



Aftermath Perpetuals

Security Assessment

Prepared for
Aftermath Finance

Michał Bochnak
Sangsoo Kang

Prepared by
Otter Audits, LLC

Prepared on
August 12th, 2026

embe221ed@osec.io
sangsoo@osec.io

Table of Contents

1. Executive Summary		3
1.1. Overview		3
1.2. Key Findings		3
2. Scope		4
3. Repositories		5
4. Findings		6
4.1. Findings summary		6
5. Advisories		7
<u>OS-AMP-ADV-00</u>	● Collateral Rounding Triggers Full Liquidation	8
<u>OS-AMP-ADV-01</u>	● Round-Trip Loss Realization Bypasses Collateral Haircut Margin	11
<u>OS-AMP-ADV-02</u>	● Positive Funding Inflates Force-Withdraw Targets	14
<u>OS-AMP-ADV-03</u>	● Aggregate Force-Withdraw Dust Can Revert Redemptions	16
<u>OS-AMP-ADV-04</u>	● Unsafe Position Downcasts Block Critical Operations	18
<u>OS-AMP-ADV-05</u>	● Favorable Fill Haircut Lets Bad-Debt Makers Fill	19
<u>OS-AMP-ADV-06</u>	● Over-Haircutted Maker Guard Cancels Solvent Depth	22
<u>OS-AMP-ADV-07</u>	● Floored Mark Triggers Full Liquidation	26
<u>OS-AMP-ADV-08</u>	● Force Withdraw Uses Divergent Collateral Prices	29
<u>OS-AMP-ADV-09</u>	● Deposits Receive Free Bad-Debt Recovery Option	31
<u>OS-AMP-ADV-10</u>	● Settled Markets Remain In Vault Membership	32
<u>OS-AMP-ADV-11</u>	● Closed Market Withdrawals Use Oracles	33
<u>OS-AMP-ADV-12</u>	● Market Closure Uses Stale Funding	34
<u>OS-AMP-ADV-13</u>	● Settlement Ordering Blocks Bad-Debt Socialization	36
<u>OS-AMP-ADV-14</u>	● Standalone Stop Orders Fail at Exact Trigger Price	37
<u>OS-AMP-ADV-15</u>	● Post-Resume Funding Charges Pause Interval	38
<u>OS-AMP-ADV-16</u>	● Cross-Account Assistant Cap Revocation	39
<u>OS-AMP-ADV-17</u>	● Self-Cancellation Enables Undercollateralized IOC Positions	40
<u>OS-AMP-ADV-18</u>	● Pending Orders Let Under-IMR Liquidators Acquire Live Positions	41
<u>OS-AMP-ADV-19</u>	● Stale Pending Orders Can Be Reposted Into Fresh Under-Margined Exposure	43
6. Suggestions		45
<u>OS-AMP-SUG-00</u>	● Default Minimum Order Bypasses Registry	46

<u>OS-AMP-SUG-01</u>	● Funding Receivables Can Hide ADL Bad Debt	49
<u>OS-AMP-SUG-02</u>	● Insurance Fee Pre-Transfer Can Block Bad-Debt Liquidations	52
<u>OS-AMP-SUG-03</u>	● Liquidation Fees Can Exceed Maintenance Margin	55
<u>OS-AMP-SUG-04</u>	● Resume Has No Cancel-Only Grace	58
<u>OS-AMP-SUG-05</u>	● Vendor Assistants Can Escalate To Perpetuals Vendor Admin	62
<u>OS-AMP-SUG-06</u>	● Socialized Bad Debt Can Be Underpaid After Collateral Price Moves	63
<u>OS-AMP-SUG-07</u>	● Asynchronous Funding Conversion Enables Collateral Arbitrage	65
<u>OS-AMP-SUG-08</u>	● Realized PnL FX Arbitrage	68
<u>OS-AMP-SUG-09</u>	● Cleanup-Only Standalone Stops Can Allocate Collateral	70
<u>OS-AMP-SUG-10</u>	● Cleanup-Only Stop Executions Can Burn Tickets Without Any Fill	72
<u>OS-AMP-SUG-11</u>	● Empty TWAP Attempts Can Allocate User Collateral	74
<u>OS-AMP-SUG-12</u>	● TWAP Fresh Chunks Bypass Execution Cap	77
<u>OS-AMP-SUG-13</u>	● Stop Executors Can Add Uncommitted Priority Taker Fees	79
<u>OS-AMP-SUG-14</u>	● TWAP Executors Can Force Priority Taker Fees	82
<u>OS-AMP-SUG-15</u>	● Stop Order Executions Bypass Vault Pending Order Cap	83
<u>OS-AMP-SUG-16</u>	● Frozen Oracles Remain Temporarily Consumable	84
<u>OS-AMP-SUG-17</u>	● Manipulated Book Marks Underfund New Positions	85
<u>OS-AMP-SUG-18</u>	● Stale Pre-Fill Marks Permit Maker Insolvency	86
<u>OS-AMP-SUG-19</u>	● Endpoint Quote Manipulation Distorts Funding Intervals	87
A. Vulnerability rating scale		88
B. Procedure		89

1. Executive Summary

1.1. Overview

Aftermath Finance engaged OtterSec to conduct a security assessment of the `perpetuals-v2` and `market-making-vault` programs. This assessment was conducted between May 19th and August 11th, 2026. For information on our auditing methodology, refer to [Appendix B](#).

