	Logical Error	Medium	Resolved
M-10	donateLiquidity Should Update Market Borrowing	Logical Error	Medium	Acknowledged
M-11	Broker Should Be Paid For Cancelling Orders	Logical Error	Medium	Acknowledged
M-12	Borrowing State Not Updated Before Config Change	Logical Error	Medium	Acknowledged
M-13	Gaming Capped Pnl By Increasing Position	Logical Error	Medium	Acknowledged
M-14	Imprecise PnL In realizeProfit	Logical Error	Medium	Resolved
M-15	Wrong Pause Check For Liquidations	Logical Error	Medium	Resolved
M-16	WithdrawUsd May Try To Withdraw 0	Logical Error	Medium	Resolved
M-17	Brokers Can Be Gas Griefed	Configuration	Medium	Acknowledged

Findings & Resolutions

ID	Title	Category	Severity	Status
M-18	MAX_COLLATERALS_PER_POSITION_ACCOUNT Can Be Bypassed	Logical Error	Medium	Acknowledged
M-19	Inaccurate aumUsd When Removing Liquidity	Logical Error	Medium	Acknowledged
M-20	Signature Replay In MuxPriceProvider Contract	Configuration	Medium	Resolved
M-21	ADL Is Triggered Per Pool	Configuration	Medium	Acknowledged
M-22	Broker Can Abuse oracleSigner Signatures	Configuration	Medium	Resolved
M-23	Zero Balance Active Collaterals	Configuration	Medium	Resolved
M-24	Profit Can Only Be Realized With The Pool's Collateral Token	Logical Error	Medium	Partially Resolved
L-01	Broker's Multicall Order Might Affect BorrowingRates	Configuration	Low	Acknowledged
L-02	Unrecoverable Revert In Broker's Multicall Execution	Configuration	Low	Acknowledged
L-03	Tokens That Do Not Support IERC20Metadata Interface Can Not Be Added As Collateral	Code Best Practices	Low	Acknowledged
L-04	Direct Delete Of EnumerableSetUpgradeable.UintSet Struct In _cancelActivatedTpslOrders Function	Code Best Practices	Low	Resolved
L-05	Liquidity Can Be Still Added To Pools On IsDraining State	Code Best Practices	Low	Resolved

Findings & Resolutions

ID	Title	Category	Severity	Status
L-06	Deallocate2 Function Does Not Prioritize Pools In IsDraining State	Configuration	Low	Acknowledged
L-07	CollateralPoolAumReader May Provide Divergent AUM Values Compared To The MUX3 Protocol	Configuration	Low	Partially Resolved
L-08	Lack Of A Double Step TransferOwnership Pattern	Code Best Practices	Low	Resolved
L-09	Missing ReentrancyGuard Initialization	Code Best Practices	Low	Resolved
L-10	Updating MM_LOT_SIZE Via setConfig Risks Precision Loss In Deallocation Algorithm	Configuration	Low	Acknowledged
L-11	Redundant Sequence State Variable In OrderBook Contract	Code Best Practices	Low	Acknowledged
L-12	Incorrect updatedAt Timestamp In SusdsOracleL2.latestRoundData Function	Configuration	Low	Acknowledged
L-13	Free Borrowing In Interval	Logical Error	Low	Acknowledged
L-14	Some Orders Cannot Be Paused	Configuration	Low	Resolved
L-15	Rebalance Orders Cannot Be Canceled	Configuration	Low	Resolved
L-16	Use Chainlink Stream's Bid/Ask Price	Logical Error	Low	Acknowledged
L-17	Unfillable TP/SL Orders Remain In The List	Logical Error	Low	Acknowledged

Findings & Resolutions

ID	Title	Category	Severity	Status
L-18	wrapNative Should Not Be Called Directly	Configuration	Low	Resolved
L-19	Fees Collected Should Round Up	Logical Error	Low	Acknowledged
L-20	Withdrawals Fail If Path Is Not Set	Configuration	Low	Resolved
L-21	Borrowing Rate Loses Precision	Configuration	Low	Resolved
L-22	Delegators Can Not Place Liquidity Orders	Configuration	Low	Acknowledged
L-23	Lack Of Fee On Transfer Support	Configuration	Low	Acknowledged
L-24	Withdrawal Slippage May Exceed 100%	Configuration	Low	Resolved
L-25	Conditional Token Check	DoS	Low	Resolved
L-26	Collateral Can Be Added To 0 Address	Code Best Practices	Low	Resolved
L-27	Unnecessary Check In _traderTotalUpnlUsd	Code Best Practices	Low	Resolved
L-28	Incorrect Event Data In donateLiquidity	Logical Error	Low	Resolved
L-29	Excess Collateral Not Refunded	Logical Error	Low	Acknowledged
L-30	Gaps Are Not Adding Up To 50	Code Best Practices	Low	Acknowledged

Findings & Resolutions

ID	Title	Category	Severity	Status
L-31	Redundant Function	Code Best Practices	Low	Resolved
L-32	addLiquidity Revert Due To Divide By 0	Configuration	Low	Resolved
L-33	Missing Withdraw Function In ChainlinkStreamProvider	Code Best Practices	Low	Resolved
L-34	Incorrect View Functions In CollateralPool	Logical Error	Low	Resolved
L-35	Unused Internal Functions	Code Best Practices	Low	Resolved
L-36	Incorrect Comment In ChainlinkStreamProvider	Code Best Practices	Low	Resolved
L-37	Warning About isWithdrawAll	Configuration	Low	Resolved
L-38	Typo	Code Best Practices	Low	Resolved
L-39	Some Transactions Update The Sequence Twice	Configuration	Low	Resolved
L-40	Unexpected Fill Price When Trigger Open	Configuration	Low	Resolved
L-41	Initial Leverage Is Mutable	Configuration	Low	Acknowledged
L-42	Referral Code Is Overwritten	Configuration	Low	Acknowledged

C-01 | Inflation Attack Steals First Deposit

Category	Severity	Location	Status
Inflation Attack	● Critical	CollateralPool.sol	Resolved

Description PoC

An inflation attack, commonly seen in ERC-4626 vaults, allows a malicious actor to steal deposits from the first depositor. This vulnerability exists in the collateral pool and can be executed as follows:

• The attacker deposits 1 wei of liquidity into the pool (e.g., ARB).
• The attacker observes a victim placing a liquidity order for 1000 ARB.
• The attacker front-runs the victim and donates 1000 ARB using the `donateLiquidity` function, inflating the `lpPrice`.
• The broker fills the victim's liquidity order, but due to the inflated `lpPrice`, the victim receives 0 shares.
• The attacker withdraws their shares, reclaiming the donated amount along with the victim's deposit.

The two-step order flow allows for 'front-running' which is usually not feasible on Arbitrum. This attack is also feasible only with 18-decimal tokens due to the `_toWad` conversion in `addLiquidity`.

Recommendation

Introduce a minimum share issuance threshold to ensure that deposits always result in a non-zero share allocation. Consider restricting the `donateLiquidity` function to privileged accounts.

Resolution

MUX Team: Resolved in commit 547c9abc7ba99d18246cee178554b209ba9a8123.

H-01 | Possible DoS Vector In The Delegate Function

Category	Severity	Location	Status
DoS	High	Delegator.sol: 32	Resolved

Description

The `Delegator` contract's `delegate` function allows users to delegate actions to a specific `delegator` by updating the `_delegators` mapping. However, this function lacks proper access control mechanisms, permitting any user to overwrite existing delegations for any `delegator` address.

Consequently, an attacker can call `delegate` with a target `delegator` address and replace the legitimate owner and action count with arbitrary values.

This vulnerability enables a DoS attack, where the original `delegator` is obstructed from performing delegated actions because their delegation has been maliciously overwritten.

Recommendation

Update the `delegate` function so it can only be called if the `_delegators[delegator].owner` is equal to the `address(0)`. Implement a new `removeDelegate` function that allows the `delegator` to reset the `_delegators[delegator]` mapping.

This way, the `delegator` can accept new delegations as now `_delegators[delegator].owner` is equal to the `address(0)`.

Resolution

MUX Team: Resolved in commit 6eab6d3adda18223f558a2086e30f9c1b23a1f7d.

H-02 | Liquidity Providers Can Abuse Unrealized Gains To Push Pool Utilization Over One Hundred

Category	Severity	Location	Status
Configuration	High	CollateralPool.sol: 491	Resolved

Description

The contract's AUM calculation logic incorrectly treats negative unrealized profit and loss from traders as an increase in the pool's available collateral. This is because in the `removeLiquidity` function the following check is performed.

However, if we take a look at the `_aumUsd` function, we can see that the `upnl` can be negative in case that the traders are at a loss and owe value to the collateral pool.

In that case, `_aumUsdWithoutPnl().toInt256()` will be subtracted a negative number which will artificially increase the final `aum` allowing liquidity providers to remove more liquidity than is genuinely available and, in some cases, even push the pool's utilization above 100%.

When utilization surpasses 100% or is very high, the borrowing rate curve causes dramatically increased costs for traders as it grows exponentially near full utilization, which severely disadvantages traders who are forced to pay these escalated costs.

Recommendation

Update the AUM computation to ensure that negative unrealized PnL owed by traders does not inflate the pool's available collateral. Instead of allowing negative unrealized PnL to translate into a higher AUM during the removal of liquidity, enforce a lower bound of zero on the `upnl = _traderTotalUpnlUsd(marketId)` calculation or introduce a separate accounting mechanism to distinguish between actual collateral and unrealized trader losses.

