ClarityAlliance

Zest Protocol CLMM Security Review

Conducted by:
Clarity Alliance

November 4th, 2026



Contents

1. About Clarity Alliance 3
2. Disclaimer 3
3. Introduction 3
4. About Zest 4
5. Risk Classification 4
   5.1. Impact 4
   5.2. Likelihood 5
   5.3. Action required for severity levels 5
6. Security Assessment Summary 5
7. Executive Summary 6
8. Findings 9
   8.1. High Findings 9
      [H-01] Frame-Scaled Shares Permit Cheap Global Exhaustion of LP Mint Capacity 9
   8.2. Medium Findings 11
      [M-01] Aggregate LP Post-Conditions Permit Token-ID Substitution 11
      [M-02] Sub-WAD Bins Cannot Enforce Effective Price Limits 12
      [M-03] Immediate Fee Changes Can Consume Direct-Swap Input Without an Output Floor 14
      [M-04] Off-Chain Price Discovery Leads to Swap DoS and Suboptimal Routes 15
      [M-05] Governance Can Permanently Disable All Orderbook Exits 17
      [M-06] Absolute min-take Bounds Make Taker Fills Race-Prone 18
      [M-07] try! Bypasses Stale Managed-Leg Recovery 20
      [M-08] Flip Residuals Are Not Aggregated 21
      [M-09] Direct Core Fills Orphan Managed-Order Proceeds 22
      [M-10] Direct DAO Cancellation Orphans Managed-Order Escrow 23
      [M-11] Token Balance Checks Run Outside the Asset Guard 25
      [M-12] Absolute Output Floors Cannot Independently Protect Fee-Inclusive Prices of Partial Fills 26
      [M-13] Gross Amounts Are Incorrectly Counted Against Per-Bin Frame Capacity 27
   8.3. Low Findings 29
      [L-01] LP Token Events and Metadata Are Inconsistent with Relevant SIPs 29
      [L-02] One Extension Can Irreversibly Disable the DAO 30
      [L-03] Indirect Calls Can Use an Operator's tx-sender to Approve an Attacker Proposal 31
      [L-04] Swap Quotes Omit Execution Preconditions and Can Diverge From Actual Swaps 32
      [L-05] Swap Functions Lack Deadlines and Can Execute With Stale Price Limits 33
      [L-06] Reversed Token Ordering Allows Duplicate Pools for the Pair 34

[L-07] Just-in-Time Liquidity Can Capture Fees From Long-Term LPs 35
[L-08] Order Fragmentation Turns a Nominal Fee Rebate Into a Material Price Premium 36
[L-09] Signer-Set Changes Revive Expired Governance Proposals 37
[L-10] An Untrusted Intermediary Can Force User-Funded Orderbook and CLMM Trades 38
[L-11] Unbounded Market Creation Fragments Liquidity for Each Token Pair 40
[L-12] Off-Chain Price Selection Cannot Guarantee Best-Price Execution 42
[L-13] Orders Lack Expiry Timestamp 44
[L-14] An Untrusted Intermediary Can Force Vault Actions 45
[L-15] Historical Order Accounting Can Be Overflowed to Revert Managed-Order Fills 46
[L-16] Vault Price Bounds Do Not Protect the Fee-Inclusive Price of LP Shares 48
8.4. QA Findings 49
[QA-01] Variable Stacks Block Heights Are Used as Wall-Clock Time 49
[QA-02] Depositor Migration Leaves the Old Vault Accepting Undeployable Deposits 50
[QA-03] Heterogeneous LP Share Totals Have No Economic Meaning 51
[QA-04] Manager Rotation Pays the Outgoing Manager's Accrued Fees to the Incoming Manager 52
[QA-05] A Paused Manager Continues Earning Fees on Subsequent Liquidity Growth 53
[QA-06] Dust Reduction Burns LP Shares Without Paying Out 54

1. About Clarity Alliance

Clarity Alliance is a team of expert whitehat hackers specialising in securing protocols on Stacks.

They have disclosed vulnerabilities that have saved millions in live TVL and conducted thorough reviews for some of the largest projects across the Stacks ecosystem.

Learn more about Clarity Alliance at clarityalliance.org.

2. Disclaimer

This report is not, and should not be considered, an endorsement or disapproval of any project, team, product, or asset, nor does it reflect on their economics, value, business model, or legal compliance. It does not provide any warranty regarding the absolute bug-free nature or functionality of the analyzed technology.

Nothing in this report should be used to make investment or participation decisions. It does not constitute investment advice. Instead, it reflects an extensive assessment process intended to help clients improve code quality and reduce the inherent risks associated with cryptographic tokens and blockchain systems.

Blockchain technology presents ongoing and significant risk, and each company or individual remains responsible for their own due diligence and security posture. Clarity Alliance aims to reduce attack vectors and technological uncertainty but does not guarantee the security or performance of any system we review.

All assessment services are subject to dependencies and active development. Your access to and use of any services, reports, or materials is at your sole risk on an as-is and as-available basis.

Cryptographic tokens are emergent technologies with high technical uncertainty, and assessment results may include false positives, false negatives, or other unpredictable outcomes. Smart contracts may depend on multiple external parties, remain vulnerable to internal or external exploitation, and may carry elevated risks if owner privileges remain active. Accordingly, Clarity Alliance does not guarantee the explicit security of any audited smart contract, regardless of the reported verdict.

3. Introduction

A time-boxed security review of Zest Protocol's CLMM, where Clarity Alliance reviewed the scope and provided insights on improving the protocol.

4. About Zest

Zest Protocol Overview

How Zest Protocol turns Bitcoin from an idle asset into productive capital through Bitcoin Collateral Vaults and its Stacks product suite.

Zest Protocol is the capital layer for Bitcoin, turning Bitcoin from an idle asset into productive capital.

Bitcoin Collateral Vaults let holders borrow stablecoins against BTC or put it to work on any chain that can read a proof. The team has built on Bitcoin since 2021 and previously ran a $100M+ Bitcoin credit market on Stacks.

What is Zest CLMM?

Zest CLMM (concentrated-liquidity market maker) is a decentralized exchange venue on Stacks. Liquidity providers deposit two tokens into pools, and traders swap against those deposits. Its distinctive: instead of spreading liquidity across all possible prices, a provider places it inside chosen price intervals ("bins"), so the same capital trades with much greater depth inside the range it picked, and earns fees from every swap that passes through it. It is a discrete-bin cousin of Uniswap v3: a pool's price axis is divided into fixed-width geometric price steps (1%, 5%, 10% or 20%), and each step ("bin") holds its own little trading curve, its own reserves, and its own fungible shares.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

Scope

Repository: Zest-Protocol/clarity-clmm

The scope includes all the smart contracts in the repository.

Contracts

- contracts/clmm-pool-v1.clar
- contracts/clmm-router-v1.clar
- contracts/clmm-lp-token-v1.clar
- contracts/orderbook-exchange-v1.clar
- contracts/governance/bootstrap.clar
- contracts/governance/executor-dao.clar
- contracts/governance/operators.clar

Initial Commit Reviewed

- `bc463cbb12273fbc7fca672de16ae575463bd5c0`

Final Commit Reviewed

- `d7c817cf4a1824e7b2ccfd8d85d572270fef275d`

7. Executive Summary

Over the course of the security review, Clarity Alliance engaged with Zest Protocol to review Zest. Protocol's CLMM In this period of time a total of **36** issues were uncovered.

Protocol Summary

Protocol Name	Zest
Report Date	November 4th, 2026

Findings Count

Severity	Amount
High	1
Medium	13
Low	16
QA	6
Total Findings	**36**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Frame-Scaled Shares Permit Cheap Global Exhaustion of LP Mint Capacity	High	Resolved
[M-01]	Aggregate LP Post-Conditions Permit Token-ID Substitution	Medium	Resolved
[M-02]	Sub-WAD Bins Cannot Enforce Effective Price Limits	Medium	Acknowledged
[M-03]	Immediate Fee Changes Can Consume Direct-Swap Input Without an Output Floor	Medium	Resolved
[M-04]	Off-Chain Price Discovery Leads to Swap DoS and Suboptimal Routes	Medium	Acknowledged
[M-05]	Governance Can Permanently Disable All Orderbook Exits	Medium	Resolved
[M-06]	Absolute min-take Bounds Make Taker Fills Race-Prone	Medium	Acknowledged
[M-07]	try! Bypasses Stale Managed-Leg Recovery	Medium	Resolved
[M-08]	Flip Residuals Are Not Aggregated	Medium	Resolved
[M-09]	Direct Core Fills Orphan Managed-Order Proceeds	Medium	Resolved
[M-10]	Direct DAO Cancellation Orphans Managed-Order Escrow	Medium	Resolved
[M-11]	Token Balance Checks Run Outside the Asset Guard	Medium	Resolved
[M-12]	Absolute Output Floors Cannot Independently Protect Fee-Inclusive Prices of Partial Fills	Medium	Partially Resolved
[M-13]	Gross Amounts Are Incorrectly Counted Against Per-Bin Frame Capacity	Medium	Acknowledged
[L-01]	LP Token Events and Metadata Are Inconsistent with Relevant SIPs	Low	Resolved
[L-02]	One Extension Can Irreversibly Disable the DAO	Low	Acknowledged
[L-03]	Indirect Calls Can Use an Operator's tx-sender to Approve an Attacker Proposal	Low	Acknowledged
[L-04]	Swap Quotes Omit Execution Preconditions and Can Diverge From Actual Swaps	Low	Acknowledged
[L-05]	Swap Functions Lack Deadlines and Can Execute With Stale Price Limits	Low	Resolved
[L-06]	Reversed Token Ordering Allows Duplicate Pools for the Pair	Low	Resolved

[L-07]	Just-in-Time Liquidity Can Capture Fees From Long-Term LPs	Low	Acknowledged
[L-08]	Order Fragmentation Turns a Nominal Fee Rebate Into a Material Price Premium	Low	Acknowledged
[L-09]	Signer-Set Changes Revive Expired Governance Proposals	Low	Acknowledged
[L-10]	An Untrusted Intermediary Can Force User-Funded Orderbook and CLMM Trades	Low	Acknowledged
[L-11]	Unbounded Market Creation Fragments Liquidity for Each Token Pair	Low	Acknowledged
[L-12]	Off-Chain Price Selection Cannot Guarantee Best-Price Execution	Low	Acknowledged
[L-13]	Orders Lack Expiry Timestamp	Low	Acknowledged
[L-14]	An Untrusted Intermediary Can Force Vault Actions	Low	Resolved
[L-15]	Historical Order Accounting Can Be Overflowed to Revert Managed-Order Fills	Low	Resolved
[L-16]	Vault Price Bounds Do Not Protect the Fee-Inclusive Price of LP Shares	Low	Acknowledged
[QA-01]	Variable Stacks Block Heights Are Used as Wall-Clock Time	QA	Acknowledged
[QA-02]	Depositor Migration Leaves the Old Vault Accepting Undeployable Deposits	QA	Acknowledged
[QA-03]	Heterogeneous LP Share Totals Have No Economic Meaning	QA	Resolved
[QA-04]	Manager Rotation Pays the Outgoing Manager's Accrued Fees to the Incoming Manager	QA	Acknowledged
[QA-05]	A Paused Manager Continues Earning Fees on Subsequent Liquidity Growth	QA	Acknowledged
[QA-06]	Dust Reduction Burns LP Shares Without Paying Out	QA	Acknowledged

8. Findings

8.1. High Findings

[H-01] Frame-Scaled Shares Permit Cheap Global Exhaustion of LP Mint Capacity

Location: clmm-pool-v1.clar

Description `clmm-lp-token-v1` represents every LP token ID with independent balance and supply maps, but it also mirrors all of those shares into one native fungible token.