1.2. Key Findings

We produced 40 findings throughout this audit engagement.

Among the most impactful results, we identified that partial liquidations can be escalated into unnecessary full liquidations due solely to collateral-rounding drift, causing users to lose the rest of their position and pay avoidable extra liquidation and insurance fees (● [OS-AMP-ADV-00](#)).

We also found that a zero-price-impact close-and-reopen sequence can improve computed margin under collateral haircuts without improving real economic exposure, allowing under-margined positions to remain open and potentially withdraw collateral they should not free up (● [OS-AMP-ADV-01](#)). In the vault flow, positive unsettled funding can inflate force-withdraw margin targets, letting withdrawing LPs force materially larger position closures than required or making otherwise-valid withdrawals fail (● [OS-AMP-ADV-02](#)).

Additionally, once a settled market remains stranded in `Vault.ch_ids`, subsequent deposits and withdrawals can become unfinishable, effectively freezing user exits and core vault operations until manual intervention (● [OS-AMP-ADV-10](#)). We further found that settlement ordering can reduce open interest before later bad-debt positions are processed, which can block or weaken bad-debt socialization exactly when it is needed most during market closure (● [OS-AMP-ADV-13](#)).

Finally, book-mark manipulation can understate the margin needed for newly opened positions because sessions cache a stale manipulated mark, allowing positions to be opened with too little collateral and pushing resulting losses toward the insurance fund or counterparties (● [OS-AMP-SUG-17](#)).

2. Scope

`perpetuals-v2` is an on-chain perpetual futures exchange that manages leveraged positions, order execution, margining, and liquidations. It integrates oracle aggregation, risk limits, and an insurance fund to maintain pricing integrity and system solvency.

`market-making-vault` manages pooled user collateral for the market-making vault, tracking balances, LP supply, and ownership while enforcing lock periods and accounting invariants.

`oracle-aggregator` is an on-chain price aggregation and validation layer used by the Perpetuals and Vault systems.

`vendor` manages who is allowed to register and administer vendors, and it keeps each vendor's profile information organized and up to date. It also supports controlled updates, revocation, and event tracking for changes across the system.

`ordered-map` is a Move package for an ordered key-value map built for on-chain use, with focus on efficient inserts, lookups, removals, and iteration. It also includes a compact optional-value utility to reduce overhead

`ifixed` implements signed 18-decimal fixed-point arithmetic in Move on top of u256.

`enum-option` is a compact library for handling values that may or may not be present, with a focus on lower cost and simpler usage than the standard approach.

`control-flow` focuses on making branching and decision-making logic more efficient by removing unnecessary overhead.

3. Repositories

The source code was delivered to us in Git repositories at:

1. <https://github.com/AftermathFinance/move-iperps>.
2. <https://github.com/AftermathFinance/move-market-making-vault>.
3. <https://github.com/AftermathFinance/move-oracle-aggregator>.
4. <https://github.com/AftermathFinance/move-vendor>.
5. <https://github.com/AftermathFinance/move-authority-cap>.
6. <https://github.com/AftermathFinance/move-ordered-map>.
7. <https://github.com/AftermathFinance/move-ifixed>.
8. <https://github.com/AftermathFinance/move-enum-option>.
9. <https://github.com/AftermathFinance/move-control-flow>.

The audit was performed against the respective commits:

1. [fc403cc](#).
2. [82f4f91](#).
3. [8074118](#).
4. [bfeb538](#).
5. [9723070](#).
6. [44d404c](#).
7. [198c4cd](#).
8. [58fbad4](#).
9. [22f0f91](#).

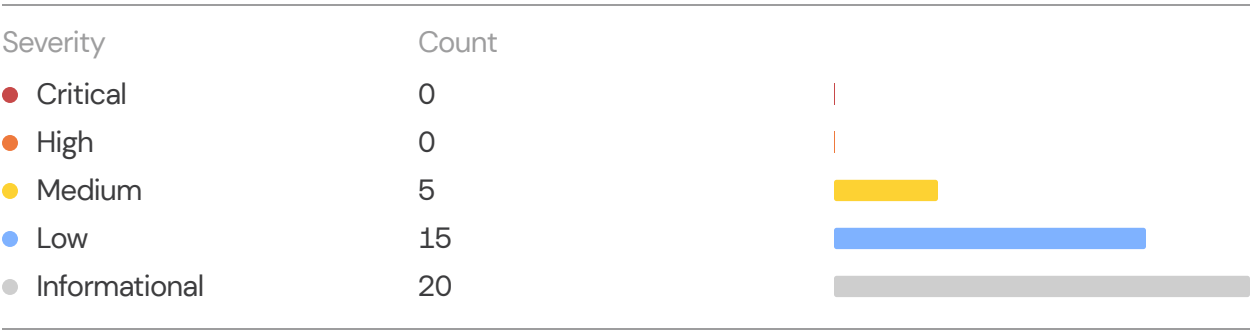
4. Findings

Findings were rated using the criteria in [Appendix A](#).

We split the findings into **advisories** and **suggestions**. Advisories have an immediate impact and should be remediated as soon as possible. Suggestions do not have an immediate impact but will aid in mitigating future vulnerabilities.

Items are tracked with a stable identifier: **OS-AMP-ADV-*** for advisories and **OS-AMP-SUG-*** for suggestions.