Resolution

Guardian: Resolved, however MUX team should ensure that in short markets:

1. A collateral pool that has the same collateral token as the market's underlying is never used.
2. Collateral pools with non-stable assets are never used.

H-03 | Liquidity Providers Can Withdraw Right Away

Category	Severity	Location	Status
Configuration	High	CollateralPool.sol	Resolved

Description

The current protocol design allows LPs to withdraw their liquidity at any time without any enforced delay. The `removeLiquidity` function only checks that the total reserved USD is less than or equal to the computed AUM.

As a result, LPs can instantly remove substantial amounts of collateral from a pool, increasing its utilization and causing borrowing rates to spike dramatically.

Traders relying on a stable borrowing environment are suddenly forced to pay higher borrowing fees or simply to close their positions as soon as possible.

Recommendation

Implement a withdrawal delay mechanism that prevents LPs from instantly removing large amounts of liquidity. By requiring a grace period before withdrawals are finalized, traders gain time to anticipate these changes by adding collateral or closing their positions.

`MCO_LIQUIDITY_LOCK_PERIOD` should be way higher than the 2 minutes set in the different test files.

Resolution

MUX Team: Resolved in commit beefa4c11533e673211514b9743fa45724bc740e.

H-04 | Price Query Failure Due To Transient Storage In PricingManager

Category	Severity	Location	Status
Configuration	High	PricingManager.sol	Resolved

Description

The `PricingManager` contract relies on transient storage to store and read price data. As a result, once the transaction that calls `setPrices` completes, the stored price data is cleared. Subsequent operations in a new transaction that rely on `_priceOf` as for example `withdrawAllCollateral` fail to retrieve valid prices, returning zero and reverting due to the require check.

This leads to a DoS scenario where users will not be able to perform certain operations that rely on the `PricingManager` returned price. Currently, the only operation that is affected is `withdrawAllCollateral` as this is the only function in the `OrderBook` that does not have to be called by a broker and uses the `_priceOf` function, querying the transient storage.

Recommendation

Switch to a permanent storage mechanism within `PricingManager` so that prices remain available across transactions. If the use of transient storage is still wanted, ensure all functions relying on `_priceOf` are executed within the same transaction that sets the price.

Resolution

MUX Team: Resolved in commit 951504ecf8da56c735039b9f0edf82e25251f82c.

H-05 | Unauthorized Position Creation

Category	Severity	Location	Status
Configuration	High	OrderBook.sol	Resolved

Description PoC

The permissionless `OrderBook.depositCollateral()` function allows anyone to deposit collateral for a given `positionId`. If the position account doesn't exist, it will be created.

As we can see from `LibCodec`, the first 20 bytes of the `positionId` are the address of its owner and the last 12 bytes show if the position supports one or many markets. If the last 12 bytes are 0, the account is a multimarket one, otherwise it supports only one market.

Each address can have up to `MAX_POSITION_ACCOUNT_PER_TRADER` different position accounts (currently set to 64). A malicious user can call `deposit` 64 times and add 1 wei of collateral to any `positionId`, which means they can create 64 single market accounts for a given address.

In result, the victim address will have to either use only single market accounts or change their address. However, the attack can be executed for their new address again.

This can also be detrimental for smart contracts if they have used only single market accounts, but try to create a multimarket one - the contract will probably experience DOS.

Recommendation

Add one of the following restrictions:
• `depositCollateral` is permissionless only if the `positionId` already exists.
• `depositCollateral` can be called only for positions owned by the caller.

Resolution

MUX Team: Resolved in commit 43bea77fee396cc2856cef7a19fec6856732dbe0.

H-06 | Incorrect reservedUsd Logic For Shorts

Category	Severity	Location	Status
Logical Error	High	CollateralPoolComputed.sol: 139	Resolved

Description

The `_reservedUsd` function calculates the liquidity that must be reserved to ensure sufficient funds are available to pay out traders. The calculation is currently implemented as:
`reserved = marketPrice * positions`

This approach is valid for long positions, as their payouts increase with rising prices. However, the logic fails for short positions. For shorts, profits are inversely related to the price. As the price increases, shorts incur losses, and as the price decreases, shorts generate profits.

The current implementation over-reserves liquidity for shorts when prices rise and, more critically, under-reserves liquidity when prices fall.

Recommendation

Consider implementing a separate `reservedUsd` logic for short positions, taking into consideration that for shorts, the maximum amount they can profit is up to their cost-basis (e.g. when opening a 1 ETH short at $3000, the maximum profit is $3000).

Resolution

Guardian: Resolved, however MUX team should ensure that in short markets:

1. A collateral pool that has the same collateral token as the market's underlying is never used.
2. Collateral pools with non-stable assets are never used.

H-07 | Delegators Cannot Cancel All Orders

Category	Severity	Location	Status
Logical Error	High	LibOrderBook.sol	Resolved

Description

The `Delegator.cancel()` function allows delegators to cancel orders on behalf of the delegation's owner. This works fine for position orders, because `LibOrderBook._cancelPositionOrder` allows the delegator to execute the certain action.

However, both `_cancelLiquidityOrder` and `_cancelWithdrawalOrder` require the `msgSender` (which is the delegator) to be equal to the position owner, which will always revert for delegated calls.

Recommendation

Add the `isDelegator` check to `_cancelLiquidityOrder` and `_cancelWithdrawalOrder` functions.

Resolution

Guardian: Resolved, however, liquidity orders can not be canceled as they can not be directly created through the Delegator.

H-08 | Multi-market Liquidations May Fail

Category	Severity	Location	Status
Logical Error	High	Global	Resolved

Description

When a position with multi-market exposure becomes liquidatable, the protocol intends to liquidate and close positions across all markets. However, the current implementation of the `liquidate` function can only process one market at a time.

This creates complications:
1. If the broker closes the profitable position in Market A first, the overall position may no longer be liquidatable due to lower margin requirements and increased collateral. This leaves the loss-making Market B active, which may be unexpected and undesirable for the trader.
2. Conversely, if the broker closes the loss-making position in Market B first, there may not be enough collateral to cover borrowing and position fees as the profits from Market A remain unrealized. This results in skipped and unpaid fees during liquidation.

Recommendation

Revise the `liquidate` function to loop through all associated markets and close every position within a single transaction.

Resolution

MUX Team: Resolved in commit 350a03bca14b6a91838e4849630d9e8afebf5dee.

H-09 | oneRound Function Potentially Allows Negative Allocations

Category	Severity	Location	Status
Logical Error	High	LibExpBorrowingRate.sol: 222, 235, 277, 278	Resolved

Description

Within the `oneRound` function, when a negative allocation is produced by `calculateXi`, the inner loop is broken but the partial candidate array is still copied into `bestXi`.

This allows negative allocations to be stored and subsequently used, totally breaking the purpose of the allocation algorithm logic.

Once a negative allocation is propagated into `mem.bestXi` and then allocated it will corrupt the overall allocation logic, causing mismatched liquidity calculations.

Recommendation

Update the `oneRound` function so `bestCost` and `mem.bestX` array are only updated if there are no negative allocations.

Resolution

MUX Team: Resolved in commit a346bf3d988eb7833bb58e36a9beba194472d164.

M-01 | ClosePosition Only Updates The Current Market's Borrowing Fee

Category	Severity	Location	Status
Configuration	Medium	FacetClose.sol: 45	Acknowledged

Description

When a position is closed, the contract updates only the borrowing fees for the currently closed market before verifying maintenance margin safety. This means that the margin check is performed without considering the borrowing fees pending to be paid in other still open markets.

As a result, once all borrowing fees eventually get updated for those other markets, the position may fall below the required maintenance margin and become liquidatable.

This could allow users to close a position on one market and leave the account appearing safe at the time of closure, despite actually being unsafe once all borrowing fees are updated.

Moreover, it also allows users to close a position that could be in a liquidated state but appears safe because the borrowing fees were not applied in the other markets.

Recommendation

Update all relevant market borrowing fees before performing the maintenance margin check by calling the `_updateBorrowingForAllMarkets` function so that the position's true value is evaluated.

Resolution

MUX Team: Acknowledged.

M-02 | Inconsistent Borrowing Fee Calculation

Category	Severity	Location	Status
Configuration	Medium	Global	Acknowledged

Description PoC

The current borrowing fee implementation can lead to situations where certain users end up paying proportionally more in borrowing fees than others, depending on how frequently their position or market is updated. The protocol defines `_updateMarketBorrowing` as a function that accumulates a per USD "borrow cost" (i.e. `cumulatedBorrowingPerUsd`) every time a funding interval elapses. However, the actual "charging" of that borrowing fee to a trader's position happens only in `_updateAndDispatchBorrowingFee` or `_updateAccountBorrowingFee` calls. Practically speaking, a position pays the difference between the current market's cumulatedBorrowingPerUsd and the position's stored `entryBorrowing` whenever the code calls `_updateAccountBorrowingFee`. In an idealized continuous interest model, it should not matter how many times you pay your borrowing fee, because the net owed over a specific time interval is the same. However, the code's logic effectively runs each time `_updateAccountBorrowingFee` is triggered for a position.