Every call to `mint` consequently increases both the selected token ID's mapped supply and the shared native FT supply.

The mapped supplies are independent, but the native FT has one protocol-wide `uint` supply capped at `2^128 - 1`. Once the sum of shares issued across every pool and bin reaches that limit, `ft-mint?` prevents the pool from minting any further LP positions due to overflow.

The pool limits reserves independently for each bin using a normalized frame of less than `2^118`, but LP shares are denominated in frame-scaled liquidity. At sufficiently negative ticks, raw X reserves are shifted left by 64 bits before liquidity is calculated, so a deposit of at most `2^54 - 1` raw X units can mint approximately `2^125` LP shares.

The PoC fills the first five valid 1% bins as follows:

Tick	Raw X deposited	LP shares minted	Percentage of global capacity
`-4458`	`2^54 - 1`	`66,532,443,134,204,839,771,336,337,238,666,510,336`	`19.552%`
`-4457`	`2^54 - 1`	`66,202,255,274,065,060,153,880,724,622,930,870,272`	`19.455%`
`-4456`	`2^54 - 1`	`65,873,706,073,470,141,329,717,399,794,957,156,352`	`19.359%`
`-4455`	`2^54 - 1`	`65,546,787,400,064,420,385,910,170,324,620,541,952`	`19.262%`
`-4454`	`2^54 - 1`	`65,221,491,161,851,619,296,832,716,910,696,071,168`	`19.167%`

These five deposits consume approximately `96.795%` of the global native FT supply while using only about `9.0 × 10^16` raw X units. The PoC then uses additional individually valid bins to reduce the remaining capacity below the shares required by a normal balanced deposit.

Consider the following scenario:

1. The attacker selects or creates a pool containing a token with sufficiently high decimals or raw-unit supply.
2. The attacker deposits only X into several bins near the minimum supported tick, causing frame scaling to mint disproportionately large LP share quantities.

3. The attacker uses progressively less extreme bins to consume enough of the remaining global supply that an ordinary LP mint no longer fits.
4. The attacker transfers every resulting LP position to the `clmm-pool-v1` contract principal. `reduce-position` can burn only shares belonging to `tx-sender`, so no external caller can withdraw or burn positions owned by the pool principal.
5. A victim attempts an ordinary first deposit into a different pool and bin. Its reserve amounts and per-bin share supply are valid, but the call to `ft-mint?` exceeds the shared aggregate supply limit and aborts with `ArithmeticOverflow`.

As a result, enough capacity is consumed to make a small deposit fail in a newly created, otherwise empty 5% pool. The attack deposits less than `10^18` raw X units in total, which is less than one whole token when X uses 18 decimals.

Recommendation

Remove the single native fungible token that aggregates exact balances across all LP token IDs, so one bin's share supply cannot consume mint capacity required by unrelated positions.

8.2. Medium Findings

[M-01] Aggregate LP Post-Conditions Permit Token-ID Substitution

Location: clmm-lp-token-v1.clar

Description Every CLMM LP position is identified by its pool and tick, and therefore can have a different economic value. `clmm-lp-token-v1` nevertheless mirrors the amounts of every token ID into one native fungible token.

A fungible-token post-condition can constrain how many aggregate LP units leave a user, but it cannot constrain which LP token ID leaves.

The transfer entry point authorizes a nested contract whenever the LP owner is the transaction sender. A malicious or compromised integration can therefore accept an intended low-value token ID from the user, ignore it, and transfer the same number of shares from a more valuable ID. An exact deny-mode FT post-condition still succeeds because it sees the expected asset contract and amount. The attacker can then redeem the substituted shares for the valuable bin's reserves.

The ratified SIP-013 Semi-Fungible Token Standard directly anticipates this problem in its **Use of post conditions** section:

1. An SFT post-condition needs to cover both the transferred amount and token ID.
2. Complete coverage is obtained by combining a native fungible token with a native non-fungible token.
3. The NFT marker's identifier should contain both the token ID and owner:

```
{ token-id: uint, owner: principal }
```

4. The marker is retagged with native NFT operations when balances change, so wallets can attach identifier-aware post-conditions in addition to the FT amount condition.

Recommendation Implement SIP-013's dual-native-asset representation: retain the fungible token for aggregate quantity post-conditions. Then, add an NFT marker keyed by `{token-id, owner}` and retag the affected owner/ID markers atomically on mint, transfer, and burn, following the SIP-013 `burn-and-mint` strategy.

[M-02] Sub-WAD Bins Cannot Enforce Effective Price Limits

Location: clmm-pool-v1.clar

Description The pool represents external prices and price limits as an integer WAD value, where one unit represents `10^-18` raw X per raw Y:

```
(define-constant WAD u1000000000000000000)
```

Internally, however, the supported tick range extends down to raw prices near `2^-64`, which is approximately `5.42 × 10^-20`. The internal frame preserves these prices using a Q60 mantissa and a signed binary exponent, but `sqrt-to-wad` converts the result back to an integer WAD value and floors it:

```
(define-private (sqrt-to-wad (fr ...) (s uint))
  (let (
      (pw (try! (mul-div (/ (* s s) ONE) WAD ONE)))
      (e (get e fr)))
    (ok (if (>= e 0)
      (bit-shift-left pw (to-uint e))
      (bit-shift-right pw (to-uint (- 0 e)))))))
```

Consequently, every real price below one WAD unit is reported as zero. The same granularity affects `frame-price-limit`, which accepts an integer WAD argument and converts it back into the internal frame before executing a swap or validating a deposit.

The following currently supported bins have an upper boundary below one WAD unit. Every possible price inside these bins is unrepresentable by the integer-WAD price arguments and get-bin-price return value.

Bin step	Affected ticks	Number of bins
`1%`	`-4458` through `-4167`	`292`
`5%`	`-909` through `-851`	`59`
`10%`	`-465` through `-436`	`30`
`20%`	`-243` through `-229`	`15`

The PoC uses the valid 20% bin at tick `-243`. Its represented raw-price interval is approximately:

```
Start: 5.740598992321453 × 10^-20 raw X per raw Y
End:   6.888718790785744 × 10^-20 raw X per raw Y

Start in WAD: 0.05740598992321453
End in WAD:   0.06888718790785744
```

There is no integer WAD value between those boundaries. `get-bin-price` therefore returns zero regardless of where the bin's actual price lies inside the interval.

For a Y-for-X swap, the caller supplies a minimum price:

`min-price`	Result

`0`	Below the entire bin and therefore nonbinding
`1`	Above the entire bin and therefore prevents all execution

For an X-for-Y swap, the caller supplies a maximum price and the behavior is reversed:

`max-price`	**Result**
`0`	Below the entire bin and therefore prevents all execution
`1`	Above the entire bin and therefore nonbinding

A user must therefore choose between allowing execution anywhere in the bin or disabling the swap completely. The user cannot express a price limit that permits the expected trade while rejecting adverse movement within the bin.

The following are affected:

- `get-bin-price` reports zero for every state in an entirely sub-WAD bin.
- `simulate-swap-x-for-y`, `simulate-swap-y-for-x`, `quote-swap-x-for-y`, and `quote-swap-y-for-x` cannot accept a meaningful in-bin price bound.
- The direct `swap-x-for-y-ioc` and `swap-y-for-x-ioc` entry points have no separate minimum-output argument, leaving their users without an effective on-chain slippage bound in these bins.
- Routed swaps use the same coarse price representation, although their aggregate `min-dx` or `min-dy` checks can independently protect the final output.
- `add-to-position` cannot express a narrow deposit price window in these bins, although `min-shares` can separately constrain share issuance.
- The `add-to-position` event prints the floored WAD price, so off-chain consumers observe zero rather than the actual represented price; swap events expose only the same integer-WAD limit supplied by the caller.

Recommendation

Replace integer WAD price arguments with a representation that retains sufficient precision across the complete supported tick range. The pool already uses a mantissa and binary exponent internally, so exposing a compatible fixed-point or `{ mantissa, exponent }` limit representation would avoid collapsing supported prices to zero and would allow limits to be converted inward without losing their intended precision.

Apply the new representation consistently to:

- Direct and routed swap limits
- Deposit price windows
- Price quote and simulation functions
- `get-bin-price`
- Printed deposit prices and swap-limit fields

[M-03] Immediate Fee Changes Can Consume Direct-Swap Input Without an Output Floor

Location: clmm-pool-v1.clar

Description Governance can immediately change either directional swap fee to any value below `100%`.

However, the direct entry points accept only a marginal curve-price limit and do not accept a minimum output or deadline. The router's single-bin wrappers add only a full-input-consumption check and therefore provide no output protection either.

The swap step first determines how much gross input to accept, calculates `fee = ceil(gross input * fee rate)`, and applies only the remaining net input to the AMM curve. The price limit constrains the price paid by this net input; it does not constrain the effective price based on the trader's gross expenditure.

Consequently, increasing the fee can leave the curve price within the trader's limit while consuming almost all their input as fees.

The PoC demonstrates the following sequence:

1. A bin is funded with `1,000,000,000,000,000` units of each token and initially has a `0%` X-to-Y fee.
2. A trader quotes an input of `1,000,000,000,000` X with a maximum price of `1.01 * WAD`. The quote returns `995,047,023,619` Y with no fee.
3. Before execution, governance raises the X-to-Y fee to `99.999999%`, represented as `99,999,999 / 100,000,000`.
4. The trader's unchanged transaction still consumes the complete `1,000,000,000,000` X input. It charges `999,999,990,000` X as fees, leaving only `10,000` X to move the curve and producing just `9,949` Y.

The executed output is therefore approximately `100,014,778` times smaller than the quote. Nevertheless, the swap satisfies the `1.01 * WAD` price limit because the relevant net curve price is approximately `10,000 / 9,949 = 1.0051` X per Y. An FoK call also succeeds because it checks only that the returned `dx` equals the requested `1,000,000,000,000` X, without constraining `dy`.

Recommendation

Add a minimum-output argument to both direct IOC swap functions and their FoK wrappers, and enforce it against the final output after fees. FoK execution should require both full input consumption and satisfaction of the caller's minimum output.

Consider capping the maximum fee percentage too such that users are guaranteed an upper fee limit.