4.1. Findings summary



5. Advisories

Here, we present a technical analysis of the 20 vulnerabilities we identified during our audit. These vulnerabilities have **immediate** security implications, and we recommend remediation as soon as possible.

ID	Severity	Status	Description
OS-AMP-ADV-00	● Medium	✓ Resolved	Collateral Rounding Triggers Full Liquidation
OS-AMP-ADV-01	● Medium	✓ Resolved	Round-Trip Loss Realization Bypasses Collateral Hair-cut Margin
OS-AMP-ADV-02	● Medium	✓ Resolved	Positive Funding Inflates Force-Withdraw Targets
OS-AMP-ADV-03	● Medium	✓ Resolved	Aggregate Force-Withdraw Dust Can Revert Redemptions
OS-AMP-ADV-04	● Medium	✓ Resolved	Unsafe Position Downcasts Block Critical Operations
OS-AMP-ADV-05	● Low	✓ Resolved	Favorable Fill Haircut Lets Bad-Debt Makers Fill
OS-AMP-ADV-06	● Low	✓ Resolved	Over-Haircutted Maker Guard Cancels Solvent Depth
OS-AMP-ADV-07	● Low	✓ Resolved	Floored Mark Triggers Full Liquidation
OS-AMP-ADV-08	● Low	✓ Resolved	Force Withdraw Uses Divergent Collateral Prices
OS-AMP-ADV-09	● Low	✓ Resolved	Deposits Receive Free Bad-Debt Recovery Option
OS-AMP-ADV-10	● Low	✓ Resolved	Settled Markets Remain In Vault Membership
OS-AMP-ADV-11	● Low	✓ Resolved	Closed Market Withdrawals Use Oracles
OS-AMP-ADV-12	● Low	✓ Resolved	Market Closure Uses Stale Funding
OS-AMP-ADV-13	● Low	✓ Resolved	Settlement Ordering Blocks Bad-Debt Socialization
OS-AMP-ADV-14	● Low	✓ Resolved	Standalone Stop Orders Fail at Exact Trigger Price
OS-AMP-ADV-15	● Low	✓ Resolved	Post-Resume Funding Charges Pause Interval
OS-AMP-ADV-16	● Low	✓ Resolved	Cross-Account Assistant Cap Revocation
OS-AMP-ADV-17	● Low	✓ Resolved	Self-Cancellation Enables Undercollateralized IOC Positions
OS-AMP-ADV-18	● Low	✓ Resolved	Pending Orders Let Under-IMR Liquidators Acquire Live Positions
OS-AMP-ADV-19	● Low	✓ Resolved	Stale Pending Orders Can Be Reposted Into Fresh Under-Margined Exposure

Collateral Rounding Triggers Full Liquidation

ID	Severity	Status
OS-AMP-ADV-00	● Medium	✓ Resolved

Description

The liquidation flow first asks `compute_liquidation_size_and_mode` to calculate a partial close size intended to bring the liqee back to the initial margin requirement. That sizing routine models the result with USD-denominated PnL and fee arithmetic and comments that collateral changes by the realized PnL minus fees divided by the collateral price.

The state mutation then follows a separate path in `reduce_liquidated_position`. After applying the base and quote deltas to the position, it computes insurance and liquidation fees from the liquidated quote amount, subtracts those fees from the realized PnL, and passes the resulting signed USD change into the position collateral conversion helper, as shown in `reduce_liquidated_position`.

packages/perpetuals/sources/clearing_house.move

```

1 public(package) fun reduce_liquidated_position([...]): (u256, u256, bool, u256, u256,
  u256, u256, u256, u256) {
2     [...]
3     // Update collateral for liqee
4     let insurance_fund_fees = ifixed::mul(insurance_fund_fee, quote_asset_delta);
5     let liquidation_fees = ifixed::mul(liquidation_fee, quote_asset_delta);
6     let total_fees = ifixed::add(liquidation_fees, insurance_fund_fees);
7     let collateral_change = ifixed::sub(pnl_to_realize, total_fees);
8     position.add_to_collateral_usd(
9         collateral_change,
10        collateral_price
11    );
12    [...]
13 }
```

`add_to_collateral_usd` performs the USD-to-collateral conversion by dividing the signed USD delta by the collateral price before adding it to the stored collateral balance. For a losing liquidation slice, this signed division can round the collateral debit against the liqee, so the mutated collateral balance may be slightly lower than the continuous arithmetic used by the sizing routine, as shown in `add_to_collateral_usd`.

packages/perpetuals/sources/position.move

```

1 public(package) fun add_to_collateral_usd(
2     position: &mut Position,
3     fixed_usd: u256,
4     collateral_price: u256
```

```

5 | ): u256 {
6 |     let fixed = ifixed::div(fixed_usd, collateral_price);
7 |     position.collateral = ifixed::add(position.collateral, fixed);
8 |     position.collateral

```

Immediately after the partial reduction, the liquidation path compares the post-reduction margin with the post-reduction requirement. If the account remains unhealthy for any reason, it calls `reduce_liquidated_position` again with the remaining absolute base amount, which closes the rest of the position instead of sizing only the rounding deficit, as shown in the unhealthy-position fallback branch in `settle_liquidated_position`.