Because the fee is calculated as: `feeUsd = positionValue * (cumulatedBorrowingPerUsd[i] - pool.entryBorrowing) / 1e18` and immediately removed from the user's collateral, if your position is "touched" or updated more frequently, you might (depending on how the position's size or price changes between updates) accumulate fees in multiple smaller increments, each pulled from your collateral. Meanwhile, a position that is only "touched" at the very end of a long interval might remain unaffected until a single large settlement occurs. If Trader A's position has `_updateAccountBorrowingFee` called multiple times over an interval, then Trader A is "paying down" incrementally. If that triggers more "`positionValue`" recalculations at times when the position might be large or at unfavorable spot prices, Trader A can end up paying more total borrowing fees across many increments. Trader B, who never triggers a fee update until the end of a long period, might in some scenarios come out behind or ahead, depending on how the position size or price environment changed. If the size is smaller or the price is favorable near the time of the eventual update, B may pay less over that same time interval.

M-02 | Inconsistent Borrowing Fee Calculation

Category	Severity	Location	Status
Configuration	Medium	Global	Acknowledged

Description PoC

Typically, in a continuous or properly time-based interest model, two traders borrowing the same amount of capital over the same timespan should pay the same interest, irrespective of how many times that interest is "settled." The logic here is subtly different because the fee for each update depends on:

• The position's value at that update moment (`positionValue = pool.size * price / 1e18`).
• The difference between `cumulatedBorrowingPerUsd[i]` and the stored `entryBorrowing` from the last time the fee was applied.

If your position value or the `feePerUsd` differs across multiple, intermediate calls, you can pay more or less than a user who sees only a single large update. Therefore, frequent updaters might end up paying more overall borrowing fees if the position's size or mark price remains large each time an incremental fee is triggered and infrequent updaters could manage to pay less in total, if the price or position size shrinks by the time they do a final update. This difference becomes especially pronounced if the protocol or the broker calls `_updateAccountBorrowingFee` for some positions more often than others. On the other hand, the protocol ties borrowing fees to the current market price rather than a fixed entry price. Consequently, short positions that are winning (as the price went down) see their borrowing fees shrink and long positions in profit see the opposite effect, facing higher charges if their position is marked up repeatedly.

Recommendation

Consider migrating to a borrowing model that continuously accrues fees in a fair, time based manner, regardless of the number of updates. One option is to accumulate interest in a strictly pro rata fashion so that a user's borrowing fee liability is locked in each second or each block rather than heavily influenced by how many times the position's borrowing fee is updated. Another approach involves partially anchoring the fee to an initial or average notional to avoid large disparities caused by frequent or infrequent position touches. The ultimate goal is to ensure that two users with the same borrowed exposure for the same total time pay comparable amounts in borrowing fees, independent of how many times their fees happen to be updated.

Resolution

MUX Team: Acknowledged.

M-03 | setPrices Function Does Not Accurately Retrieve The Asset Prices

Category	Severity	Location	Status
Configuration	Medium	ChainlinkStreamProvider.sol: 55, MuxPriceProvider.sol: 30	Acknowledged

Description

The protocol's current configuration for the oracle-driven pricing mechanism allows the price expiration to be set anywhere between 30 seconds and up to 24 hours which.

As a result, when a broker performs a `multicall` that begins with calling `setPrices` to load price data and then proceeds to fill position orders, liquidity orders etc. the entirety of these operations may rely on prices that belong to different timestamps from the range: `[block.timestamp - 24 hours, block.timestamp]` .

Traders can use multiple collateral tokens to maintain their positions. Let's imagine that a trader holds a position backed by USDC, WETH and WBTC and the `oracleIds` for these assets have an expiration time of 24 hours and are using Chainlink Price Feeds with a heartbeat of 86400 seconds.

When the broker calls `setPrices` the price received for USDC could be from second 86399, for WETH 40000 and for WBTC 1. Therefore, this could result in positions that might be incorrectly assessed as safe or unsafe leading to unfair liquidations, missed opportunities for profitable trades…

Recommendation

Consider integrating with low-latency, high-frequency oracle solutions like Pyth, which provide more up-to-date prices with tighter heartbeat intervals.

Resolution

MUX Team: Acknowledged.

M-04 | Avoid Using _strictStableIds

Category	Severity	Location	Status
Configuration	Medium	PricingManager.sol: 23	Acknowledged

Description

The PricingManager's pricing logic for stable assets includes a mechanism to use a hardcoded reference value of 1 USD unless there is a major depeg.

While intended to maintain stability, this approach causes the protocol to rely on an artificially normalized value rather than accurate market data. This can cause positions backed by these stable assets to appear safer or riskier than they actually are, leading to incorrect margin calculations, mispriced positions, unexpected liquidations or unearned profits.

Essentially, traders and LPs might face unfair conditions due to the system forcibly ignoring genuine price signals and relying instead on an arbitrary stable reference price.

Recommendation

Rather than forcing a stable asset's price to a hardcoded value, consider a dynamic safeguard that triggers protocol-level responses during actual depeg events.

For instance, if the asset's price deviates too far from its peg, the protocol could pause certain trading operations, tighten collateral requirements, or invoke other protective measures until accurate prices are restored.

Resolution

MUX Team: Acknowledged.

M-05 | Lack Of A Grace Period For Configuration Changes

Category	Severity	Location	Status
Configuration	Medium	OrderBook.sol: 411	Acknowledged

Description

The contract's use of `setConfig` calls to modify critical parameters, such as `MCP_ADL_MAX_PNL_RATE` and other parameters, can impose immediate and drastic changes to the trading environment without providing users any time to adjust their positions.

Sudden alterations to margin requirements, liquidation thresholds or other essential settings can leave traders caught off guard and result in unexpected position losses and forced liquidations.

Recommendation

Introduce a delayed activation mechanism for all configuration changes. When a `setConfig` call modifies a critical parameter, the new value should enter a pending state for a predefined buffer period, for example 24 hours, before it becomes effective.

During this transitional time frame, traders can receive alerts or monitor upcoming changes, ensuring they have sufficient time to rebalance their portfolios, add collateral or close their positions.

Resolution

MUX Team: Acknowledged.

M-06 | Add And RemoveLiquidity Orders Are Missing Slippage Checks

Category	Severity	Location	Status
Configuration	Medium	CollateralPool.sol	Acknowledged

Description

When a Liquidity Order is placed there is no way for the caller to specify a minimum acceptable output amount.

In typical AMM or liquidity pool scenarios, a user will include a `minAmountOut` parameter to protect themselves from adverse price movements or sandwich attacks that may occur between the placement of the order and the fulfilment.

Without this parameter, the liquidity providers are subject to price changes and certain actions that can occur between the placement of their order and the fulfilment and therefore the liquidity provider could receive far fewer tokens (in the case of removing liquidity) or far fewer shares (in the case of adding liquidity) than they would expect under stable conditions.

Recommendation

Consider adding a `minAmountOut` parameter to the `LiquidityOrderParams` struct.

Resolution

MUX Team: Acknowledged.

M-07 | No Post-Deposit Safety Check In depositCollateral Function

Category	Severity	Location	Status
Configuration	Medium	LibOrderBook.sol: 864	Acknowledged

Description

The `depositCollateral` function allows users to add collateral to an existing position but fails to verify whether the position is then sufficiently collateralized or safe from liquidation immediately afterward.

A user might deposit too little collateral, resulting in a position that still remains under collateralized and can be liquidated right away.

Recommendation

Include a safety check after the collateral deposit to confirm that the position's margin requirements are now satisfied. If the position is still unsafe, revert the transaction.

Resolution

MUX Team: Acknowledged.

M-08 | Position Fee Not Factored Into Margin

Category	Severity	Location	Status
Logical Error	Medium	PositionAccount.sol	Acknowledged

Description

During liquidation, when `_isMaintenanceMarginSafe` is called, pending borrow fees are included in the required Maintenance Margin (MM). However, the position fee is not factored into the MM calculation.

Since this position fee depends on the position value, it can sometimes be substantial. In such cases, `_isMaintenanceMarginSafe` might return true, even when the collateral is insufficient to cover the position fee.

Despite this, the liquidation proceeds due to `shouldCollateralSufficient = false` in `_dispatchPositionFee`, leading to unpaid fees. This results in lost fee revenue for LPs and veMUX holders.

Recommendation

Update `_isMaintenanceMarginSafe` to account for the liquidation position fee as part of the collateral sufficiency check.

Resolution

MUX Team: Acknowledged.

M-09 | Empty Collateral Can Be Activated

Category	Severity	Location	Status
Logical Error	Medium	Market.sol	Resolved

Description

`Market._realizeProfit()` calls `CollateralPool.realizeProfit` which charges a given amount of the pool's collateral token from the LPs in order for the trader to be paid. After that, the charged collateral token is added towards the `activeCollaterals` of that trader.

This is done regardless of the charged amount. Because `CollateralPool.realizeProfit` converts from USD to the collateral token and divides multiple times, it's possible to have positive PnL in USD, but the result in collateral token to be 0.

When this happens, LPs won't be charged and the trader won't profit, but the collateral token of the pool will still be added to their `activeCollaterals`.

This will result in users having tokens with 0 balance in their `activeCollaterals`, which:
• Increases unnecessarily their collateral tokens count, making it harder to add new tokens.
• Can block the fulfilment of a close order which withdraws USD because `withdrawUsd` goes over each active collateral and tries to withdraw from it by passing the needed amount to `_withdrawFromAccount`. Inside that internal function there is a `require` assertion that will revert the transaction if the amount is 0.