[M-04] Off-Chain Price Discovery Leads to Swap DoS and Suboptimal Routes

Location: clmm-router-v1.clar

Description The pool does not maintain a global active bin. `get-bin-price` returns the price of a caller-selected tick, while the multi-bin swap functions accept a caller-selected list of at most 32 ticks.

The route validates only that this list is strictly ascending for X-to-Y or strictly descending for Y-to-X. It does not require the first tick to be the best currently executable bin, require later ticks to be the next executable bins, or detect that a better populated bin was omitted.

Price discovery and bin selection are therefore delegated entirely to a frontend or off-chain indexer. The contract has no authoritative pool-wide price against which an integration can validate its route, and a transaction cannot adapt its bin list to the state that exists when it is mined.

For example, consider a pool with a 20% bin step and no fees:

- Tick `1` covers prices from approximately `0.8333` to `1.00` X per Y and contains deep Y liquidity near its lower boundary.
- Tick `0` covers prices from `1.00` to `1.20` X per Y and also contains deep Y liquidity near its lower boundary.
- A small X-to-Y trade obtains approximately `1.20` Y per X from tick `1`, but only approximately `1.00` Y per X from tick `0`.

A caller may nevertheless submit only tick `0`. The list is valid, and a trade of 100 X can return approximately 100 Y even though the same pool contemporaneously offers approximately 120 Y through tick `-1`. The transaction can satisfy both a `1.20` maximum marginal price and any minimum output at or below 100 Y, so neither existing check detects that the caller received approximately 16.7% less output than the pool-wide optimum.

Importantly, this problem occurs when a route was optimal when signed but becomes stale before execution:

1. A frontend quotes a route through ticks `[0, 1]`.
2. Before execution, another transaction drains or worsens one of those bins, or adds better liquidity to tick `1`.
3. The pool recomputes execution against the current reserves of ticks `[0, 1]`, but never considers tick `1` because it was not supplied.
4. The transaction either executes against a suboptimal subset, partially fills, or reverts as unfilled even when other contemporaneous pool liquidity could have provided a better or complete fill.

An LP withdrawal creates the same availability failure even when the path was computed correctly from complete and current off-chain data. After the user signs the transaction, an LP can remove the output-side liquidity from one or more selected bins before the swap is mined. The route then reaches execution with insufficient liquidity to consume the requested input and, when partial fills are disabled, reverts with `ERR-NOT-FILLED`. This is unavoidable if the withdrawal leaves the entire pool without sufficient liquidity, but the current design also reverts when other, unlisted bins

could complete the trade because the fixed off-chain path cannot be recomputed against the execution-time state.

The absence of a global market state also permits independent bins to become mutually inconsistent or crossed. Consequently, there is no single authoritative current pool price: integrations must reconstruct directional best prices from external sources, and stale liquidity is left for arbitrageurs to reconcile. This makes correctness and availability depend on off-chain indexing infrastructure despite all information needed to select the best within-pool execution being present at transaction execution time.

Recommendation

Maintain one authoritative active bin for each pool and enforce that it is the only bin permitted to contain both X and Y. Every initialized bin below the active bin should contain only X, and every initialized bin above it should contain only Y. Deposits that would place Y below the active bin, X above the active bin, or mixed liquidity into any inactive bin should revert. This gives the pool one coherent market state and prevents mutually inconsistent or crossed bin inventories from being created through deposits.

Swaps should begin at the active bin and select subsequent bins on chain from the state that exists during execution. An X-to-Y swap should consume the active bin's Y, leave that bin X-only when it reaches its upper boundary, and advance the active pointer to the next higher bin containing Y. A Y-to-X swap should symmetrically leave the exhausted bin Y-only and move the pointer to the next lower bin containing X.

Deposits, withdrawals, and swaps must update the active-bin state and bin index. Expose the active tick and its current price through read-only functions, and make routed swaps derive their path from that tick rather than accepting an unrestricted off-chain path.

[M-05] Governance Can Permanently Disable All Orderbook Exits

Location: orderbook-exchange-v1.clar

Description Order cancellation depends on `orderbook-exchange-v1` remaining an enabled DAO extension. The DAO may revoke that status at any time.

The same proposal can also disable every extension capable of executing a recovery proposal. Afterwards, makers cannot call the core directly, cannot cancel plain or managed orders through the exchange, and cannot re-enable either contract. Every open order remains in `orderbook-core-v1` with its escrow permanently locked. Any managed-order residuals and cross-chain claims already held by the exchange also lose their governance-mediated release path.

This contradicts the stated safety property that governance cannot block exits. The documentation says that governance “cannot touch LP shares or orderbook escrow, or block exits”. It further states that `cancel-order` and `cancel-managed-order` remain available in paused and sunset markets and instructs users not to rely on governance to leave.

Pausing or sunsetting a market preserves cancellation, but revoking the exchange's extension status does not. Therefore, this is not merely an accepted consequence of the DAO's general extension-management power: it violates an explicit documented custody and exit guarantee.

The core has no maker-authenticated exit path. Instead, it authenticates the currently enabled exchange and trusts the exchange-supplied `maker` argument:

```
(define-public (cancel-order (maker principal) ...)
  ...
  (try! (is-dao-or-extension))
  ...
  (asserts! (is-eq (get maker order) maker) err-invalid-maker)
  ...)
```

The exchange supplies `tx-sender`, but the core accepts the call only while the exchange remains enabled:

```
(define-public (cancel-order (asset-trait <ft-trait>) (order-id uint))
  ...
  (contract-call? .orderbook-core-v1 cancel-order
    tx-sender asset-trait order-id))
```

Extension membership is therefore both an administrative permission and a hard dependency of the user exit path. Because membership is revocable, the exit path is revocable too.

A valid governance action can therefore permanently lock all assets escrowed in open orderbook orders. The attack requires control of the governance threshold, or an already-enabled extension with equivalent DAO power, but it breaks an explicit fund-safety boundary and has no on-chain recovery path once all production extensions are disabled.

Recommendation

Make user exits independent of revocable governance state. Add a core cancellation entry point that authenticates the maker from `tx-sender`, or use a non-revocable, narrowly scoped cancellation adapter, while keeping managed-order cancellation separately restricted to the exchange.

[M-06] Absolute min-take Bounds Make Taker Fills Race-Prone

Location: orderbook-core-v1.clar orderbook-exchange-v1.clar

Description `orderbook-exchange-v1::fill-orders` receives an absolute `min-take` value, executes against the state that exists when the transaction is mined, and reverts unless the final output meets that value.

Before execution, another taker can consume a targeted order, a maker can cancel it, or a maker can change the shape of a queue such that reachable orders within the limited processing limits result in a smaller total amount. Each action can reduce the amount reachable by the pending fill while leaving its fixed `min-take` unchanged.

The global 256-step cap makes this stale-state race controllable and repeatable. The cap is shared by every price in the taker's `prices` list: it is not reset when matching moves from one price level to the next. Once the 256th maker order is processed, `live` becomes false and every later invocation of `match-price` returns without inspecting its price level.

```
;; match-price
(if (get live s)
  ...
  (ok s))

;; match-step
(live (and order-filled (> left u0) (< steps u256)))
```

At a populated price, the matcher can process sixteen passes of sixteen orders. Each order consumes one step, regardless of how little economic value it represents.

The relevant liquidity is therefore not the aggregate liquidity at the prices supplied by the taker, but the liquidity reachable through the first 256 queue entries. A maker can change the queue between quotation and execution: adding orders increases total resting liquidity, yet can decrease the amount the capped matcher can reach in the same call by pushing later, larger orders beyond step 256. The static `min-take` cannot adapt to that change.

New orders are appended to the FIFO tail, and the public exchange provides `create-limit-orders`, which accepts up to 256 orders in one transaction:

```
(define-public (create-limit-orders
    (market-id uint)
    (maker-asset-trait <ft-trait>)
    (orders (list 256 { price: uint, amount: uint })))
  (fold create-limit-orders-iter orders ...))
```

An attacker can therefore observe a multi-level fill. Suppose the first selected price `A` has one small legitimate maker order and the taker intends to continue to significant liquidity at price `B`:

1. The taker submits a fill with `prices = [A, B]` and a `min-take` based on the expected liquidity at both levels.
2. Before it executes, the attacker appends 255 minimum-sized valid orders at `A`. They follow the existing head order, so FIFO does not prevent the attack.
3. The fill processes the original order and the 255 attacker orders, consuming all 256 matching steps. The later level `B` is never examined despite the taker retaining input.

4. The resulting output is below `min-take`, so `orderbook-exchange-v1.fill-orders` reverts at its slippage check.

The attacker can also back-run the failed fill with `cancel-order` calls. This restores the visible queue to its pre-attack state and releases the escrow, so a fresh quote and resubmission alone do not resolve the problem: the attacker can reinsert the batch before the next attempt and cancel it afterwards.

The one-off case of two takers competing for one maker order is expected serialized exchange behavior. The issue is the use of a stale absolute output as the only execution condition, combined with a bounded matcher that lets an untrusted maker deliberately create enough valid, negligible-relative-to-the-target FIFO entries to make that condition unreachable. `min-take` protects the taker's assets but supplies no liveness guarantee.

`min-order` limits the attacker's capital requirement but does not remove the vector as the taker's requested amount can be arbitrarily larger than `256 * min-order`.

Recommendation

For an immediate-or-cancel interface that permits partial fills, use a limit on the aggregate execution price instead of using an absolute `min-take` as the sole success condition. The limit should compare the actual input spent (including the taker fee) with the actual output received. A buy, for example, can require that its effective base-per-token price not exceed the taker's maximum. The call can then return a partial fill and refund the unspent input when queue padding prevents it from reaching later levels, rather than reverting merely because a stale absolute output quote was not met.

Alternatively, consider making queue griefing economically unattractive by charging an order-registration fee, or by applying an equivalent cancellation penalty to short-lived orders.

[M-07] try! Bypasses Stale Managed-Leg Recovery

Location: orderbook-exchange-v1.clar

Description `cancel-leg` is expressly intended to tolerate a managed leg which is already terminal in the core contract. A closed leg is expected to return zero, and the `match` branch attempts to translate `CORE-ORDER-CANCELED` (`u1010`) and `CORE-ORDER-FILLED` (`u1011`) into `(ok u0)`:

```
;; Cancels one leg and returns what it still held. A leg the core already
;; closed (it cannot normally be stale, see post-process-fill) returns zero.
(match (as-contract? () (try!
  (contract-call? .orderbook-core-v1 cancel-order tx-sender asset-trait order-id)))
  canceled (ok (get remaining-amount canceled))
  code (if (or (is-eq code CORE-ORDER-CANCELED)
    (is-eq code CORE-ORDER-FILLED)) (ok u0) (err code)))
```

However, `try!` unwraps only an `(ok ...)` response. When `cancel-order` returns `(err u1010)` or `(err u1011)`, `try!` returns that error immediately from `cancel-leg`. The enclosing `match` is never evaluated, so its `code` branch is unreachable for every core error.