packages/perpetuals/sources/clearing_house.move

```

3576 fun settle_liquidated_position<T>([...]) {
3577     [...]
3578     // If the position is still unhealthy for any reason, fully liquidate
3579     if (ifixed::less_than(margin_after, margin_req_after)) {
3580         // Abort if specified.
3581         // This is just a sanity assert intended to catch eventual implementation
3582         // errors.
3583         assert!(!constants::abort_on_unhealthy_liquidation!(),
3584             liquidated_position_still_unhealthy!());
3585
3586         let (base_asset_amount, _q) = position.base_and_quote_amounts();
3587
3588         let (
3589             _margin_after,
3590             _margin_req_after,
3591             _is_liqee_long,
3592             more_base_asset_delta,
3593             more_quote_asset_delta,
3594             more_pnl_to_realize,
3595             more_liquidation_fees,
3596             more_insurance_fund_fees,
3597             more_bad_debt,
3598             more_open_interest,
3599             ) = reduce_liquidated_position(
3600                 position,
3601                 ifixed::to_balance(ifixed::abs(base_asset_amount),
3602                     constants::b9_scaling!()),
3603                 mark_price,
3604                 hot_potato.collateral_price,
3605                 insurance_fund_fee,
3606                 liquidation_fee,
3607                 margin_ratio_initial,
3608                 collateral_haircut

```

```

3606         );[...]
3607     };[...]
3608 }

```

As a result, a partial liquidation that was sized to leave the account at or just above IMR can be converted into a full liquidation because of a sub-USD collateral rounding shortfall. The liqee may lose the remaining exposure and pay avoidable liquidation and insurance fees, and any additional bad debt is handled by the same insurance-fund or socialized-loss path already present in liquidation.

Proof of Concept

1. Use a market with a B9-aligned mark price of `0.000100000000000000`, collateral price `100000`, IMR `10%`, MMR `5%`, liquidation fee `1%`, and insurance fee `1%`.
2. A long account holds `18,000,000,000` base and `2.7` collateral units. Its quote notional is `1,980,000.0000000000000250000`, reachable by opening almost all size at `0.000110000` and `0.00025` base at one tick higher.
3. The account margin is `89999.99999999999750000`, below the MMR requirement of `90000.000000000000000000`, so it is liquidatable.

`compute_liquidation_size_and_mode` returns `alpha = 0.625000000000000002` after its rounded-up division, giving `alpha * |b| = 11,250,000,000.000000036` base. `clip_size_to_liquidate` then adds one B9 atom, so the size actually liquidated is `11,250,000,000.000000037`. Because the mark is already B9-aligned, the execution quote for this slice is `1,125,000.0000000000003700000`.

1. The slice realizes a loss of `112,500.000000000000526251` PnL and `22,500.00000000000074000` total fees, so `collateral_change = -135,000.000000000000600251`.
2. Without collateral-division rounding, the first partial liquidation would leave margin `67499.99999999999676000`, above the IMR requirement `67499.99999999999630000`.
3. The implementation calls `add_to_collateral_usd`, which rounds the negative collateral delta to `1.350000000000000007` collateral units. Re-valued at the same collateral price, the liqee's margin becomes `67499.99999999999576251`, below IMR by `0.00000000000053749`.
4. The fallback branch then liquidates the remaining `674,999.99999999996300000` USD notional. At a `1%` liquidation fee and `1%` insurance fee, that avoidable fallback charges about `6,750` USD to the liqee for each fee bucket and gives the liquidator the extra liquidation fee.

Remediation

Make liquidation sizing use the same rounded accounting as the mutation path. Preview the exact post-reduction position update, convert `pn1 - fees` through the same `add_to_collateral_usd` division semantics, and compare the rounded post-state against the margin requirement before deciding the liquidation size is sufficient.

Patch

Resolved in [f5c42ad](#) and [0800431](#).

Round-Trip Loss Realization Bypasses Collateral Haircut Margin

ID	Severity	Status
OS-AMP-ADV-01	● Medium	✓ Resolved

Description

The position margin calculation is path-dependent when a market uses a nonzero `collateral_haircut`. The market parameters define the haircut as a discount on allocated collateral, and `position.move` applies that discount only to non-negative collateral. In `compute_margin_and_requirement`, the discounted collateral value is then added to full unrealized PnL, so an unrealized loss reduces margin dollar-for-dollar while the same loss, once realized into collateral, is only reflected through the haircut-discounted collateral balance.

packages/perpetuals/sources/position.move

```

569 public fun compute_margin_and_requirement(
570     position: &Position,
571     collateral_price: u256,
572     mark_price: u256,
573     margin_ratio: u256,
574     collateral_haircut: u256
575 ): (u256, u256) {
576     // Calculate pnl and min_margin_in_position
577     let pnl = unrealized_pnl(position, mark_price);
578     let min_margin = margin_requirement(
579         position,
580         mark_price,
581         margin_ratio,
582     );
583
584     let collateral_usd = effective_collateral_usd_value(position, collateral_price,
585         collateral_haircut);
586
587     let margin = ifixed::add(collateral_usd, pnl);
588
589     (margin, min_margin)
590 }
```

This difference means a close-and-reopen sequence can increase computed margin without changing the trader's raw equity or final directional exposure. Before realization, positive collateral is discounted and the unrealized loss is subtracted at full value. After realizing the loss, collateral is reduced by that loss and the reopened position starts with no unrealized PnL at the current price, leaving the realized loss subject only to the collateral haircut in subsequent margin math.