Recommendation

In `Market._realizeProfit` add the `collateralToken` to `activeCollaterals` only if `collateralAmount` is not 0.

Resolution

MUX Team: Resolved in commit 4159688f031436f551b843a4d84c89b1e326f861.

M-10 | donateLiquidity Should Update Market Borrowing

Category	Severity	Location	Status
Logical Error	Medium	OrderBook.sol	Acknowledged

Description

Whenever a pool's liquidity changes, the `updateMarketBorrowing` function must be called to snapshot `cumulatedBorrowingPerUsd`. This ensures the borrowing costs are accurately tracked and distributed across the pool participants.

Currently, the `donateLiquidity` function does not perform this update. While this omission may not pose a risk when fees are donated through the fee distributor—since borrowing is updated during add/remove liquidity steps—`donateLiquidity` can also be called directly by external parties.

This could lead to an inconsistency in `cumulatedBorrowingPerUsd`, particularly if significant liquidity is donated without triggering the borrowing update, impacting the fair distribution of borrowing costs.

Recommendation

Ensure that `updateMarketBorrowing` is called within the `donateLiquidity` function

Resolution

MUX Team: Acknowledged.

M-11 | Broker Should Be Paid For Cancelling Orders

Category	Severity	Location	Status
Logical Error	Medium	LibOrderBook.sol: 743	Acknowledged

Description

Brokers have the ability to cancel stale position and withdrawal orders and the gas fees incurred for these cancellations are refunded to the user who created the order.

This design leaves brokers unreimbursed for the gas costs of performing the cancellations. A malicious actor can exploit this by creating multiple orders with unrealistic limit prices, ensuring that the orders never get filled and eventually go stale.

The broker would then be forced to expend gas to cancel these orders repeatedly, incurring significant costs without compensation.

Recommendation

Deduct a portion of gas fees to reimburse the broker when orders are cancelled.

Resolution

MUX Team: Acknowledged.

M-12 | Borrowing State Not Updated Before Config Change

Category	Severity	Location	Status
Logical Error	Medium	FacetManagement.sol	Acknowledged

Description

Borrowing params such as `baseApy` can be changed by admin in `FacetManagement.setConfig`. However, when borrowing params are changed, it affects the interest accrued by users since the last update time.

For example, if `baseApy` was increased, users will be unfairly charged the higher rate since the last update.

Consider this scenario:
T0 (last updated timestamp):
• `baseApy` = 1%

T5:
• Admin updates `baseApy` to 2%

T10:
• Trader closes position, borrowing rate is based on `baseApy` 2%, when it should have been 1% from T0 - T5 and 2% from T5-T10

Recommendation

Call `pool.updateMarketBorrowing` before changing any borrowing params.

Resolution

MUX Team: Acknowledged.

M-13 | Gaming Capped Pnl By Increasing Position

Category	Severity	Location	Status
Logical Error	Medium	CollateralPool.sol: 253	Acknowledged

Description PoC

Traders' PnL is capped based on `_adlMaxPnlRate`, and the capped PnL is used when realizing profits.

```
uint256 maxPnlRate = _adlMaxPnlRate(marketId); uint256 maxPnlUsd = (size * entryPrice) / 1e18; maxPnlUsd = (maxPnlUsd * maxPnlRate) / 1e18;
```

The capped PnL is calculated as above, with `maxPnlUsd` being directly influenced by the average entry price and the position size.

When a trader's PnL is higher than the capped PnL, the trader can intentionally increase their position at the current price just before closing it, then immediately close the position.

This will result higher profit for the trader without any additional risk since both the `size` and `entryPrice` are momentarily inflated by the trader, causing `maxPnlUsd` to be higher.

Recommendation

Consider implementing a grace period before allowing a position to be closed when a trader increases their initial position. This measure would discourage malicious traders by introducing an increased risk of loss during the grace period.

Resolution

MUX Team: Acknowledged.

M-14 | Imprecise PnL In realizeProfit

Category	Severity	Location	Status
Logical Error	Medium	Global	Resolved

Description

`Market._realizeProfit` accepts a `poolPnlUsd` amount that should be realized as a profit in the collateral pool. The same `poolPnlUsd` is being returned from the function as `deliveredPoolPnlUsd` and will later be saved in the corresponding `ClosePositionResult/LiquidatePositionResult` struct.

However, when `Market_.realizeProfit()` calls `CollateralPool.realizeProfit`, the pool recalculates the wad amount of collateral for tokens with different decimal tokens.

This will cause a discrepancy between the real USD value realized as profit and the value assigned to `deliveredPoolPnlUsd` for tokens with less than 18 decimals.

This can cause some close orders with `isWithdrawProfit = true` to revert because the `deliveredPoolPnlUsd` will be added towards the `withdrawUsd` that the order book will try to withdraw from the user's collateral.

Recommendation

Recalculate the actual USD profit value based on the `collateralAmount` returned from `CollateralPool.realizeProfit` and assign that new value to `deliveredPoolPnlUsd`.

Resolution

MUX Team: Resolved in commit b358c86d36239854f5ef8dffca1694a490009928.

M-15 | Wrong Pause Check For Liquidations

Category	Severity	Location	Status
Logical Error	Medium	OrderBook.sol	Resolved

Description

`OrderBook.liquidate()` liquidates a user by closing their position for a given market and potentially withdrawing their collateral if their MM is above their margin balance. This allows the Broker to keep the market in a healthy state.

Currently, `OrderBook.liquidate()` will execute the `whenNotPaused` modifier and revert if `LiquidityOrder` is paused. `LiquidityOrder` has nothing to do with liquidate even though their names are similar.

In result, if the market has the liquidity orders paused to manage some type of risk, the broker won't be able to liquidate position accounts.

Recommendation

Consider removing the `whenNotPaused` modifier from `OrderBook.liquidate()`. There is already a `_marketDisableTrade()` in `FacetClose.liquidatePosition()` which will stop the liquidation if this is a desirable feature.

Resolution

MUX Team: Resolved in commit/s bd5511cc3be7515304dc67c9306585e5eee9aa54, 53dd46cab5f62b6f14808b389acb6fe0de0f358b.

M-16 | WithdrawUsd May Try To Withdraw 0

Category	Severity	Location	Status
Logical Error	Medium	FacetPositionAccount.sol	Resolved

Description

`FacetPositionAccount.withdrawUsd` calculates the amount of `payingCollateral` to withdraw from the user's account by calling `_withdrawFromAccount`. It's possible for `payingCollateral` to result in 0 because of rounding if the user has small amount of collateral (may happen naturally by realizing negative PnL).

If this happens, the whole transaction will revert since `_withdrawFromAccount` fails if the amount to be withdrawn is 0. In result, the user won't be able to use `withdrawUsd`.

Recommendation

Skip the current iteration if the amount of collateral to be withdrawn is 0.

Resolution

MUX Team: Resolved in commit 8049b41bdce3e251e0660ecf5c614ba20a4c29a5.

M-17 | Brokers Can Be Gas Griefed

Category	Severity	Location	Status
Configuration	Medium	LibOrderBook.sol	Acknowledged

Description

Users place orders in the order book and pay a gas fee to compensate the broker that fulfils their orders. Users can also cancel their orders after a given time passes. When they do that, they will be refunded the whole gas amount they paid to compensate the broker.

This can be used to place multiple always reverting orders. For example, withdrawal orders with slippage over 100%, liquidity orders that exceed the liquidity cap, or just normal orders that happen to revert.

The broker will try executing them, which means it will pay gas for execution for the reverting transaction. The user can then cancel all their orders and get their funds back, resulting in lost funds for the broker.

Recommendation

Consider splitting the paid gas in two parts - send the first part back to the user and the second part to the broker to compensate them for any order they have tried fulfilling.

Resolution

MUX Team: Acknowledged.

M-18 | MAX_COLLATERALS_PER_POSITION_ACCOUNT Can Be Bypassed

Category	Severity	Location	Status
Logical Error	Medium	Market.sol: 249	Acknowledged

Description

When closing a position, collaterals are added to the user's account, without validation. If a user only partially closes their account, they will then have all of these collaterals as active.

When going to deposit collateral, the validation inside of `_depositToAccount()` will see that the collateral is already active and not revert even though the user is using more collateral assets than allowed.

Recommendation

If a user is not fully closing their position, validate that they are not exceeding the `MAX_COLLATERALS_PER_POSITION_ACCOUNT`.

Resolution

MUX Team: Acknowledged.

M-19 | Inaccurate aumUsd When Removing Liquidity

Category	Severity	Location	Status
Logical Error	Medium	CollateralPool.sol: 471, 491	Acknowledged

Description

When removing liquidity from the collateral pool, a check is performed to ensure that the remaining asset value is sufficient to cover the `reservedUsd` amount. This check is implemented using `require(reservedUsd = aumUsd)`.

This `aumUsd` value being compared to `reservedUsd` is calculated by subtracting the value of the removed asset from the pool's previous `aumUsd` value: `aumUsd = removedValue`. However, after this calculation, a portion of the `removedValue` is deposited back into the pool during `_distributeFee`.

Normally, this deposited amount increases the pool's `aumUsd`. However, this increase is not accounted for, making the `aumUsd` value used in the `require(reservedUsd = aumUsd)` statement inaccurate.

As a result, liquidity removal may fail incorrectly with an `InsufficientLiquidity` error, even though the actual `aumUsd` is sufficient to cover `reservedUsd`.