Consequently, the actual outcomes differ from the intended ones:

Core state of the leg	Intended `cancel-leg` result	Actual result
Open	Remaining escrow amount	Remaining escrow amount
Cancelled	`(ok u0)`	`(err u1010)`
Filled	`(ok u0)`	`(err u1011)`

When this occurs inside `cancel-managed-order`, the outer `try!` also propagates the error. The entire transaction reverts, including the preceding `map-delete` operations, so the managed-order record remains and continues to reference the terminal leg. Its maker cannot cancel the managed order, withdraw its recorded residuals, or cancel the other still-live leg through this entry point.

The defect is triggered whenever the exchange record references a leg that is already terminal in the core. For example, after the DAO or an enabled extension other than the exchange contract directly calls `orderbook-core-v1::fill-orders` and as a result an exchange-owned leg is fully filled.

Recommendation

Match on the raw response from `cancel-order`; do not unwrap it before the branch that is meant to inspect its error code. This restores the intended idempotent cancellation behavior for terminal legs.

[M-08] Flip Residuals Are Not Aggregated

Location: orderbook-exchange-v1.clar orderbook-core-v1.clar

Description When a managed leg fills, `post-process-fill` attempts to place the received proceeds on the opposite side. If that opposite leg does not yet exist, `core-fund` calls `register-order` with only the proceeds from the current fill:

```
;; orderbook-exchange-v1.post-process-fill
(amount (get amount fill))
...
(flipped (if (and open (> flip-price u0) (> amount u0))
  (match (core-fund flip-asset flip-order amount (get market-id mo) flip-price)
    new-order (some new-order)
    err-value none)
  none))
```

If that registration fails, the current proceeds are added to the appropriate residual:

```
;; no flip: proceeds are held for the maker
(if is-buy
  (merge mo { token-residual: (+ (get token-residual mo) amount) })
  (merge mo { base-token-residual: (+ (get base-token-residual mo) amount) }))
```

However, a later fill again submits only its own `amount`; it does not combine that amount with the accumulated residual. This matters because `register-order` rejects a new order whose value is below the market's `min-order`:

```
;; orderbook-core-v1.register-order
(asserts! (>= (if is-buy amount (try! (mul-div amount price WAD)))
              (get min-order market))
          err-order-too-small)
```

For example, consider a market with a 10-B minimum order. Alice opens a sell-only managed order for 100 A at 2 B per A, with a configured buy flip price of 1 B per A. A taker can fill 1 A at a time, paying 2 B per fill. Every received 2 B amount is too small to register the missing buy leg, so every flip attempt fails and each 2 B is added to `base-token-residual`. After 100 such fills, Alice's managed order has received 200 B—far more than enough to create the configured buy leg—yet no flip has occurred.

When the final sell amount is consumed, both recorded leg IDs are zero, so the exchange settles and deletes the managed order instead of re-establishing its buy liquidity.

As a result, a taker can terminate a managed liquidity strategy by splitting otherwise equivalent execution into individually sub-minimum fills. A single fill of the same aggregate size would create the opposite leg and keep the strategy active.

Recommendation

Do not weaken `orderbook-core-v1`'s minimum-order rule. Instead, in `post-process-fill`, calculate the amount available for the attempted flip as the new proceeds plus the existing residual of the same asset. Submit that aggregate amount to `core-fund` and clear the residual only after `core-fund` succeeds. If the aggregate amount is still below `min-order`, or another flip precondition fails, retain the existing residual plus the new proceeds.

This preserves the market-level minimum while making the result independent of how a taker partitions an economically equivalent fill.

[M-09] Direct Core Fills Orphan Managed-Order Proceeds

Location: orderbook-core-v1.clar orderbook-exchange-v1.clar

Description The core permits the DAO and every enabled extension to call `fill-orders` directly:

```
;; orderbook-core-v1.fill-orders
(try! (is-dao-or-extension))
...
(try! (fold settle-fill fills ...))
```

`settle-fill` pays each maker directly from the core. A managed leg has `maker = .orderbook-exchange-v1`, so the proceeds of a direct fill are transferred to the exchange contract:

```
;; orderbook-core-v1.settle-fill
(and (> (get amount f) u0)
  (try! (transfer-out (get taker-asset-trait ctx) (get amount f)
    (get maker f))))
```

The core returns the fill records to its immediate caller, but it does not invoke the exchange. The only code which attributes the proceeds of a managed fill to its maker is `orderbook-exchange-v1::post-process-fill`, reached exclusively when the fill is initiated through `orderbook-exchange-v1::fill-orders`:

```
;; orderbook-exchange-v1.fill-orders
(try! (fold post-process-fill (get fills fill-result) ...))
```

As a result, whenever the DAO or an extension other than the exchange itself fills orders directly through the core contract, managed orders' proceeds can be permanently stranded.

Recommendation

Store the exchange principal in `orderbook-core-v1`. After transferring a fill's proceeds and refund to the exchange for an order whose recorded maker is the exchange, atomically call an authenticated exchange settlement callback. The callback must require `contract-caller == .orderbook-core-v1` and run the same managed-order processing as `post-process-fill`: update residuals, clear or replace filled legs, and fund the opposite leg where applicable.

[M-10] Direct DAO Cancellation Orphans Managed-Order Escrow

Location: orderbook-core-v1.clar orderbook-exchange-v1.clar

Description `orderbook-core-v1::cancel-order` authorizes either the DAO itself or an enabled extension. That is appropriate for market administration, but it also applies to custody cancellation. A threshold-approved proposal can therefore call the core directly and cancel a managed leg with `maker = .orderbook-exchange-v1`.

Managed legs are owned by the exchange in the core, while the exchange keeps the maker information, leg IDs, and residual accounting in `managed-orders`:

```
;; orderbook-exchange-v1.new-managed-order
(contract-call? .orderbook-core-v1 register-order
  market-id current-contract asset-trait sell-price amount)
...
(map-set managed-orders managed-order-id {
  ...
  sell-order: sell-order-id,
  token-residual: u0,
  base-token-residual: u0 })
```

The core authorization does not distinguish an administrative DAO call from the exchange's managed-order lifecycle call:

```
;; orderbook-core-v1.is-dao-or-extension
(or (is-eq tx-sender .executor-dao)
    (contract-call? .executor-dao is-extension contract-caller))
```

Consequently, a proposal executed through the production path `operators.approve -> executor-dao.execute -> proposal.execute` can invoke:

```
(contract-call? .orderbook-core-v1 cancel-order
  .orderbook-exchange-v1 asset-trait managed-leg-id)
```

The proposal runs with `tx-sender = .executor-dao`, so the core accepts the call. It marks the leg cancelled and transfers its `remaining-amount` to its recorded maker, the exchange:

```
;; orderbook-core-v1.cancel-order
(map-set orders order-id (merge order { canceled: true, ... }))
(try! (transfer-out asset-trait remaining-amount maker))
```

The exchange is not called during that transfer. Its managed-order record therefore still points to the now-cancelled leg and does not credit the amount to either residual field or to a cross-chain claim.

As a result, the amount transferred to the exchange contract remains permanently locked: it is not recorded as a managed-order residual nor as a cross-chain claim, and the exchange exposes no function that can transfer unaccounted balances to the maker or any recovery recipient.

Recommendation

Store the exchange principal in `orderbook-core-v1`. When cancelling an order whose recorded maker is `orderbook-exchange-v1`, require `contract-caller == orderbook-exchange-v1`. This prevents the

DAO and other extensions from directly cancelling managed legs and bypassing the exchange's accounting.

If governance needs to cancel managed orders, authorize the DAO to call `orderbook-exchange-v1::cancel-managed-order` for Stacks makers. The exchange then performs the core cancellation through its normal path, updates its managed-order records, and pays the recorded maker correctly.

[M-11] Token Balance Checks Run Outside the Asset Guard

Location: clmm-pool-v1.clar, orderbook-core-v1.clar, orderbook-exchange-v1.clar

Description Pulls wrap the token's `transfer` in `restrict-assets?`, and payouts wrap it in `as-contract?`. The balance-delta checks around them call the same token's `get-balance` outside the guard:

```
;; clmm-pool-v1.transfer-in
(let ((before (try! (contract-call? token get-balance current-contract))))
  (try! (restrict-assets? owner ((with-ft asset "*" amount))
    (try! (contract-call? token transfer amount owner current-contract none))))
  (ok (asserts! (is-eq (try!
    (contract-call? token get-balance current-contract)) (+ before amount)) ERR-TRANSFER-MISMATCH)))

;; clmm-pool-v1.transfer-out
(let (
    (pool-before (try! (contract-call? token get-balance current-contract)))
    (recipient-before (try! (contract-call? token get-balance recipient))))
  (try! (as-contract? ((with-ft asset "*" amount)) ...))
```

SIP-010 permits a public `get-balance`. Outside the guard it runs with the caller's unrestricted `tx-sender` authority, so a hostile or compromised listed token can:

1. Transfer a trader's other assets during a swap, deposit, withdrawal or orderbook pull. `restrict-assets?` does not cover the balance reads.
2. Call `executor-dao set-extension` during a governance proposal that triggers a payout, such as `claim-fees`. The proposal's `tx-sender` is `.executor-dao`, so `is-self-or-extension` passes and the token gains full DAO authority.

Recommendation

Run every token call, balance reads included, inside the guard: the reads and the transfer of a pull inside `restrict-assets?`, and those of a payout inside `as-contract?`. Route protocol fees to a treasury contract rather than the DAO, since a token the DAO moves itself still runs with DAO authority.

[M-12] Absolute Output Floors Cannot Independently Protect Fee-Inclusive Prices of Partial Fills

Location: orderbook-core-v1.clar orderbook-exchange-v1.clar clmm-pool-v1.clar clmm-router-v1.clar

Description The orderbook and pool both support executions that spend less than the caller's maximum input, but protect them with only an absolute minimum output. The input budget and output floor impose the coarse fee-inclusive bound `actualOutput / actualInput >= minOutput / inputBudget`. When an expected partial fill spends much less than the budget, however, that bound is proportionally weaker than the quoted execution rate. An absolute floor cannot independently express both the smallest partial quantity the user accepts and the minimum rate they accept for that quantity.

In `orderbook-exchange-v1::fill-orders`, the actual input is `amount - remaining-amount`, including the taker fee, while `min-take` checks only the absolute `take-amount`. Listed maker prices exclude that fee.

Consider an order for 100 tokens at a price of 1 and a taker who submits a 10,000-unit input budget. A `min-take` of 9,900 would provide the intended approximately 1.01 maximum rate but rejects the valid 100-token partial fill. Lowering it to 100 accepts that fill, which initially costs 100 and refunds 9,900. After governance raises the taker fee to 99%, the same call still returns 100 and passes `min-take`, but now spends the complete 10,000, of which 9,900 is the fee.

The pool has the same issue when an IOC or partial route reaches its price target before exhausting the input budget. `accept` grosses up the fixed net curve input as `ceil(net / (1 - feeRate))`; therefore, a fee increase can raise gross input while leaving net input, output and curve state unchanged. The marginal `max-price` or `min-price` still passes, as does the unchanged absolute `min-dy` or `min-dx`.