The taker session flow allows both sides of a round trip to be accumulated before final settlement. The public session API permits repeated `place_limit_order` and `place_market_order` calls before

`end_session`, and `process_fill_maker` records taker fills on both the ask and bid sides into the same `SessionSummary`, as shown by the session aggregation logic.

```
packages/perpetuals/sources/clearing_house.move
1 fun process_fill_maker<T>([...]): (u64, bool, events::FilledMakerOrder, u256) {
2   [...]
3   let (pnl, fees, integrator_fee_paid_usd, maker_integrator_id, open_interest) = if
    (taker_size_matched != 0) {
4     [...]
5     // Aggregators for taker's amounts
6     if (order_is_ask) {
7       session_summary.base_filled_bid = ifixed::add(
8         session_summary.base_filled_bid, base_asset_delta
9       );
10      session_summary.quote_filled_bid = ifixed::add(
11        session_summary.quote_filled_bid, quote_asset_delta
12      );
13    } else {
14      session_summary.base_filled_ask = ifixed::add(
15        session_summary.base_filled_ask, base_asset_delta
16      );
17      session_summary.quote_filled_ask = ifixed::add(
18        session_summary.quote_filled_ask, quote_asset_delta
19      );
20    };
21    [...]
22  }[...]
23 }
```

When the session is settled, `process_fill_taker` applies all accumulated ask fills before all accumulated bid fills. It calls `add_base_to_position` for the ask side and then for the bid side, sums the realized PnL, and converts the net result into collateral through `add_to_collateral_usd`. For a losing long, the ask portion can close the position and realize the loss, while the bid portion can reopen long exposure at the current price in the same session.

```
packages/perpetuals/sources/clearing_house.move
1 fun process_fill_taker([...]): (u256, u256, u256) {
2   [...]
3   if (session_summary.base_filled_ask != 0) {
4     (pnl_ask, base) = position.add_base_to_position(
5       constants::ask!(),
6       session_summary.base_filled_ask,
7       session_summary.quote_filled_ask
8     );
```

```

9      };
10
11      if (session_summary.base_filled_bid != 0) {
12          (pnl_bid, base) = position.add_base_to_position(
13              constants::bid!(),
14              session_summary.base_filled_bid,
15              session_summary.quote_filled_bid
16          );
17      };
18      [...]
19  }

```

After that settlement, `end_session` recomputes margin against the updated collateral and position state. If the recomputed margin exceeds the initial requirement, the same flow can deallocate free collateral. If it remains below the initial requirement and no missing margin is allocated, the no-allocation branch accepts the session when the position is not in bad debt and the margin/risk checks improve or do not worsen. The resulting improvement comes from accounting treatment of the realized loss rather than from reduced market exposure, so an under-margined account may be able to remain open, and in some configurations withdraw collateral, while retaining substantially the same exposure.

Remediation

Ensure a zero-price-impact close and reopen cannot improve the position's margin.

Patch

Resolved in [f5c42ad](#).

Positive Funding Inflates Force-Withdraw Targets

ID	Severity	Status
OS-AMP-ADV-02	● Medium	✓ Resolved

Description

`process_clearing_house_for_force_withdraw` computes `position_margin_ratio_target` before closing the position size. In the achievable MR branch, it adds `upnl_before` and `ufundings` directly to `market_imr`, treating positive funding as if each dollar contributes one dollar of margin.

```
packages/market-making-vault/sources/vault.move
```

```

1 public(package) fun process_clearing_house_for_force_withdraw<L, C>([...]) {
2     [...]
3     ifixed::add(
4         ifixed::div(ifixed::add(upnl_before, ufundings),
5             ifixed::mul(ifixed::abs(b), mark_price)),
6         market_imr
7     )
8     [...]
9 }
```

The target is computed before funding is settled, but it is enforced after. When `position_margin_ratio_target` is derived, the funding is still unsettled, so both the target and `compute_margin_with_fundings` value it at full USD and the two agree. The subsequent `deallocate_collateral` call settles that funding into `position.collateral`, where `effective_collateral_usd_value` discounts it by `1 - collateral_haircut`. The post-withdraw check then compares that discounted margin against a target that was frozen before the discount applied, so the target overstates the position's achievable margin ratio by `ufundings * collateral_haircut / notional`.

Exploitation will require a position with positive unsettled funding, a nonzero collateral haircut, and a matured withdrawal request. Under these conditions, an unprivileged withdrawing LP can cause localized disruption and trading losses by forcing the vault to close materially more exposure than the actual margin-ratio preservation rule requires.

Remaining LPs and the vault owner may absorb unnecessary taker fees, spread, slippage, and strategy disruption from excessive market-order position closures. In markets where the inflated target cannot be reached due to lot-size or liquidity constraints, force withdrawals may also fail even though preserving the actual pre-withdraw margin ratio would have been possible.

Proof of Concept

1. A vault position has `notional = $1,000,000`, market IMR of `10%`, collateral haircut of `20%`, and enough collateral for a real margin ratio of `10%`.
2. Positive funding of `$200,000` accrues to the position and remains unsettled when a user starts a force withdraw.