Recommendation

Re-added fees should be accounted for during the calculation.

Resolution

MUX Team: Acknowledged.

M-20 | Signature Replay In MuxPriceProvider Contract

Category	Severity	Location	Status
Configuration	Medium	MuxPriceProvider.sol	Resolved

Description

The `getOraclePrice` function in the `MuxPriceProvider` requires `oracleData.sequence` to be greater than the sequence of the contract. It doesn't specifically require it to be the next sequence, nor does it require it to be greater than the sequence of the last accepted oracle data.

When a valid signature sequence is beyond the contract's sequence, this signature can be used repeatedly by the broker until the sequences match. For example, if the current contract sequence is 4 and oracle data with a sequence of 10 is used, the contract sequence will be updated to 5.

Another oracle data with a sequence of 8 would still be considered valid. Assuming this is expected behavior, any broker can repeatedly call the `getOraclePrice` function, increment the contract's sequence to 10, and render the oracle data with a sequence of 8 invalid.

Recommendation

If oracle data is expected to be ordered and the behavior described above is not acceptable, either enforce the oracle sequence to be the exact next sequence or update the contract sequence to the latest accepted oracle data.

This ensures that oracle data with a smaller sequence will not be valid. If unordered oracle data is the intended behavior, consider implementing access control for the `getOraclePrice` function.

Resolution

MUX Team: Resolved in commit 57cd9a513eb7a9418903b5121592f14bcd54c283.

M-21 | ADL Is Triggered Per Pool

Category	Severity	Location	Status
Configuration	Medium	FacetReader: 230	Acknowledged

Description

When `fillADLOrder` is called, it validates that the position is viable for an auto deleverage. This is accomplished in `isDeleverageAllowed` function when it checks if any of the pools that the user has their position in is above the trigger PnL threshold.

The issue is that a user could have a position, in its entirety, not exceed the trigger threshold, but has exceeded the threshold in a singular pool. This will cause the entire position to be auto deleverage, even if their position has negative PnL.

Recommendation

Validate that the entire position is in a profitable enough state to trigger the auto deleverage.

Resolution

MUX Team: Acknowledged.

M-22 | Broker Can Abuse oracleSigner Signatures

Category	Severity	Location	Status
Configuration	Medium	MuxPriceProvider.sol: 54	Resolved

Description

The MuxPriceProvider contract does not include oracleData.oracleId in the message that is signed and verified. The signed message is built as:

```
bytes32 message = ECDSAUpgradeable.toEthSignedMessageHash(keccak256(abi.encodePacked(
Block.chainid, address(this), oracleData.sequence, oracleData.price,
oracleData.timestamp)));
```

Because the oracleId is not part of the signed data, a malicious broker can exploit this by taking a valid signature intended for one asset and using it to update the price of another asset.

By calling the setPrices function with a different oracleId but supplying the same oracleData and signature, they can manipulate the price feeds of other assets.

Recommendation

The oracleId must be included in the message that is signed by the oracleSigner and verified in the getOraclePrice function. This ensures that each signature is uniquely tied to a specific asset.

Resolution

MUX Team: Resolved in commit 31473d705b8caaaa9a09d76e83b40a66920ad443.

M-23 | Zero Balance Active Collaterals

Category	Severity	Location	Status
Configuration	Medium	PositionAccount.sol: 47-55	Resolved

Description

The `_depositToAccount` function checks the raw amount is not zero, converts raw amount to `wad` amounts, and then adds collateral to the active collaterals. However, if the collateral's decimal value is higher than 18, `wad` amount can be zero while the raw amount is not zero.

Since the balances are tracked as `wad` in the protocol, this collateral will still be added to active collaterals even though the `wad` balance is zero. Similarly, there will be some precision loss when withdrawing collaterals if the decimal value of the collateral is higher than 18.

Recommendation

Check the `wad` amount is not zero as well before adding collateral to the active collaterals.

Resolution

MUX Team: Resolved in commit 34dc6509682c10b1a89269d5a74e9fa31ef4eb29.

M-24 | Profit Can Only Be Realized With The Pool's Collateral Token

Category	Severity	Location	Status
Logical Error	Medium	CollateralPool.sol: 337	Partially Resolved

Description

The CollateralPool contract calculates its overall assets under management by summing balances of its primary collateral token and any additional tokens that arrive via fees, trader losses or rebalance operations. Although traders can open positions based on this total multi-token value, the code that settles profit only checks the availability of the main collateral, see CollateralPool.realizeProfit function below.

When traders attempt to realize profits, if there is insufficient balance of the primary collateral, even though other tokens in the pool collectively exceed the required amount, the function realizeProfit will revert, blocking some traders from closing and profiting from their positions due to the following require check:

```
require(wad = _liquidityBalances[token], InsufficientLiquidity(wad,
_liquidityBalances[token]));
```

The expected approach would be for rebalancers to convert these other tokens into the main collateral token, but there is no guarantee that this can be done quickly as it requires that the pool1's foreign token is held by any of the other pools. Therefore a significant time gap might occur before rebalancing is performed.

Furthermore, even if the protocol attempts to swap these foreign tokens for the main collateral on, for example through Uniswap, the pool would have to pay a high fee of around 5%, causing the CollateralPool to lose a portion of its AUM with each of those swaps.

Recommendation

Consider updating the realizeProfit function so trader's PNL can be paid with multiple tokens. These tokens should be any supported collateral token by the protocol and held by the actual collateral pool.

Resolution

MUX Team: Partially Resolved in commit ff0bd47258a1158414737c222e8164d031071cac.

L-01 | Broker's Multicall Order Might Affect BorrowingRates

Category	Severity	Location	Status
Configuration	Low	Global	Acknowledged

Description

When a broker executes operations within a single `multicall`, including setting token prices, fulfilling position orders, and then processing liquidity orders, the intended stability of borrowing rates can be compromised.

Position order fulfillment involves an allocation/deallocation algorithm designed to maintain balanced borrowing rates across collateral pools.

However, if a liquidity removal order is executed at the end of the same `multicall`, it immediately changes the utilization within a specific pool and thereby disrupts the previously established equilibrium.

This abrupt shift in utilization causes the borrowing rate to deviate, in at least one of the pools, significantly from the rate conditions calculated and maintained by the allocation algorithm.

Recommendation

Every time a broker fulfills orders within a `multicall`, the sequence of operations significantly affects the equilibrium of borrowing rates across collateral pools.

If liquidity removal occurs after position order fulfillment, it can distort the carefully balanced borrowing conditions achieved through the allocation algorithm.

Therefore, consider executing position orders last in the `multicall` array to ensure that the utilization and borrowing rates stay as balanced as possible.

Resolution

MUX Team: Acknowledged.

L-02 | Unrecoverable Revert In Broker's Multicall Execution

Category	Severity	Location	Status
Configuration	Low	OrderBook.sol: 87	Acknowledged

Description

The brokers will make use of the `OrderBook.multicall` function to execute a sequence of order-related operations, including setting prices, filling position orders and processing liquidity or rebalance orders.

However, if any single call within the `multicall` sequence fails, the entire transaction reverts and all previously completed operations within the same `multicall` are effectively rolled back.

This all-or-nothing behavior may cause unnecessary transaction failures when certain orders depend on the outcomes or state changes from preceding calls.

Recommendation

Consider introducing an extra parameter in the `multicall` function called `canFail`. If this parameter is set to `true` for a certain call, do not revert in case of failure and continue with the execution of the remaining calls.

Resolution

MUX Team: Acknowledged.

L-03 | Tokens That Do Not Support IERC20Metadata Interface Can Not Be Added As Collateral

Category	Severity	Location	Status
Code Best Practices	Low	CollateralManager.sol: 28	Acknowledged

Description

The `_retrieveDecimals` function unconditionally calls `IERC20MetadataUpgradeable(token).decimals` within a try/catch block.

While this appears to handle errors gracefully, there are still some scenarios where the call will still revert:
• If the address called is not a smart contract.
• If the address called implements the `decimals` function but does not return an `uint8`.

Recommendation

Merely informative issue.

Resolution

MUX Team: Acknowledged.

L-04 | Direct Delete Of EnumerableSetUpgradeable.UintSet Struct In _cancelActivatedTpslOrders Function

Category	Severity	Location	Status
Code Best Practices	Low	LibOrderBook.sol: 282	Resolved

Description

Within the `_cancelActivatedTpslOrders function`, the contract explicitly calls delete `orderBook.tpslOrders[positionId][marketId]` on an `EnumerableSetUpgradeable.UintSet` structure.

As per the Openzeppelin documentation:
"Trying to delete such a structure from storage will likely result in data corruption, rendering the structure unusable."

The exact impact would be:
• contains() returns true for previously stored values. It should not be a problem as it's never used in the codebase for the `tpslOrders` struct.
• add() behaves as if the set still contains values however only new `orderIds` (in an ascending order) will be added, so it is not an issue.
• Length of the set is reset to 0, even though there are still some elements that are part of it. In the current implementation this is not a problem.

Recommendation

At present, no immediate security issues arise from this approach. However, for clean design and to prevent complications in future upgrades, it is safer to clear each item in the set via `remove` rather than deleting the structure outright.

Resolution

MUX Team: Resolved in commit b4d1a45090f051ee896cfd13921e036f391bca8d.