This pool variant affects both direct IOC directions and `swap-x-for-y-route` / `swap-y-for-x-route` when `allow-partial-fill` is true, including their router entry points. Pool fill-or-kill execution is not affected because its gross input is fixed.

Recommendation

Enforce a fee-inclusive execution-rate limit using the actual gross input spent and actual output received. One option is to define the submitted minimum output as the requirement for spending the complete input budget, calculate `scaledMinimum = ceil(minimumOutput * actualInput / inputBudget)`, and require `actualOutput >= scaledMinimum`. Provide a separate absolute minimum-fill parameter when callers also need to reject fills below a particular quantity. Retain the pool's marginal curve-price limit because it protects a different property.

[M-13] Gross Amounts Are Incorrectly Counted Against Per-Bin Frame Capacity

Location: clmm-pool-v1.clar

Description The swap steps limit the trader's gross input to the raw room remaining below the bin's frame bound before splitting that input into its net and fee portions.

This treats every input token as if it will be stored in the bin. In fact, the non-rebated protocol-fee portion is accumulated separately in `pool-fees`; only the net input and the LP-rebated fee increase the bin reserve.

For gross input `g`, fee rate `r`, fee rebate `q`, and denominator `D = 100,000,000`, the actual reserve increase is:

```
fee              = ceil(g * r / D)
LP fee           = floor(fee * q / D)
reserve increase = g - fee + LP fee
```

Therefore, capping `g` itself to the available room is correct only when no protocol fee is removed, such as at a zero fee rate or a 100% LP rebate. Otherwise the implementation rejects gross input that would leave the stored reserve strictly below the frame bound.

Because the same step functions serve both swap directions and are used by direct swaps, quotes, single-pool routes, and multi-pool routes, IOC execution can underfill and fill-or-kill execution can revert with `ERR-NOT-FILLED` even though the requested trade safely fits.

For example, let the raw frame capacity be `C`, the X reserve be `C - 1,001`, the remaining room be `1,000`, the X-to-Y fee be 10%, and the rebate be zero. A gross input of `1,100` pays a fee of `110`, so only `990` enters the bin and the final reserve is `C - 11`. The vulnerable cap nevertheless exposes only `min(1,100, 1,000) = 1,000` input to the swap calculation, leaves `100` unfilled, and makes a fill-or-kill route revert. Splitting the same input into separate `1,000` and `100` IOC swaps succeeds and produces the same safe final reserve.

Similarly, `add-to-position` frame-checks the gross post-deposit reserves: `deposit` first computes `L1g = liquidity(to-frame (x + dx), to-frame (y + dy))` on the full deposited amounts, while the reserves the bin actually stores are net of the deposit's own implicit-swap fee:

```
implicit swap n   = min(excess, ceil((z + dz) * L0 / L1) - z)
gross swap input  = min(ceil(n / (1 - r)), dz)
fee               = ceil(gross swap input * r / D)
LP fee            = floor(fee * q / D)
stored increase   = dz - fee + LP fee
```

So the capacity gate counts `dz` in full although at most `dz - fee + LP fee` is recorded. As with the swap cap, the gross check is exact only when no fee is removed: a first deposit into a bin, an in-ratio deposit, or a zero fee rate. Any fee-paying out-of-ratio deposit (fee up to the 10% pool cap, charged on the deposit's implicit swap) is rejected earlier than necessary, and unlike the swap path it has no partial-fill mode to soften the failure, so the whole `add-to-position` reverts with `ERR-AMOUNT-TOO-LARGE`.

For example, in a 1% pool at tick `-4458` the frame exponent is `e = -64`, so the X-side threshold is `C = 2^54 = 18,014,398,509,481,984` raw units. Seed the bin to `x = C - 1,000` with equal frame reserves

on the Y side (mid-bin price) and a 10% X fee with zero rebate. Mid-bin, a one-sided deposit's implicit-swap input is about half the deposit (`n ~ dx / 2`), so the fee is material: a one-sided X deposit of `dx = 1,050` pays `fee ~ 59` and would store `~991`, leaving the reserve at about `C - 9`, strictly inside the frame. The gross check nevertheless sees `x + dx = C + 50` and reverts the deposit. Depositing `949` and then `100` instead (the same liquidity, in two calls) succeeds, pays fees of `53` and `6` on chain, and leaves the reserve at `C - 10`, essentially the state the single deposit was denied.

The issue causes false reverts and deterministic underfills.

Recommendation

Cap input using the exact amount that will be recorded in the bin rather than gross input.

Pass the pool's fee rebate into both swap-step functions and choose the greatest gross input for which `gross - ceil(gross * feeRate / D) + floor(ceil(gross * feeRate / D) * feeRebate / D) <= room`. Preserve the existing gross-input cap when the fee rate is zero or the rebate is 100%, use identical logic for X-to-Y and Y-to-X execution and quoting paths.

Apply the same principle to deposits by reordering rather than inverting: `fee-x`/`fee-y` and their LP shares depend only on `L0` and the gross `L1g`, so compute them first, then frame-check the reserves that will actually be written, `x + lp-fee-x + dx - fee-x` and `y + lp-fee-y + dy - fee-y`, instead of `x + dx` and `y + dy`.

8.3. Low Findings

[L-01] LP Token Events and Metadata Are Inconsistent with Relevant SIPs

Location: clmm-lp-token-v1.clar

Description `clmm-lp-token-v1` declares the SIP-013 trait but does not emit the transfer events prescribed by the standard.

Its `sft_transfer` tuple contains an additional `memo` field, while `transfer-memo` and `transfer-many-memo` do not print the raw memo as a separate event after the standard transfer event.

Its metadata behavior is also inconsistent with SIP-016, which requires `get-token-uri` to return `some` if and only if the referenced SFT token ID exists; once a base URI is configured, the contract constructs a URI for every arbitrary ID without checking whether it has been minted.

Conversely, `set-token-base-uri` accepts a 256-character base even though `get-token-uri` appends the decimal token ID and `.json`, allowing an accepted configuration to make the final `as-max-len?` return `none` for valid IDs.

Finally, changes made through `set-token-base-uri`, `set-name`, and `set-symbol` do not emit the SFT metadata-update notification required by SIP-019, so indexers have no standardized signal to invalidate cached metadata.

These inconsistencies do not affect token accounting, but can cause standards-compliant wallets and indexers to reject or misclassify transfers and memos, expose metadata for nonexistent IDs, omit metadata for valid IDs, or continue displaying stale information.

Recommendation

Emit the exact SIP-013 `sft_transfer` tuple after the native FT event and print the raw memo as the final event for each memo-bearing transfer, including every item processed by `transfer-many-memo`.

Track whether each token ID exists and make `get-token-uri` return `none` for IDs that have never been minted; also restrict or validate the base URI so that the base, maximum decimal token ID, and `.json` suffix always fit within 256 characters.

Make metadata immutable or emit SIP-019 SFT notifications containing the affected token IDs whenever the base URI, name, or symbol changes.

[L-02] One Extension Can Irreversibly Disable the DAO

Location: executor-dao.clar

Description `set-extension` and `set-extensions` allow an enabled extension to disable itself and every other extension without preserving a recovery authority. Once `construct` has transferred the executive role to the DAO contract, no external principal can satisfy the self branch of `is-self-or-extension`; if all extensions are disabled, every later attempt to execute a recovery proposal or re-enable an extension therefore fails permanently.

This also disables functionality in contracts that use extension membership as an operational authorization rather than only as an administrative permission: in the current deployment, revoking `orderbook-exchange-v1` prevents the orderbook core from accepting maker cancellations, so existing escrow can be locked.

A governance proposal, or any enabled extension with a suitable call path, can thus irreversibly deny all governance operations and dependent user exits.

Recommendation

Prevent an update from disabling the final designated governance executor, and use a two-step rotation that enables and verifies a replacement before the previous executor can be removed.

Alternatively, consider retaining a narrowly scoped, immutable recovery mechanism that can only restore the governance executor and cannot exercise general protocol powers.

[L-03] Indirect Calls Can Use an Operator's tx-sender to Approve an Attacker Proposal

Location: operators.clar:76

Description `operators::approve` authenticates an approval only by checking that `tx-sender` is present in the operator map and does not require the operator to be the immediate `contract-caller`.

When an operator calls an intermediary contract, Clarity preserves that operator as `tx-sender`, so the intermediary can call `approve` with an attacker-controlled proposal and the operator check succeeds even though the operator never called the operators contract directly.

Once enough distinct operators have been induced to call the intermediary, the approval reaching the threshold executes the attacker proposal with the executor DAO's full authority, allowing arbitrary governance actions such as enabling attacker extensions or permanently disabling governance and dependent exit paths.

The shipped 1-of-1 bootstrap requires only one such call for successful exploitation.

Recommendation

If operators are standard accounts, require both `is-operator tx-sender` and `contract-caller == tx-sender` before recording an approval.

If contract operators must be supported, authenticate the immediate `contract-caller` against an explicitly configured operator principal and require each operator contract to enforce its own authorization.

[L-04] Swap Quotes Omit Execution Preconditions and Can Diverge From Actual Swaps

Location: clmm-pool-v1.clar

Description `quote-swap-x-for-y` and `quote-swap-y-for-x` load the current pool and bin state and invoke `step-x-for-y` or `step-y-for-x` directly. The corresponding state-changing swap functions use the same step calculations, but impose additional execution conditions that the quote functions do not model.

Consequently, a successful quote does not necessarily represent a swap that can execute:

- Every public execution path calls `check-amounts` and rejects gross input greater than or equal to `MAX-FRAME`, while the quote functions omit this check. A quote can therefore return a nonzero result for an input that execution deterministically rejects with `ERR-AMOUNT-TOO-LARGE`.
- Execution rejects swaps while the pool is paused with `ERR-POOL-PAUSED`, whereas quotes remain available and return otherwise executable-looking amounts.

The missing checks nevertheless contradict the contract's statement that quotes are "exactly what the swap functions would execute against current state."

Recommendation

Run `check-amounts` and the pool pause check in both quote functions before invoking the swap step, so deterministic pool-level acceptance conditions match the state-changing entry points.

[L-05] Swap Functions Lack Deadlines and Can Execute With Stale Price Limits

Location: clmm-router-v1.clar

Description None of the public swap entry points in `clmm-pool-v1` or `clmm-router-v1` accepts or enforces a transaction deadline. This affects the single-bin IoC functions, their FoK wrappers, the pool and router multi-bin functions, and the multi-pool `swap-routes` function.

A user normally derives `max-price`, `min-price`, and any minimum-output value from the pool state observed when constructing the transaction. Without a deadline, the signed transaction remains eligible for execution after those assumptions have become stale. In the meantime, swaps and liquidity changes can move the relevant bins, and governance can change the pool's fee rate.

The transaction may consequently execute much later at the edge of an outdated price or output tolerance even though the user would no longer consider the trade acceptable given the updated market state.

This can expose users to adverse market movements, fee updates, and delayed or intentionally withheld transaction submission.