3. The real margin after funding is $(\text{collateral} + \text{funding}) * 80\%$, so the real margin ratio is 26%.
4. The target calculation uses $\text{funding} / \text{notional} + \text{market_imr}$, producing a 30% target.
5. If a user withdraws 1% of LP supply, deallocating the pro-rata value leaves the position near the real 26% ratio, but the force-withdraw post-check rejects it because it is below the inflated 30% target.
6. The user can pass a much larger `size_to_close` so the remaining notional falls enough to satisfy the inflated target, forcing the vault to close materially more exposure than the withdrawal requires.

Remediation

Compute the achievable target with the haircut that applies once the funding is settled into collateral.

Patch

Resolved in [9e38e4c](#) and [4704533](#).

Aggregate Force-Withdraw Dust Can Revert Redemptions

ID	Severity	Status
OS-AMP-ADV-03	● Medium	✓ Resolved

Description

For each processed clearing house, the force-withdraw path derives the withdrawing user's market share from the LP ratio and the market's current position value:

```
packages/market-making-vault/sources/vault.move
1 public(package) fun process_clearing_house_for_force_withdraw<L, C>([...]) {
2     [...]
3     // Calculate the expected amount of collateral to deallocate from this position in
4     // order to
5     // satisfy the force withdrawal. Slippage and fees will be applied later if
6     // necessary.
7     let withdraw_ratio = ifixed::div(
8         ifixed::from_balance(withdraw_session.user_lp_coin.lp_balance.value(),
9             B9_SCALING),
10        ifixed::from_balance(withdraw_session.vault.lp_supply_value(), B9_SCALING)
11    );
12    // Use "position_value" to calculate the amount to withdraw. Haircut value here is
13    // not relevant.
14    let mut amount_to_withdraw = ifixed::mul(position_value, withdraw_ratio);
15    [...]
16 }
```

After any forced close/slippage adjustment, that fixed-point value is converted to raw collateral units. `ifixed::to_balance` floors toward zero, so any fractional raw unit is not deallocated from the market.

```
packages/market-making-vault/sources/vault.move
1 public(package) fun process_clearing_house_for_force_withdraw<L, C>([...]) {
2     [...]
3     let amount_to_withdraw_balance = ifixed::to_balance(
4         ifixed::div(amount_to_withdraw, withdraw_session.collateral_price),
5         withdraw_session.vault.vault_params.scaling_factor
6     );
7     let amount_to_deallocate_balance = min(amount_to_withdraw_balance,
8         free_collateral_balance);
9     let amount_to_withdraw_dust = if (amount_to_deallocate_balance ==
10         amount_to_withdraw_balance) {
11         ifixed::sub(
12             amount_to_withdraw,
13             ifixed::mul(
14                 amount_to_deallocate_balance,
15                 withdraw_session.collateral_price
16             )
17         )
18     }
19 }
```

```

12         ifixed::from_balance(amount_to_deallocate_balance,
13                               withdraw_session.vault.vault_params.scaling_factor),
14                               withdraw_session.collateral_price
15     )
16 } else 0;
17 [...]
18 }

```

Only the floored integer amount is returned to the account. For markets that remain open, the vault subtracts the full per-market dust from the market's remaining value and adds it to the session's aggregate vault value. The issue is the ordering of this correction: finalization does not pay out the actual integer amounts returned to the account, but instead recomputes the user's share from aggregate vault value and converts that value to raw collateral only once.

Thus, an LP may be unable to complete a force withdrawal even after the configured delay has passed and all markets have been processed. Because the transaction reverts atomically at final payout, the user remains dependent on owner or operator intervention, despite the force-withdraw path being intended as a fallback when the owner does not process the request.

Proof of Concept

1. Create three USDC markets and zero their fees.
2. Create a vault with total LP/value of `3_000.000000003` USDC.
3. Alice owns `1_000.000000001` LP, exactly one third of supply.
4. Allocate `1_000.000000001` USDC into each market.
5. Open a small non-dust vault long in each market and provide counterparty bid liquidity so the force-withdraw partial close can pass margin checks while leaving the market non-empty.
6. Alice force-processes all three markets and closes `0.003333334` BTC in each.

Each market's one-third share of `1_000.000000001` USDC is `333.33333333666...` USDC, but raw USDC deallocation floors to `333.333333333` USDC. Across three markets, only `999.999999999` USDC is actually returned to the account. Finalization recomputes Alice's aggregate payout as `1_000.000000000` USDC and calls `withdraw_collateral` for that amount. The account is short by one raw USDC atom, so the transaction aborts in `sui::balance::split` with `ENotEnough(2)`.

Remediation

Do not finalize force withdrawal from a fresh aggregate pro-rata amount unless it is capped by the integer collateral actually available for this withdrawal.

Patch

Resolved in [9e38e4c](#).

Unsafe Position Downcasts Block Critical Operations

ID	Severity	Status
OS-AMP-ADV-04	● Medium	✓ Resolved

Description

Positions are stored as fixed-point `u256` values and are not inherently bounded by the `u64` range accepted by `ifixed::to_balance`. In `process_fill_maker` and `assert_reduce_only`, wrapping the conversion in `min` is ineffective because `to_balance(abs(position))` is evaluated before `min`, causing an abort for oversized positions. The same downcast in `execute_adl` can prevent an oversized bankrupt position from being closed, leaving its bad debt unresolved. Consequently, sufficiently large positions can trigger denial of service in reduce-only matching and ADL, the protocol's critical insolvency-resolution mechanism.