L-05 | Liquidity Can Be Still Added To Pools On IsDraining State

Category	Severity	Location	Status
Code Best Practices	Low	CollateralPool.sol	Resolved

Description

The `CollateralPool` logic currently permits users to call `addLiquidity` even for pools that have their `isDraining` flag set. This state indicates that the protocol aims to reduce usage or permanently wind down the pool by only allowing deallocations.

Therefore, the borrowing rates there can just decrease as the pool's utilization will only decrease as the time passes. As the pool will eventually be deprecated, it is recommended to restrict the addition of any new liquidity.

Recommendation

Enforce a condition in the `addLiquidity` function to reject deposit attempts when a pool is flagged as `isDraining`.

Resolution

MUX Team: Resolved in commit 22e464b5636ef5c4dbcd16836e8eab99a018769e.

L-06 | Deallocate2 Function Does Not Prioritize Pools In IsDraining State

Category	Severity	Location	Status
Configuration	Low	LibExpBorrowingRate.sol: 332	Acknowledged

Description

In the current `deallocate2` implementation, the algorithm calculates how much liquidity `xTotal` should be proportionally removed from each pool. However, there is no consideration for whether a pool is in an `isDraining` state.

When a pool is flagged as draining, the intention is that liquidity should be prioritized for removal or deallocation, allowing that pool to wind down more quickly.

By simply performing a proportional distribution based on `mySizeForPool`, the function may remove only a minimal portion from a draining pool, rather than ensuring that the draining pool's allocation is decreased as much as possible to effectively close it out or free up its usage.

Recommendation

Similarly to how the allocation algorithm ignores pools in an `isDraining` state, the deallocation algorithm should be updated to prioritize pulling liquidity from pools that are in an `isDraning` state first.

Resolution

MUX Team: Acknowledged.

L-07 | CollateralPoolAumReader May Provide Divergent AUM Values Compared To The MUX3 Protocol

Category	Severity	Location	Status
Configuration	Low	CollateralPoolAumReader.sol	Partially Resolved

Description

The `CollateralPoolAumReader` contract is designed to let external projects (e.g., lending protocols) obtain an approximate "AUM" (Assets Under Management) for a given collateral pool.

It accomplishes this by reading Chainlink price feeds, whereas the core of the MUX3 protocol itself uses Chainlink data streams, which can produce slightly different price values than standard Chainlink aggregator feeds.

Because these two separate sources may yield inconsistent quotes for the same token, there is a risk that external contracts relying on `CollateralPoolAumReader` obtain an AUM measurement that deviates from the actual in-protocol valuation.

Inaccurate external references to the AUM can, in turn, lead to misguided lending rates, incorrect collateralization checks.

Recommendation

Projects that build on top of `CollateralPoolAumReader` should be aware of the possibility of discrepancies between the `CollateralPoolAumReader`'s AUM and the internal MUX3 protocol calculations.

To minimize the resulting risk of mispricing or suboptimal collateral usage, external integrators are advised either to adopt the same data source as MUX3's core or to introduce their own feed validation and tolerance checks, ensuring that any gap between the two sets of price data will not cause any impact on the protocol logic.

Resolution

MUX Team: Partially Resolved in commit 3056b7c351e554b1d3004d7d1584eaadeb54be4a.

L-08 | Lack Of A Double Step TransferOwnership Pattern

Category	Severity	Location	Status
Code Best Practices	Low	Global	Resolved

Description

The current ownership transfer process for all the contracts inheriting from the `OwnableUpgradeable` contract involves the current owner calling the `transferOwnership` function.

If the nominated EOA account is not a valid account, it is entirely possible that the owner may accidentally transfer ownership to an uncontrolled account, losing the access to all functions with the `onlyOwner` modifier.

Recommendation

It is recommended to implement a two-step process transfer ownership process where the owner nominates an account and the nominated account needs to call an `acceptOwnership` function for the transfer of the ownership to fully succeed.

This ensures the nominated EOA account is a valid and active account. This can be easily achieved by using OpenZeppelin's Ownable2Step contract.

Resolution

MUX Team: Resolved in commit 8562e574bc92bcc2a6efa2f110fb55f1debd5b35.

L-09 | Missing ReentrancyGuard Initialization

Category	Severity	Location	Status
Code Best Practices	Low	OrderBook.sol	Resolved

Description

The `OrderBook` contract inherits from `OpenZeppelin`'s `ReentrancyGuardUpgradeable` but never calls the `__ReentrancyGuard_init` function in its `initialize` method.

Recommendation

Within the `initialize` function, ensure to invoke the `ReentrancyGuard` initializer:

```
__ReentrancyGuard_init();
```

Resolution

MUX Team: Resolved in commit 3c25d0a70bc0b2fe24e28990bf8485d60dba9f11.

L-10 | Updating MM_LOT_SIZE Via setConfig Risks Precision Loss In Deallocation Algorithm

Category	Severity	Location	Status
Configuration	Low	LibExpBorrowingRate.sol: 332	Acknowledged

Description

The `MM_LOT_SIZE` parameter should never be updated through `setConfig` once there are positions open/allocated in the protocol because it directly affects the precision in the deallocation logic within `LibExpBorrowingRate`. The `deallocate2` function evenly distributes a total size `xTotal` across several pools based on each pool's proportion.

If `MM_LOT_SIZE` is updated, it will disrupt the prior assumptions about distribution granularity, causing significant rounding discrepancies when deallocating and therefore, in many cases, the deallocation would revert in the following require check present in the `_deallocateLiquidity` function blocking the closure or partial closure of user positions.

Recommendation

Never update `MM_LOT_SIZE` if there are any position open in the protocol.

Resolution

MUX Team: Acknowledged.

L-11 | Redundant Sequence State Variable In OrderBook Contract

Category	Severity	Location	Status
Code Best Practices	Low	OrderBook.sol	Acknowledged

Description

In the `OrderBook` contract, there is a `_storage.sequence` field which is incremented by the `updateSequence` modifier in several functions.

Despite these increments, the contracts never actually uses this `sequence` value to validate or order user operations; nor does any other function consume it to enforce conditions like strict ordering or replay protection.

Consequently, this sequence state variable becomes effectively useless, as it increments without being referenced.

Recommendation

Consider removing the sequence variable altogether or implement an actual usage of it as a required parameter for ordering or replay checks.

Resolution

MUX Team: Acknowledged.

L-12 | Incorrect updatedAt Timestamp In SusdsOracleL2.latestRoundData Function

Category	Severity	Location	Status
Configuration	Low	SusdsOracleL2.sol: 55	Acknowledged

Description

In the `SusdsOracleL2` contract, the `latestRoundData` method includes a field named `updatedAt` that is set to `block.timestamp`.

This is misleading because it implies that the price data was updated at the current block time on L2, whereas in reality, the update transaction originates on L1 via `SusdsOracleL1`.

The correct timestamp of the price update is the moment the message was relayed to L2 from L1, not necessarily the local `block.timestamp` of the L2 block executing `latestRoundData`.

As a result, the users may rely on a timestamp that does not accurately reflect when the underlying data (chi, rho, ssr) was actually refreshed.

Recommendation

Update the `SusdsOracleL2` contract to store the last updated timestamp passed from L1 when `updateFromL1` is called, rather than relying on `block.timestamp`. The L1-provided timestamp can be included alongside `chi_`, `rho_` and `ssr_`.

For example, extend the `updateFromL1` arguments to accept and record a `lastUpdated` value. Then, in `latestRoundData`, return that stored time as the `updatedAt`.

This ensures that any consumer reading the `SusdsOracleL2` receives a consistent and correct notion of when the oracle data was actually updated.

Resolution

MUX Team: Acknowledged.

L-13 | Free Borrowing In Interval

Category	Severity	Location	Status
Logical Error	Low	CollateralPool.sol: 611-619	Acknowledged

Description PoC

Traders must pay a borrowing fee for their positions, which is calculated based on the fee rate, duration, and position size. These borrowing rates and the cumulative borrowing per USD value are regularly updated when positions are opened or closed.

The `_updateMarketBorrowing` function performs the necessary calculations based on the `MC_BORROWING_INTERVAL` (currently set to 1 hour).

During these calculations, the `nextFundingTime` value is rounded down to the nearest interval timestamp using the formula `nextFundingTime = (blockTime / interval) * interval`. This value is then assigned to `market.lastBorrowingUpdateTime`.

If `market.lastBorrowingUpdateTime + interval = blockTime`, the cumulative borrowing per USD will not be updated. This allows traders to exploit the system by opening and closing positions within the same interval without incurring any borrowing fees, regardless of the position size.

With the current configuration, traders can freely borrow large amounts for approximately one hour without paying any borrowing fees by timing their actions.

```
T0                                                                T0 + 1 hour
|.................................................................|
    |                                                             |
Open position                                              Close position
```

Recommendation

Charge every position at least one interval rate regardless of the position's duration.
Alternatively, consider implementing a fixed borrowing fee that is independent of the duration, and then adding a time-based fee on top of this fixed fee.

Resolution

MUX Team: Acknowledged.

L-14 | Some Orders Cannot Be Paused

Category	Severity	Location	Status
Configuration	Low	OrderBook.sol	Resolved

Description

When orders are placed and filled in the `OrderBook`, the `whenNotPaused` modifier is executed first. It will revert if the current order type is paused. Currently, `_isOrderPaused` doesn't support `Rebalance` and `Adl` orders.