Recommendation

Add a `deadline` parameter to every public swap function that does not already have one, including both IoC swaps and both route swaps in `clmm-pool-v1`, as well as both FoK swaps, both multi-bin router swaps, and `swap-routes` in `clmm-router-v1`.

Reject the call with a dedicated expiry error unless `stacks-block-time` is less than or equal to the supplied deadline.

[L-06] Reversed Token Ordering Allows Duplicate Pools for the Pair

Location: clmm-pool-v1.clar

Description Pool uniqueness is enforced using the ordered tuple `{ token-x, token-y, bin-step }`. Creating a pool for tokens `A` and `B` therefore occupies only the key `{ token-x: A, token-y: B, bin-step: step }`. A subsequent call with the same bin step and reversed token arguments uses `{ token-x: B, token-y: A, bin-step: step }`, which is a different map key and passes the duplicate-pool check.

The two pool IDs represent the same economic token pair and bin spacing but maintain completely independent bins, LP shares, fees, prices, and status. Reversing the tokens changes the price denomination and swap direction, but does not create a distinct underlying market. In particular, when the directional fees are equal, the second pool is only an inverted representation of the first pool.

This can fragment liquidity and trading volume between two equivalent deployments, increasing overall price impact.

`create-pool` is restricted to the DAO or an enabled extension, so an unprivileged user cannot create the reversed pool directly. The issue is therefore primarily a governance misconfiguration and integration risk.

Recommendation

Treat the token pair as unordered when enforcing pool uniqueness. Before creating `{ token-x: A, token-y: B, bin-step: step }`, check that neither that key nor `{ token-x: B, token-y: A, bin-step: step }` is already registered, and reject the creation if either exists.

[L-07] Just-in-Time Liquidity Can Capture Fees From Long-Term LPs

Location: clmm-pool-v1.clar

Description A liquidity provider can observe a pending swap, add liquidity to its target bins immediately before execution, and remove the position immediately afterward. Because the swap's LP fee rebate is distributed across the shares present during execution, the temporary provider receives part of the fee that would otherwise accrue to long-term LPs.

This behavior does not directly harm swappers and can improve their execution by temporarily deepening liquidity and reducing price impact. Nevertheless, persistent just-in-time liquidity may reduce the incentive to provide stable, long-term liquidity.

Recommendation

Just-in-time fee capture is common in liquidity pools and generally causes no material issue. For safety, consider implementing a configurable deposit fee that defaults to zero and can be activated only if observed just-in-time activity begins materially disincentivizing long-term LPs. Prefer distributing such a fee to existing LPs so it directly compensates the providers being diluted.

[L-08] Order Fragmentation Turns a Nominal Fee Rebate Into a Material Price Premium

Location: orderbook-core-v1.clar

Description `orderbook-core-v1` calculates and rounds the taker fee independently for every matched order. It then applies the maker rebate independently to each rounded fee. An untrusted maker can split one level into 256 valid minimum-size orders and, when the market has a full maker-fee rebate, receive one whole raw base unit of rebate for each match.

For every individual order, `match-step` computes the fee and rebate before logging the maker payout:

```
(fee (try! (mul-div-up taker-amount fee-rate (- ONE_8 fee-rate))))
(rebate (try! (mul-div fee (get rebate-rate s) ONE_8)))
...
(map-set fill-log n {
  ...
  amount: (+ taker-amount rebate)
  ... })
```

With `fee-rate = 10,000`, a one-unit `taker-amount` produces:

```
ceil(1 * 10,000 / (100,000,000 - 10,000)) = 1
```

instead of a fractional fee. At a full maker rebate, every one of those rounded units is transferred to the maker in `settle-fill`. The aggregate `fee-amount` field instead sums `fee - rebate`, so it reports zero in this configuration even though 256 fee units were charged to and paid out from the taker's input.

The per-call loss is bounded by the 256-fill work cap in raw base units.

Recommendation

Consider calculating the fee against the cumulative executed consideration rather than rounding it per order. Track the total maker consideration during a fill and make each order's fee the delta between the cumulative rounded fee before and after that order. The cumulative formula retains a single rounding unit for the entire call:

```
fee_delta = ceil((previous_paid + current_paid) * f / (1 - f)) - ceil
  (previous_paid * f / (1 - f))
```

Then derive the rebate from `fee_delta` and log the corresponding payout. The matcher must use the same cumulative fee calculation when determining how much of the next order fits within the taker's input budget.

[L-09] Signer-Set Changes Revive Expired Governance Proposals

Location: operators.clar

Description `operators::approve` treats a proposal created under an earlier signer-set epoch as a new proposal. It replaces the stored record with `fresh`, whose `opened-at` is the current `stacks-block-height`, before checking whether the approval window has expired:

```
(
  fresh{approvals:u0,
  opened-at:stacks-block-height,
  epoch:current,
  executed:false}
)
(stored (default-to fresh (map-get? proposals key)))
(state (if (or (get executed stored) (is-eq
  (get epoch stored) current)) stored fresh))
...
(asserts! (< stacks-block-height (+
  (get opened-at state) APPROVAL-WINDOW)) ERR-EXPIRED)
```

Both `set-operator` and `set-threshold` increment `epoch`. Consequently, after a proposal's approval window has expired, any later signer-set or threshold change lets a current operator submit a new first approval for the same proposal. The proposal then receives a new full `APPROVAL_WINDOW` rather than remaining permanently expired.

Recommendation

Make expiry terminal for a proposal principal. Preserve its original opening height across epochs, instead of resetting it. If reconsideration is needed, require a new proposal contract/principal instead of silently reopening the prior one.

[L-10] An Untrusted Intermediary Can Force User-Funded Orderbook and CLMM Trades

Location: orderbook-exchange-v1.clar clmm-pool-v1.clar clmm-router-v1.clar

Description The orderbook exchange identifies a user exclusively through `tx-sender`, which remains the original wallet throughout nested Clarity contract calls. It does not require `contract-caller` to be that user, nor does it require the immediate caller to be user-authorized. Consequently, an arbitrary contract that a user calls can invoke the exchange with attacker-selected order parameters and cause the core to escrow or spend that user's assets.

`create-limit-order` passes `tx-sender` to the core as the order maker:

```
(define-public (create-limit-order (market-id uint)
  (maker-asset-trait <ft-trait>) (price uint) (amount uint))

    (contract-call? .orderbook-core-v1 register-order market-id tx-sender maker-asset-trait price amount))
```

`orderbook-core-v1::register-order` then pulls `amount` from the same ambient sender and records the supplied maker:

```
(try! (transfer-in maker-asset-trait amount tx-sender))
(map-set orders order-id { maker: maker, ... })
```

The taker path has the same authorization gap. `orderbook-exchange-v1::fill-orders` calls the core without authenticating its immediate caller, and the core transfers the taker asset from `tx-sender`. It then settles that input to the maker selected by the supplied price levels, while the exchange forwards the acquired maker asset to `tx-sender`:

```
;; orderbook-core-v1.fill-orders
(try! (transfer-in taker-asset-trait amount tx-sender))
...
(try! (fold settle-fill fills ...))

;; settle-fill
(try! (transfer-out (get taker-asset-trait ctx) (get amount f) (get maker f)))
```

An attacker can therefore deploy a bait contract, place an attacker-controlled order, and induce a victim to call the bait contract. The bait contract can either create a sell order using the victim's tokens at an artificially low price, then let the attacker fill it, or fill the attacker's intentionally overpriced sell order using the victim's base asset. In the latter case the exchange has no user-provided `min-take`: the bait contract supplies `u0`, so the entire unwanted purchase settles atomically in the victim's transaction.

The same authorization gap applies to the CLMM pool's public swap entry points. `swap-x-for-y-ioc`, `swap-y-for-x-ioc`, `swap-x-for-y-route`, and `swap-y-for-x-route` take the swap parameters from `contract-caller`, but debit the input token from `tx-sender` and credit the output token to `tx-sender`. None authenticates the immediate caller:

```
;; clmm-pool-v1.swap-x-for-y-ioc
(try! (transfer-in token-x-trait (get token-x pool)
  (get native-x pool) in tx-sender))
(try! (transfer-out token-y-trait (get token-y pool)
  (get native-y pool) out tx-sender))
```

The stateless router compounds the issue: its public entry points forward the caller-selected swap configuration to the pool and deliberately execute the underlying calls with the same ambient `tx-sender`. Thus, a bait contract can call either the pool directly or the router and select the pool, tick or route, input amount, price limit, partial-fill choice, and minimum output for the victim.

Recommendation

Either require user-owned orderbook and CLMM operations to be invoked by their originating wallet, or provide explicit per-user delegation.

The simplest option is to add the following check before any user-owned operation that can escrow, spend, cancel, or swap assets:

```
(asserts! (is-eq contract-caller tx-sender) err-unauthorized-sender)
```

If composability through user-selected contracts is required, instead add a map keyed by user and whitelisted contract, for example `{ user: principal, contract: principal } -> bool`. Expose `whitelist-contract-for-user` and require it to begin with `tx-sender == contract-caller == user`, so no intermediary can set its own authorization using an ambient sender. Each user-owned orderbook entry point and each direct pool swap should then authorize only a direct call or a call where `contract-caller` is currently whitelisted for that `tx-sender`.

[L-11] Unbounded Market Creation Fragments Liquidity for Each Token Pair

Location: orderbook-core-v1.clar

Description `add-market` creates a fresh sequential `market-id` for every invocation. It stores the supplied tuple in `markets`, but it maintains no index from a token pair to a market ID and performs no duplicate-pair or duplicate-configuration check:

```
(define-data-var market-nonce uint u0)
(define-map markets uint { token: principal, base-token: principal, ... })

(define-public (add-market
  (market { token: principal, base-token: principal, ... }))
  (let ((market-id (+ (var-get market-nonce) u1)))
    ...
    (asserts! (not (is-eq (get token market)
      (get base-token market))) err-asset-mismatch)
    ...
    (map-set markets market-id market)
    (var-set market-nonce market-id)
    (ok market-id)))
```

A DAO proposal can therefore create arbitrarily many independent markets for the same economic pair, including identical `A/B` markets and reciprocal `A/B` and `B/A` markets with the same tick, minimum order, fee schedule, and fee recipient.

Each ID has an independent `orderbook` map, FIFO queues, maker escrow, managed orders, fees, and status. No matching, price discovery, or pair-level lookup combines them. This unnecessarily fragments the liquidity of a pair: a taker or integration that selects one ID can miss resting liquidity in another, and makers on the separate books lose queue depth and execution probability. Reciprocal markets are especially unnecessary for a conventional orderbook because a single canonical `A/B` market already supports both economic directions through its buy and sell sides; reversing the pair merely inverts the price unit.

Creation is restricted to the DAO or an enabled extension, so this is a governance configuration and integration risk rather than a permissionless issue. Nevertheless, the contract lacks the expected invariant that a token pair has one discoverable, consolidated orderbook.