Remediation

Compare and clip position sizes in the `u256` fixed-point domain, use `u128` for ADL deltas, and downcast only after proving the value fits.

Patch

Resolved in [196794e](#).

Favorable Fill Haircut Lets Bad-Debt Makers Fill

ID	Severity	Status
OS-AMP-ADV-05	● Low	✓ Resolved

Description

`can_fill_maker_order_without_bad_debt` is intended to reject maker fills that would leave the maker insolvent, but it does so with an approximation rather than the same accounting path used by the fill. The precheck starts from current haircut-adjusted collateral and unrealized PnL, then applies only the aggregate mark-value difference between the signed base and quote deltas before subtracting fees.

```
packages/perpetuals/sources/position.move
```

```

1 public(package) fun can_fill_maker_order_without_bad_debt([...]): bool {
2     [...]
3     // Apply only the aggregate position-value delta of the hypothetical fill.
4     // This avoids simulating the full close/reverse position update.
5     let (signed_base_delta, signed_quote_delta) = if (order_is_ask) {
6         (
7             ifixed::neg(base_asset_delta),
8             ifixed::neg(quote_asset_delta)
9         )
10    } else (base_asset_delta, quote_asset_delta);
11    let position_value_diff = ifixed::sub(
12        ifixed::mul(signed_base_delta, mark_price),
13        signed_quote_delta
14    );
15    let margin_after = ifixed::sub(
16        ifixed::add(margin_before, position_value_diff),
17        fees
18    );
19    [...]
20 }
```

When a collateral haircut is configured, the same precheck tries to compensate for value that will be moved into collateral by discounting an upper bound of positive realized mark PnL and any maker rebate. In the haircut branch of `can_fill_maker_order_without_bad_debt`, that bound is derived from pre-fill positive unrealized PnL at the mark price and the reducing fraction of the existing position; it does not account for additional realized PnL caused by the maker receiving a more favorable execution price than the mark.

The actual position update realizes PnL from the execution quote, not only from the mark-price value difference. If the maker reduces a long, `add_base_to_position` realizes PnL as `quote_asset_delta - quote_closed`, and if the maker reduces a short, it realizes PnL as

`quote_closed_mag - quote_asset_delta.process_fill_maker` then moves that realized PnL, net of fees, into collateral with `add_to_collateral_usd`.

That distinction matters because positive collateral is discounted by `effective_collateral_usd_value`. Once favorable execution PnL is moved into collateral, the margin calculation values it after applying the haircut, but the precheck can accept the fill using a margin estimate that did not subtract the haircut on that favorable execution gain.

packages/perpetuals/sources/position.move

```

676 fun effective_collateral_usd_value(
677     position: &Position,
678     collateral_price: u256,
679     collateral_haircut: u256
680 ): u256 {
681     let collateral_usd = ifixed::mul(position.collateral, collateral_price);
682     // Haircuts discount positive collateral only. Applying the factor to
683     // negative collateral would make debt look smaller.
684     if (
685         collateral_haircut != 0 &&
686         !ifixed::is_neg(position.collateral)
687     ) {
688         ifixed::mul(
689             collateral_usd,
690             ifixed::sub(constants::one_fixed!(), collateral_haircut)
691         )
692     } else collateral_usd
693 }
```

Positive collateral is haircut-discounted

The maker precheck is performed before the maker position is mutated in `process_fill_maker`. If the precheck returns true, the later fill path updates the maker position with `add_base_to_position` and moves realized PnL net of fees and integrator fees into collateral through `add_to_collateral_usd`, with no exact post-mutation maker solvency check. A reducing maker order can therefore be accepted even though the exact post-fill margin after collateral conversion and haircutting is negative.

The resulting impact is limited to cases where a reducing maker fill realizes favorable execution PnL while a collateral haircut is active and the maker is close enough to insolvency for the missing haircut amount to change the decision. In that case, liquidity that the bad-debt precheck is meant to cancel may instead execute against the taker, leaving the maker with negative margin that must be handled by later liquidation or deficit mechanisms.

Remediation

Make the maker solvency check compute the exact post-fill margin using the same position update, rounding, fee, integrator-fee, collateral conversion, and haircut logic as the actual fill path before deciding whether to cancel the maker order.

Patch

Resolved in [f5c42ad](#).

Over-Haircutted Maker Guard Cancels Solvent Depth

ID	Severity	Status
OS-AMP-ADV-06	● Low	✓ Resolved

Description

Maker fills are screened in `process_fill_maker` before the position is actually updated. After reduce-only, expiration, and self-trade checks determine `allowed_maker_size`, the function pre-views the maker slice that would trade and calls `can_fill_maker_order_without_bad_debt` when that helper returns false, `allowed_maker_size` is set to zero.