Because of this it will always return `false` and the modifier will always let the execution begin.
In result, `Rebalance` and `Adl` orders can be placed or filled at any point in time, even though the associated functions use the `whenNotPaused` modifier.

Recommendation

Make the `isOrderPaused()` function fetch paused configuration for `Rebalance` and `Adl` orders as well.

Resolution

MUX Team: Resolved in commit 53dd46cab5f62b6f14808b389acb6fe0de0f358b.

L-15 | Rebalance Orders Cannot Be Canceled

Category	Severity	Location	Status
Configuration	Low	LibOrderBook.sol	Resolved

Description

The LibOrderBook.cancel() function should be able to cancel placed orders. Currently there is not support for canceling a rebalance order. Because of this any expired rebalance orders will stay forever in the contract state.

Recommendation

Consider allowing the broker to cancel rebalance orders.

Resolution

MUX Team: Resolved in commit 70771d6ac481f4b40ab5ccb5829ef5a5034f4755.

L-16 | Use Chainlink Stream's Bid/Ask Price

Category	Severity	Location	Status
Logical Error	Low	ChainlinkStreamProvider.sol	Acknowledged

Description

`getOraclePrice` returns price from `verifiedReport` which is the mid-price (median of the bid and ask prices). While this is acceptable in stable market conditions, during periods of high volatility, the bid-ask spread can widen significantly.

References:
Chainlink recommends using liquidity-weighted price over mid-price:
https://docs.chain.link/data-streams/concepts/liquidity-weighted-prices

Synthetix studies past data of bid-ask price spreads: https://sips.synthetix.io/sips/sip-398/

Recommendation

Use the liquidity-weighted `bid/ask` prices provided in the `verifiedReport`, especially for the settlement of trades.

Resolution

MUX Team: Acknowledged.

L-17 | Unfillable TP/SL Orders Remain In The List

Category	Severity	Location	Status
Logical Error	Low	LibOrderBook.sol	Acknowledged

Description

When opening a position, users can create `tpsl` orders that will be filled in the future. The size of a `tpsl` order is the same as the size of the position when it is opened.

These `tpsl` orders will be cancelled in the `_fillClosePositionOrder` function only if the market is fully closed for that position account. When a user partially closes a position, these `tpsl` orders cannot be filled due to a mismatch between the remaining position size and the `tpsl` order size.

However, these orders will still remain in the `tpsl` list since the market is not fully closed. As a result, a position account might reach the `MAX_TP_SL_ORDERS` limit as some of these orders can never be filled.

Recommendation

Consider removing `tpsl` orders even when partially closing a position, document this behavior and request users to create new `tpsl` orders if they update their positions along the way.

Resolution

MUX Team: Acknowledged.

L-18 | wrapNative Should Not Be Called Directly

Category	Severity	Location	Status
Configuration	Low	OrderBook.sol	Resolved

Description

The function `wrapNative` was intended to be used as part of a multi-call to wrap ETH before it is deposited as gas or collateral. However, if it is called directly the user would end up losing that ETH as anyone can call `depositGas` to take the WETH that was left in the contract.

Recommendation

Consider documenting this risk for users.

Resolution

MUX Team: Resolved in commit 0f81979f86bb4c732d4dae5b7f21ec2a36136206.

L-19 | Fees Collected Should Round Up

Category	Severity	Location	Status
Logical Error	Low	PositionAccount.sol	Acknowledged

Description

Fees calculated in `_collectFeeFromCollateral` are rounded down after division. If `feeUsd` is small enough, it may round down to zero and allow the user to avoid paying a fee. This also applies to `_borrowingFeeUsd` and `_updatePositionFee`, to a smaller extent.

Recommendation

Always round up when calculating fees.

Resolution

MUX Team: Acknowledged.

L-20 | Withdrawals Fail If Path Is Not Set

Category	Severity	Location	Status
Configuration	Low	Swapper.sol	Resolved

Description

When users withdraw their collateral, they can specify a `withdrawToken` to receive. For example, USDC collateral may be withdrawn as DAI. To perform the swap, `Swapper` uses Uniswap.

In `swapAndTransfer`, if there is no `path` configured for the given pair of `tokenIn - tokenOut`, the transaction will revert. This will cause any withdrawal request from token A to token B where such swaps are not supported, to fail.

Recommendation

Consider sending the `tokenIn` to the receiver if no path is configured.

Resolution

MUX Team: Resolved in commit 216f2ae29a60117a54959846ad5a4ccb989f41ca.

L-21 | Borrowing Rate Loses Precision

Category	Severity	Location	Status
Configuration	Low	LibExpBorrowingRate.sol	Resolved

Description

`LibExpBorrowingRate.getBorrowingRate2` calculates the variable apy as `e^(k * utilization + b)`. Currently, both rounding down and division before multiplication happen when calculating the utilization.

Because of this, the utilization ends up being less than it should actually be. When later multiplied by `k`, this will lead to a smaller positive number to be added to the negative `b`.

The whole expression `k * utilization + b` will yield smaller negative number, therefore the variable apy will end up smaller than it should be.

Recommendation

Consider multiplying before dividing.

For example:
`fr = pool.poolSizeUsd > 0 = pool.k * pool.reservedUsd / pool.poolSizeUsd + pool.b : pool.b;`

Resolution

MUX Team: Resolved in commit 128aed0c42374f5cf5555e0d4ac388099fb22ded.

L-22 | Delegators Can Not Place Liquidity Orders

Category	Severity	Location	Status
Configuration	Low	Delegator.sol, OrderBook.sol, LibOrderBook.sol	Acknowledged

Description

Currently, delegators can place position orders and withdraw orders, but there is no support for liquidity orders.

Recommendation

Consider adding support for placing liquidity orders as well.

Resolution

MUX Team: Acknowledged.

L-23 | Lack Of Fee On Transfer Support

Category	Severity	Location	Status
Configuration	Low	Global	Acknowledged

Description

The protocol doesn't support fee on transfer tokens - it takes it for granted that the received amount is always the transferred amount. If such tokens were to be added, MUX would be broken.

Recommendation

Don't use fee on transfer tokens with the current version of the project.

Resolution

MUX Team: Acknowledged.

L-24 | Withdrawal Slippage May Exceed 100%

Category	Severity	Location	Status
Configuration	Low	LibOrderBook.sol	Resolved

Description

When users place withdrawal orders in `LibOrderBook` they may specify a desired`withdrawSwapSlippage` in percents with 18 decimals precision. This parameter is not validated and can be set to any valid `uint256`.

If its value is greater than `1e18`, the order will always revert on fulfillment.

Recommendation

Consider adding a validation in `placeWithdrawalOrder` that the slippage must not exceed `1e18`.

Resolution

MUX Team: Resolved in commit 3936e7fda7064e867ad3a875359eed0e5f13f92b.

L-25 | Conditional Token Check

Category	Severity	Location	Status
DoS	Low	LibOrderBook.sol	Resolved

Description

`LibOrderBook.placeWithdrawalOrder` checks if the passed `tokenAddress` is a valid collateral token only if it's not `address(0)`.

This means `address(0)` can be passed as token to be withdrawn. Such orders will always revert on fulfillment because there is the same check in `_withdrawFromAccount` however it's unconditional - the token must be a collateral token no matter if it's `address(0)`.

In result, users putting such orders will pay gas to the broker for executing an always reverting order.

Recommendation

Make the check in `LibOrderBook` always revert if the token to be withdrawn is not a collateral token, no matter if it's `address(0)`.

Resolution

MUX Team: Resolved in commit baadcd0eb79d1dc9ea664f5d1df5c09611364c72.

L-26 | Collateral Can Be Added To 0 Address

Category	Severity	Location	Status
Code Best Practices	Low	PositionAccount.sol	Resolved

Description

There is a check in `PositionAccount._depositToAccount()` that ensures the `positionId` is not 0. This will pass successfully even if the account passed is address(0), but one of the other 12 bytes is not 0. In result, users can deposit collateral for address(0).

Recommendation

Be aware of this behavior.

Resolution

MUX Team: Resolved in commit 43bea77fee396cc2856cef7a19fec6856732dbe0.

L-27 | Unnecessary Check In _traderTotalUpnlUsd

Category	Severity	Location	Status
Code Best Practices	Low	CollateralPoolComputed.sol: 129	Resolved

Description

In the function `_traderTotalUpnlUsd`, the check: `require(maxPnlRate > 0, IErrors.EssentialConfigNotSet("MCP_ADL_MAX_PNL_RATE"));` is unnecessary because it was already performed in `_adlMaxPnlRate`.

Recommendation

Remove the check for `maxPnlRate > 0`.

Resolution

MUX Team: Resolved in commit 221e1e48f09ff6f199f682e8862c758b6d6f6c5a.

L-28 | Incorrect Event Data In donateLiquidity

Category	Severity	Location	Status
Logical Error	Low	CollateralPool.sol	Resolved

Description

If an external party calls `donateLiquidity`, the `receiveFee` function in `CollateralPool` is called which emits two events with `collateralPrice`.

However, this price would be inaccurate as it relies on the broker to update token price, which is stored transiently in each block. If the call did not originate from the broker, price would be zero.

Recommendation

Consider restricting calls to `donateLiquidity`, so that it is certain that the call originated from the broker. Or else, document this risk for any external parties that rely on this event data.

Resolution

MUX Team: Resolved in commit e8b69415c11222eb44060e5bf341a18b7f3cbe6d.