Recommendation

Represent each unordered token pair by exactly one market ID. Select and document a canonical orientation at creation (for example, order the principals deterministically) and use a `market-id-by-pair` map keyed by that canonical pair. Reject a second creation for the pair regardless of token orientation or any other configuration fields. The canonical market's buy and sell sides should serve both trading directions.

Make the non-pair parameters updateable through versioned market configurations rather than by redefining the execution terms of open orders. On registration, associate each order with an immutable configuration version, or snapshot the relevant terms directly on the order. The immutable terms should include the raw price (already stored), tick/grid information used to interpret it, and the fee terms: taker fee and maker rebate, with the fee recipient being optionally saved based on intention.

`fill-orders` should calculate each order's fee and maker payout from that order's recorded configuration, not from the market's latest configuration. A configuration update then creates a

new version used only by newly registered orders. Existing orders remain visible and matchable at their stored prices even after a tick change, and cancellation must remain available for every version.

[L-12] Off-Chain Price Selection Cannot Guarantee Best-Price Execution

Location: orderbook-core-v1.clar orderbook-exchange-v1.clar

Description `orderbook-core-v1` stores a FIFO queue for each exact `(market-id, price, side)` key, but stores no ordered index of the non-empty price levels. The taker must therefore discover levels off chain and supply their exact raw prices to `fill-orders`:

```
(define-map orderbook
  { market-id: uint, price: uint, is-buy: bool }
  { head-order: uint, tail-order: uint })

(define-public (fill-orders ... (prices (list 200 uint)))
  ...
  (fold match-price prices ...))
```

`match-price` performs only an exact map lookup at each supplied price. It does not search for a better non-empty level, require the list to be complete or ordered, or compare a level's price with a taker limit. Thus the price list is a stale off-chain snapshot by the time its transaction is mined.

For example, a buyer obtains an off-chain quote showing no sell liquidity at price `A` and liquidity at a more expensive price `B`, and submits `prices = [B]`. Before the transaction executes, another maker posts a sell at `A`. The fill still looks up only `B`, skips the now-best ask at `A`, and can execute at the worse price. If its output remains at least `min-take`, the call succeeds; `min-take` protects an absolute amount, not best-price execution. The analogous case applies to a seller when a higher bid appears after the quote.

Submitting every current-grid tick between the current best price and the taker's limit can cover a newly created level within that narrow range, but it is not a general best-execution guarantee:

- the list is bounded to 200 prices;
- each absent price consumes one of the global 256 matching steps; and
- a sparse list outside that range still omits any better level created between quotation and execution; and

The result is that integrations must operate an accurate indexer and continually re-quote, yet still cannot guarantee that a fill uses the best liquidity present at execution. This creates avoidable price-quality loss: newly available better liquidity can be omitted while the worse, quoted level still satisfies `min-take`.

Recommendation

Consider moving best-orders discovery on-chain to support best-price guarantees.

Maintain an ordered linked list of non-empty price levels for each market side in addition to the existing FIFO order queue at each price. Each price-level node should retain its queue head and tail and add predecessor/successor prices. Keep separate lists for asks and bids: a token buyer starts from the lowest ask, while a token seller starts from the highest bid. Store a per-side best-level pointer for each market.

When a new order creates a price level, insert that level into the sorted side list. Since an on-chain contract cannot safely scan an unbounded list during registration, accept predecessor/successor insertion hints from the caller and verify on chain that they are adjacent, belong to the correct

market and side, and bracket the new price. When a fill or cancellation empties a level, unlink its price node atomically.

Remove the arbitrary `prices` list in fills and instead start from the optimal price for the taker with non-zero liquidity, walking the new linked list until the required amount is filled.

[L-13] Orders Lack Expiry Timestamp

Location: orderbook-core-v1.clar orderbook-exchange-v1.clar

Description The orderbook does not allow makers to specify when an order must stop being executable. `orderbook-core-v1::register-order` accepts a market, maker, asset, price, and amount, but no expiry timestamp:

```
(define-public (register-order (market-id uint) (maker principal)
  (maker-asset-trait <ft-trait>) (price uint) (amount uint))
```

The `orders` map likewise contains no expiry field. `match-step` calculates the executable amount from the order's remaining escrow, stored price, and taker budget without checking any time condition. Consequently, an unfilled remainder does not become ineligible through the passage of time; a maker must successfully cancel it to withdraw the quote.

Consider the following scenario:

1. Alice posts a sell order for 100 tokens at 10 base tokens each, intending to offer that price only until 12:00 o'clock.
2. The contract cannot record that cutoff. Alice must arrange for a separate cancellation transaction to execute by then.
3. Alice's cancellation is delayed, her automation fails, or she is offline. After 12:00, the external market price rises to 15.
4. A taker can still purchase the remaining tokens at 10 through the normal fill path. If a cancellation is pending, the fill can execute first.

The transaction respects Alice's stored price, but the protocol cannot enforce the intended duration of her offer. An off-chain timer or frontend expiry setting cannot provide that guarantee because cancellation itself depends on transaction inclusion and ordering.

Recommendation

Add a maker-selected absolute `expires-at` timestamp to orders and propagate it through all exchange creation paths. Allow execution only while the chain's current block timestamp is less than `expires-at`. Reject registration with an already-expired timestamp.

[L-14] An Untrusted Intermediary Can Force Vault Actions

Location: clmm-vault-v1.clar:199-269 clmm-vault-v1.clar:287-339

Description Every client-restricted vault operation authenticates only `tx-sender` and does not authenticate the immediate `contract-caller`. Clarity preserves the original transaction signer as `tx-sender` throughout nested contract calls, so any intermediary contract called by the client inherits the client's complete vault authority even though the client never called the vault or approved that intermediary as a vault delegate.

For example, `withdraw` and the manager-pause controls check only the ambient sender. An attacker can therefore deploy a bait contract and induce a vault client to call it. The bait can call `unpause-manager` after the client deliberately paused the manager, or call `withdraw` with `percent = ONE_8` and attacker-selected zero minimum outputs to unwind the client's complete managed position at an adverse time. The forced unwind can crystallize losses and manager fees, remove all deployed liquidity, and terminate the strategy without the client having requested a vault withdrawal. Asset postconditions cannot constrain the manager-pause state transition.

The vault hardcodes withdrawal payments to the configured client, so the `withdraw` call by itself cannot select an attacker as recipient, but this not prevent the intermediary from stealing the proceeds afterwards while the bait contract is still executing with the client as ambient `tx-sender`.

Recommendation

Require the immediate caller to be the configured role in addition to checking the ambient sender. Centralize the following client authorization and apply it to `deposit`, `withdraw`, `withdraw-ticks`, `pause-manager`, and `unpause-manager`:

```
(asserts! (and (is-eq tx-sender (get client cfg)) (is-eq contract-caller
  (get client cfg))) ERR-NOT-CLIENT)
```

Apply the equivalent condition to `add-liquidity`, `remove-liquidity`, `rebalance`, and `harvest` for the manager role.

If integrations require contracts to act for a client or manager, implement explicit per-role delegation instead of treating every contract called by that role as authorized.

[L-15] Historical Order Accounting Can Be Overflowed to Revert Managed-Order Fills

Location: orderbook-core-v1.clar orderbook-exchange-v1.clar

Description The exchange intends to catch errors raised while flipping managed-order proceeds so that a rejected flip becomes a residual instead of reverting the taker's fill.

However, `orderbook-core-v1::increase-order` updates both the live `remaining-amount` and the cumulative historical `amount` with native addition:

```
(map-set orders order-id (merge order {
  remaining-amount: (+ (get remaining-amount order) amount),
  amount: (+ (get amount order) amount)
}))
```

The live amount is constrained by current token custody and decreases when the order is filled, but the historical `amount` never decreases. Repeated fills and top-ups can therefore make it reach `2^128 - 1` while the live order still has room for another top-up. The next top-up raises a Clarity `ArithmeticOverflow` rather than returning a catchable error.

An attacker can initialize the managed sell leg with `q` target tokens plus a one-raw-unit sentinel. The sentinel is additional to the seed and exists only to keep the sell order ID open, because closing and recreating that leg would start a fresh historical `amount`. The attacker can then repeatedly perform the following round trip:

1. Flash-borrow enough base tokens to buy `q` target tokens from the managed sell leg, leaving its one-unit sentinel untouched. The base-token proceeds create or fund the managed buy leg.
2. Sell the same `q` target tokens into that buy leg.
3. `post-process-fill` receives the target-token proceeds and calls `core-fund` to add them back to the still-open sell leg. This restores its live balance to `q + 1` while increasing its cumulative historical `amount` by `q`. The received base tokens repay most of the flash loan, while the attacker supplies the small price-spread and fee shortfall.

The live liquidity moves between the attacker and the two legs, but each round permanently increases the sell leg's historical `amount`. Multiple rounds can be executed within one flash-loan transaction, while the accumulated value persists between transactions. A zero-fee or fully rebated market lets essentially the same target-token liquidity be recycled; otherwise, the attacker must replace the target-token fees that are not rebated.

Once the historical amount is close to `2^128 - 1`, the attacker leaves the managed sell leg open. A later taker filling the corresponding buy leg causes its proceeds to be flipped into the poisoned sell leg. This occurs while `orderbook-exchange-v1.post-process-fill` evaluates `(match (core-fund ...) ...)`; `match` can handle a returned error but cannot catch a runtime abort. The entire taker fill therefore rolls back. If the managed buy leg is the head of its price level, orders behind it cannot be reached until clients skip the level or the attacker cancels the managed order.

Consider a market between PAW and DAI. Using an illustrative PAW price snapshot of approximately `$1.729e-10`, a `$50,000` liquidity seed represents approximately `289.18` trillion PAW. Reaching `2^128 - 1` from that seed requires `1,176,696` additional round trips. An isolated batching wrapper can execute 72 complete round trips according to the read transaction limit. The

optimistic path therefore requires at least `16,343` flash-loan transactions before accounting for the lender's own execution cost.

In practice, the attack is not realistic currently due to fees and transaction costs. Nevertheless, cumulative informational accounting should not be allowed to introduce an uncatchable runtime abort.

Recommendation

Saturate the informational value `order.amount` at `MAX-U128` without evaluating a potentially overflowing addition first. For example, compare the increment against `(- MAX-U128 (get amount order))` and return `MAX-U128` when it exceeds the remaining capacity.

Alternatively, since this field is not used for matching, custody, cancellation, or settlement, consider removing the field entirely if it's not required for off-chain processes.

[L-16] Vault Price Bounds Do Not Protect the Fee-Inclusive Price of LP Shares

Location: clmm-vault-v1.clar

Description The pool protects liquidity additions with both price bounds and `min-shares`. The vault forwards the manager's `min-price` and `max-price`, but always passes zero as `min-shares`:

```
(contract-call? .clmm-pool-v1 add-to-position
  tx ty pool-id tick dx dy min-price max-price u0)
```

The price bounds constrain the bin's final marginal X/Y price, not the fee-inclusive price the vault pays per minted LP share. Consequently, if governance increases the pool fee after the manager prepares an `add-liquidity` or `rebalance` transaction but before it is mined, the transaction can mint fewer shares than quoted without violating its price bounds. With a `100%` fee rebate, the fee remains in the bin and is credited to the existing LPs before the vault's shares are minted, leaving the final reserves and marginal X/Y price unchanged.

For example, the PoC quotes a `100,000 X` addition at a `0%` fee and receives `13,125,817` shares. Governance then raises `fee-rate-x` to `2%`. The same transaction succeeds, charges a `1,077 X` fee, and mints only `12,984,451` shares. The effective price therefore increases from `0.0076185734 X` to `0.0077015193 X` per share, approximately `1.0887%`, while the quoted and executed marginal X/Y prices remain identical and inside an approximately `1.73e-18`-wide relative price interval. This affects both `add-liquidity` and additions performed by `rebalance`, as both use the same helper that hardcodes `min-shares` to zero.

Recommendation

Add `min-shares` to `add-liquidity` and to each addition accepted by `rebalance`, and forward it to `add-to-position` instead of passing zero. This implicitly bounds the fee-inclusive price paid per LP share.

8.4. QA Findings

[QA-01] Variable Stacks Block Heights Are Used as Wall-Clock Time

Location: operators.clar:71-78

Description The production codebase uses `stacks-block-height` in `operators.clar` to record `opened-at` and expire approvals after `u241920` blocks, which is documented as approximately two weeks only by assuming a constant five-second block interval.

Stacks block production is not a fixed clock, so faster production shortens the period available to coordinate governance while slower production keeps stale approvals live longer than the intended two weeks; this can cause unexpected governance liveness or stale-proposal risk even though execution still requires the configured threshold.

Clarity 5 supports `stacks-block-time`, which returns the current block's Unix timestamp in seconds and expresses this policy directly.

Recommendation

Replace the height read with `stacks-block-time`, rename the constant to `APPROVAL-WINDOW-SECONDS`, and set it to the number of seconds in two weeks.

[QA-02] Depositor Migration Leaves the Old Vault Accepting Undeployable Deposits

Location: clmm-pool-v1.clar:313-319 clmm-vault-v1.clar:199-209 clmm-vault-v1.clar:287-309

Description Governance can change a private pool's exclusive depositor with `set-pool-depositor`. This is intended to support moving a pool to a new vault, but the operation only updates the pool record. It does not notify or retire the old vault, and the old vault does not revalidate that it remains the pool's depositor when accepting client deposits.

Consequently, after governance assigns the pool to a different principal, the old vault remains initialized and its `deposit` entry point continues pulling tokens from the configured client and crediting them to `idle`. The transaction succeeds and emits an ordinary deposit event even though the vault can no longer place those funds into any bin.

Every later `add-liquidity` call reaches the pool with the old vault as `tx-sender` and fails with `ERR-NOT-DEPOSITOR` (`u1006`). A `rebalance` containing both removals and additions fails for the same reason.

This turns the old vault into an implicit exit-only vault without recording that state in its configuration, so new client deposits succeed but cannot be deployed.

Recommendation

Introduce an explicit vault retirement state and make pool-to-vault migration a coordinated operation. Before changing a vault-backed pool's depositor, governance should retire the old vault. A retired vault should reject new deposits and manager placements while continuing to allow removals, harvests, and client withdrawals.

[QA-03] Heterogeneous LP Share Totals Have No Economic Meaning

Location: clmm-pool-v1.clar clmm-vault-v1.clar clmm-lp-token-v1.clar

Description Each CLMM bin is represented by a distinct SIP-013 token ID derived from `(pool-id, tick)`. An amount of shares in one bin therefore has unit `LPShare[pool-id, tick]` and cannot be added meaningfully to an amount from another bin: the bins can have different prices, reserves, total supplies, and values per share. Nevertheless, the pool, vault, and LP-token contracts construct raw numerical totals across these heterogeneous token IDs.

The production code performs these sums in every one of the following places:

1. `clmm-pool-v1::reduce-positions-iter` adds the per-bin `shares` burned by a batched withdrawal into `acc.shares`.
2. `clmm-vault-v1::burn-iter` passes each bin's `sh` to `add-shares`, producing the same cross-bin total for client withdrawals, manager removals, and the removal portion of rebalances.
3. `clmm-vault-v1::harvest-iter` passes each bin's harvested `sh` to `add-shares`, producing a cross-bin total for `harvest` and `set-manager-fee`.
4. `clmm-lp-token-v1` adds and subtracts amounts from different token IDs in each owner's `overall-balances` wide counter during mint, transfer, and burn.
5. `clmm-lp-token-v1` similarly aggregates every token ID into the global `overall-supply` wide counter.

For example, the pool's batch accumulator performs the dimensional equivalent of:

```
LPShare[pool, tick A] + LPShare[pool, tick B]
```

The result is only a heterogeneous raw count. It cannot be redeemed, converted into reserves, compared economically with another aggregate, or interpreted as the size or value of an LP position.

The current internal call graph does not use any of these sums to determine economic settlement.

Recommendation

Do not expose a heterogeneous sum under the unqualified name `shares`. For batched pool and vault operations, either omit the aggregate or return exact per-bin `{pool-id, tick, shares}` results. The existing per-bin events already provide the information needed by indexers.

Replace `pool-burn`'s zero check with an explicit `has-burn` boolean or a check that the filtered item list is nonempty. If cross-contract verification of the burns is required, verify per-bin results rather than comparing two identically capped heterogeneous totals.

Document the overall LP getters for SIP-013 as capped raw-count summaries with no economic interpretation.

[QA-04] Manager Rotation Pays the Outgoing Manager's Accrued Fees to the Incoming Manager

Location: clmm-vault-v1.clar

Description The vault does not associate accrued manager fees with the manager or fee recipient that earned them. Manager fees realized during liquidity removals are accumulated in the single global `owed` record, while fees on liquidity that remains deployed continue to exist as unharvested manager shares. Neither `set-manager` nor `set-fee-recipient` settles or checkpoints either form of accrual before replacing the corresponding role.

Consequently, consider a vault whose manager and fee recipient are both A. After A earns fees, governance rotates both roles to B without first harvesting. B can immediately call `harvest`, which calculates the manager shares from all growth since the previous checkpoint, adds every token already present in `owed`, and transfers the combined amount to the current `fee-recipient`, B:

```
(let (
    (w (try! (fold harvest-iter (var-get held) ...)))
    (o (var-get owed))
    (dx (+ (get x o) (get ox w)))
    (dy (+ (get y o) (get oy w))))
  ...
  (var-set owed { x: u0, y: u0 })
  (try! (pay token-x-trait (get token-x cfg) (get native-x cfg) dx
    (get fee-recipient cfg)))
  (try! (pay token-y-trait (get token-y cfg) (get native-y cfg) dy
    (get fee-recipient cfg))))
```

The incoming manager therefore receives fees earned during the outgoing manager's tenure. The already-realized amounts in `owed` make the misallocation possible even if no further pool growth occurs between the rotation and the harvest. Unharvested fees from prior growth are redirected in the same way.

`set-manager-fee` already avoids the analogous problem for fee-rate changes by calling `do-harvest` under the old configuration before installing the new rate. Role rotation lacks the same epoch boundary.

Recommendation

Settle all accrued fees to the old fee recipient before changing the payout recipient. Prefer a single atomic manager-rotation function that accepts both token traits, harvests using the old configuration, and only then updates `manager` and `fee-recipient`.

[QA-05] A Paused Manager Continues Earning Fees on Subsequent Liquidity Growth

Location: clmm-vault-v1.clar

Description `pause-manager` is intended to let the client suspend active management of its vault. The resulting `manager-paused` flag is checked by `add-liquidity`, `remove-liquidity`, and `rebalance`, but it is not incorporated into the fee accounting. Existing positions remain deployed, and `split-shares` continues applying the configured `manager-fee` to all liquidity-per-share growth without distinguishing whether that growth occurred before or during the pause.

The paused manager can realize this new entitlement immediately since `harvest` deliberately ignores `manager-paused`:

```
(define-public (harvest (token-x-trait <ft-trait>) (token-y-trait <ft-trait>))
  (let ((cfg (try! (load token-x-trait token-y-trait))))
    (asserts! (is-eq tx-sender (get manager cfg)) ERR-NOT-MANAGER)
    (let ((w (try! (do-harvest token-x-trait token-y-trait cfg false))))
      ...)))
```

If the client withdraws while management is paused, `withdraw` and `withdraw-ticks` still pass the unchanged `manager-fee` into `burn`. The manager's share of growth during the pause is then removed from the client's proceeds and accumulated in `owed` for a later harvest.

The pause therefore suspends the manager's operational authority without suspending its compensation. A client using the emergency brake continues paying performance fees to an inactive manager until governance separately sets the fee to zero or the client exits the position.

Recommendation

Make pause and unpause explicit fee-accounting epoch boundaries. On pause, preserve the manager's entitlement accrued up to that point for the existing fee recipient, checkpoint every position, and disable fee accrual before setting manager-paused. Do not require immediate payment to the recipient, because a reverting recipient such as due to being blacklisted must not be able to block the client's emergency action.

On reactivation, checkpoint the positions again before enabling the configured fee so that paused-period growth is not charged retroactively. Because a fee-free pause would otherwise let the client retain the manager's deployed strategy while selectively avoiding its performance fee, treat reactivation as a governed lifecycle transition: the client may pause unilaterally, but only the DAO should be able to unpause after confirming the manager, fee recipient, and fee terms.

If unilateral client unpausing is retained, the protocol must explicitly accept and document that clients can keep manager-created positions deployed during fee-free periods. Monitoring can expose such behavior but does not prevent it.

[QA-06] Dust Reduction Burns LP Shares Without Paying Out

Location: clmm-pool-v1.clar

Description `reduce-position` computes the share burn first and the payouts second, and only the payouts are guarded:

```
shares = floor(balance-of-sender * percent / 1e8)   ;; asserted > 0
out-x  = floor(balance-x * shares / total-supply)   ;; no floor guard
out-y  = floor(balance-y * shares / total-supply)   ;; no floor guard
```

When the caller's share burn is small relative to the bin's total supply, both payouts can floor to zero. The burn still executes, `total-supply` is reduced, the bin's reserves are unchanged, and the `(and (> out-x u0) ...)` / `(and (> out-y u0) ...)` transfer guards skip both payments. The caller loses the share and receives nothing; its sub-raw-unit claim on the reserves accrues to the remaining holders.

The batched `reduce-positions` has the same geometry: it skips a bin whose *share burn* rounds to zero, but not a bin whose burn is positive while both payouts round to zero, so a listed bin can likewise be debited shares for nothing inside an otherwise useful batch.

As a result, such a transaction would act as a small donation to the pool.

Recommendation

Refuse to destroy shares for nothing: after computing both payouts, require that at least one is positive

```
(asserts! (or (> out-x u0) (> out-y u0)) ERR-ZERO-PERCENT)
```

and apply the same check per bin in `reduce-positions-iter`, which currently skips only zero-share bins.