L-29 | Excess Collateral Not Refunded

Category	Severity	Location	Status
Logical Error	Low	Global	Acknowledged

Description

When adding liquidity, a user is expected to first transfer tokens into the order book before calling `placePositionOrder`. However, when the action is filled via `fillPositionOrder`, there is no refund mechanism if the tokens transferred in exceeds the order's `rawAmount`.

Any excess tokens will remain in the contract and can be taken by the next user.

Recommendation

In `_transferIn`, consider refunding the difference between `realRawAmount` and `rawAmount`. Or else, document this risk for users.

Resolution

MUX Team: Resolved in commit 73d1534fb2587fcff5bd6f8dfcdb2a1c741df553.

L-30 | Gaps Are Not Adding Up To 50

Category	Severity	Location	Status
Code Best Practices	Low	Global	Acknowledged

Description

There are upgradability gaps left in the MUX contracts. These gaps don't follow the common practice of adding up to 50.

Recommendation

Be aware of this

Resolution

MUX Team: Acknowledged.

L-31 | Redundant Function

Category	Severity	Location	Status
Code Best Practices	Low	PositionAccount.sol: 313	Resolved

Description

`_isAccountExist` function in the `PositionAccount` contract is exactly the same as `_isPositionAccountExist`, and not used anywhere in the codebase.

Recommendation

Remove redundant function.

Resolution

MUX Team: Resolved in commit 22ed68cb7deb2cc59e62d3cf2f513fa0371af728.

L-32 | addLiquidity Revert Due To Divide By 0

Category	Severity	Location	Status
Configuration	Low	CollateralPool.sol: 399, 425	Resolved

Description

When users want to add liquidity to a collateral pool, the LP share price of that pool is calculated. This calculation is performed based on `_aumUsd`, which includes PnL. `_aumUsd` returns 0 when traders' total PnL is greater than underlying USD value of the pool.

In the `addLiquidity` function, `lpPrice` will also be 0 when `_aumUsd` is 0. This will cause a revert with divide by 0 error later in the function since share amount to mint to the user is calculated with `result.shares = (collateralAmount * collateralPrice) / lpPrice`.

Recommendation

There is no security consideration as long as `adl_max_pnl_rate < 100%` since the total PnL of all traders is also capped with this rate. However, be aware of this in case of updating `adl_max_pnl_rate` configuration values in the future.

Resolution

MUX Team: Resolved in commit 7515b499d4ac71cf6458dabec645c7f745fa9a0f.

L-33 | Missing Withdraw Function In ChainlinkStreamProvider

Category	Severity	Location	Status
Code Best Practices	Low	ChainlinkStreamProvider.sol	Resolved

Description

One of the price providers in the codebase is `ChainlinkStreamProvider`, which verifies oracle data using the Chainlink verifier. This verification requires a fee, which is paid using the `LINK` token.

The `ChainlinkStreamProvider` contract must maintain a balance of `LINK` tokens to cover these fee payments. However, the contract does not include a function that allows the owner to withdraw LINK balances if necessary.

Recommendation

Consider implementing a withdraw function.

Resolution

MUX Team: Resolved in commit e3fd954a04921fbb279c355ef424818f1c6f8b00.

L-34 | Incorrect View Functions In CollateralPool

Category	Severity	Location	Status
Logical Error	Low	CollateralPool.sol	Resolved

Description

The `getAumUsdWithoutPnl` and `getAumUsd` functions are external view functions in the `CollateralPool` contract. These functions read token prices from the `FacetReader` to perform calculations.

However, the prices are written to and read from transient storage. As a result, these functions will return incorrect values when called by external users.

Recommendation

These functions should be removed from the `CollateralPool`. External users should instead utilize the `CollateralPoolAumReader`, where prices are fetched from oracles.

Resolution

MUX Team: Resolved in commit d3657f5e4b12bd37779ef54103c0453a76f85eee.

L-35 | Unused Internal Functions

Category	Severity	Location	Status
Code Best Practices	Low	OrderBookGetter.sol, Mux3FeeDistributor.sol	Resolved

Description

The `_isMaintainer` and `_balance` functions in the `OrderBookGetter` contract, as well as the `_validateCollateral` function in the `Mux3FeeDistributor` contract, are internal functions but never used.

Recommendation

Consider removing unused functions.

Resolution

MUX Team: Resolved in commit e9a12903444d693631f0fcd01c420f7086305020.

L-36 | Incorrect Comment In ChainlinkStreamProvider

Category	Severity	Location	Status
Code Best Practices	Low	ChainlinkStreamProvider.sol: 20	Resolved

Description

The `Report` struct in the `ChainlinkStreamProvider` has this comment: "DON consensus median price, carried to 8 decimal places". However, current Chainlink streams have 18 decimal places.

Recommendation

Update the comment.

Resolution

MUX Team: Resolved in commit 626feebadd224dde3daf2a6d9f54800aa6e9c6ef.

L-37 | Warning About isWithdrawAll

Category	Severity	Location	Status
Configuration	Low	LibOrderBook.sol: 496, 634	Resolved

Description

The broker will pass the `isWithdrawAll` boolean value during liquidations and adl orders. The `isWithdrawAll` value must be set to `false` for `adl` orders if the position has multiple active markets (position index 0).

Otherwise, `adl` orders will fail, as the position will not be empty even after closing that market.

Recommendation

The broker should keep this scenario in mind.

Resolution

MUX Team: The issue was resolved in commit 9383952.

L-38 | Typo

Category	Severity	Location	Status
Code Best Practices	Low	LibOrderBook.sol: 799	Resolved

Description

"remove ths current order from tp/sl list" comment should be "remove the current order from tp/sl list" in `LibOrderBook` contract

Recommendation

Update the comment

Resolution

MUX Team: Resolved in commit 24218462d453759c141b66ad2c6e039e13d1d7c0.

L-39 | Some Transactions Update The Sequence Twice

Category	Severity	Location	Status
Configuration	Low	OrderBook.sol	Resolved

Description

The `OrderBook` has a sequence as a storage value. Some transaction flows update this sequence twice along the way due to the `donateLiquidity` function. This could cause issues if the off-chain part of the protocol expects the sequence to be updated one by one at all times.

Recommendation

Be aware of this behavior.

Resolution

MUX Team: Resolved in commit 0e33a86d6a7952551cec822ef0a13f64c3f9cdf0.

L-40 | Unexpected Fill Price When Trigger Open

Category	Severity	Location	Status
Configuration	Low	LibOrderBook.sol: 356-369	Resolved

Description

The fill price check is different for limit orders and trigger orders. Stop-loss orders are trigger and close orders, and the fill price check works correctly for these types of orders. However, users can also create trigger and open orders.

In this scenario, the current behavior and users' expectations might differ. Currently, trigger open orders work in a way that triggers a long position when prices are already going up (expecting a momentum/breakout strategy).

However, users might expect a long order to trigger when prices are going down (expecting a bounce or reversal/dip-buying strategy). For example, the current price is 100, and the user expects the price to drop to 90 and then bounce back.

From the user's perspective, a trigger open order in the long market at a price of 90 should create the order when the price reaches or goes below 90. However, this order would be immediately filled by the broker at the current price of 100.

Recommendation

Document this behavior and explain how trigger open orders work to prevent misunderstandings about these types of orders.

Resolution

MUX Team: Resolved in commit 92d115f8777e1abb5fcb88562d8ceb8b5bf90779.

L-41 | Initial Leverage Is Mutable

Category	Severity	Location	Status
Configuration	Low	OrderBook.sol: 147	Acknowledged

Description

A user can change the initial leverage of their position once it is opened. This is accomplished by simply calling `setInitialLeverage()` with a new value.

This will allow users to withdraw more collateral than they would otherwise be allowed to via `placeWithdrawalOrder()`, because the validation in `fillWithdrawalOrder()` will use the leverage value passed after the position was created.

Recommendation

Validate that the user is not modifying the initial leverage for a position that has already been opened.

Resolution

MUX Team: Acknowledged.

L-42 | Referral Code Is Overwritten

Category	Severity	Location	Status
Configuration	Low	Global	Acknowledged

Description

When users place position orders, they pass a `referralCode` parameter. After that, the code is passed to the referral manager's `setReferrerCodeFor` function which will update the code of the trader in the manager contract.

When position fees are to be paid, the code for the given trader is fetched from the referral manager and the recipient (therefore the tiers as well) are inferred from it.

There are 2 problems in the current implementation:
1. The code is being changed on each `placePositionOrder`.
2. The fulfilment of the orders is asynchronous, so even if 1 didn't exist, the code may still have changed.

For example, if a trader submits two position orders - the first with referral code A and the second with referral code B, at the time of fulfilment referral code B will be used for both orders.

Recommendation

Consider having the referral code encoded in the `orderParams`.

Resolution

MUX Team: Acknowledged.

Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Guardian to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Guardian’s position is that each company and individual are responsible for their own due diligence and continuous security. Guardian’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Guardian is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract’s safety and security. Therefore, Guardian does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

About Guardian Audits

Founded in 2022 by DeFi experts, Guardian Audits is a leading audit firm in the DeFi smart contract space. With every audit report, Guardian Audits upholds best-in-class security while achieving our mission to relentlessly secure DeFi.

To learn more, visit https://guardianaudits.com

To view our audit portfolio, visit https://github.com/guardianaudits

To book an audit, message https://t.me/guardianaudits