packages/perpetuals/sources/clearing_house.move

```

1  fun process_fill_maker<T>([...]): (u64, bool, events::FilledMakerOrder, u256) {
2      [...]
3      if (allowed_maker_size != 0) {
4          // The maker order may be larger than the taker's remaining size, and reduce-
           only
5          // may have clipped it, so check the size that would actually trade.
6          let maker_size_to_match = min(allowed_maker_size, taker_size_to_match);
7          let (base_asset_delta, quote_asset_delta) = fill_base_and_quote_deltas(
8              order_id::price(maker_order_id),
9              maker_size_to_match,
10         );
11         let mut fees = ifixed::mul(maker_fee, quote_asset_delta);
12         if (option::is_some(&maker_integrator_info)) {
13             let info = option::borrow(&maker_integrator_info);
14             fees = ifixed::add(
15                 fees,
16                 ifixed::mul(info.integrator_fee(), quote_asset_delta)
17             );
18         };
19
20         if (!position.can_fill_maker_order_without_bad_debt(
21             order_is_ask,
22             base_asset_delta,
23             quote_asset_delta,
24             fees,
25             collateral_price,
26             mark_price,
27             collateral_haircut
28         )) {
29             allowed_maker_size = 0;
30         };
31     };
32     [...]
33 }
```

The zero allowed size then flows into the cancellation path in the same function. When the taker size is at least the now-zero allowed maker size, the function treats the unfilled resting amount as cancelled, subtracts it from the maker's pending amount, and later decrements the pending-order count, so a failed solvency preview can remove the live order without filling it.

```
packages/perpetuals/sources/clearing_house.move
```

```

1  fun process_fill_maker<T>([...]): (u64, bool, events::FilledMakerOrder, u256) {
2      [...]
3      } else {
4          [...]
5          // Proceed to check if all the size required by the taker is exhausted too
6          if (taker_size_to_match >= allowed_maker_size) {
7              // Taker's required size is enough to fully close maker's position.
8              // Proceed to calculate the amount of the order that should be cancelled.
9              order_size_cancelled = order_size - allowed_maker_size;
10             position.sub_from_pending_amount(
11                 order_is_ask,
12                 ifixed::from_balance(order_size_cancelled, constants::b9_scaling!())
13             );
14
15             allowed_maker_size
16         } else {
17             // Taker's required size is NOT enough to fully close maker's position.
18             // The order should not be dropped yet and there is no size to cancel too.
19             order.reduce_order_size(taker_size_to_match);
20             include_last_order_id = false;
21
22             taker_size_to_match
23         }
24     };
25     // Update number of pending orders if order has been completely filled or cancelled
26     if (include_last_order_id) {
27         position.update_pending_orders(false, 1);
28     };
29     [...]
30 }
```

`can_fill_maker_order_without_bad_debt` computes an approximate post-fill margin from current effective collateral, current unrealized PnL, the position-value delta of the hypothetical fill, and fees. When `collateral_haircut` is nonzero, the helper then subtracts a haircut from an upper bound of positive unrealized PnL that could be realized by a reducing fill, and also includes maker rebates in that maximum collateral gain.

The actual maker settlement path does not apply that approximation. For a matched maker fill, `process_fill_maker` calls `add_base_to_position` to realize PnL, computes the net collateral change from realized PnL minus fees and integrator fees, and applies that value through `add_to_collateral_usd`.

The margin functions value collateral through `effective_collateral_usd_value`, which only discounts non-negative collateral and leaves negative collateral at full value. As a result, realized gain that first offsets a negative collateral balance should not be haircut-discounted for the portion that merely repays debt, because exact margin accounting does not discount negative collateral.

packages/perpetuals/sources/position.move

```

676 fun effective_collateral_usd_value(
677     position: &Position,
678     collateral_price: u256,
679     collateral_haircut: u256
680 ): u256 {
681     let collateral_usd = ifixed::mul(position.collateral, collateral_price);
682     // Haircuts discount positive collateral only. Applying the factor to
683     // negative collateral would make debt look smaller.
684     if (
685         collateral_haircut != 0 &&
686         !ifixed::is_neg(position.collateral)
687     ) {
688         ifixed::mul(
689             collateral_usd,
690             ifixed::sub(constants::one_fixed!(), collateral_haircut)
691         )
692     } else collateral_usd
693 }

```

Collateral haircut applies only to non-negative collateral

For a maker with negative collateral backed by positive unrealized PnL, the helper can therefore produce a lower post-fill margin than exact accounting would produce. The preview subtracts `collateral_haircut` from the full positive realized-PnL upper bound even when part or all of that gain would first bring collateral toward zero, so a solvent reducing order near the solvency boundary may be classified as bad-debt-prevented and cancelled. This removes executable risk-reducing depth from the orderbook and may deny the maker an available close or reduction path.

Remediation

Use a non-mutating maker-fill preview that mirrors the real settlement path: compute the same realized PnL that `add_base_to_position` would produce for the proposed maker slice, apply the same fee and integrator-fee deductions to a preview collateral balance, and evaluate post-fill margin with `effective_collateral_usd_value`.

Patch

Resolved in [f5c42ad](#).

Floored Mark Triggers Full Liquidation

ID	Severity	Status
OS-AMP-ADV-07	● Low	✓ Resolved

Description

`settle_liquidated_position` sizes a liquidation by passing the session's cached `mark_price` directly into `compute_liquidation_size_and_mode`. The liquidation sizing helper uses this full fixed-point mark in its margin checks, numerator and denominator calculations, and rounded-up alpha computation.

```
packages/perpetuals/sources/clearing_house.move
1 fun settle_liquidated_position<T>([...]) {
2   [...]
3   // Calculate size to liquidate necessary to bring the position's margin ratio
4   // back above market's IMR
5   let (size_to_liquidate, force_cancel_only) = compute_liquidation_size_and_mode(
6     position,
7     hot_potato.collateral_price,
8     mark_price,
9     margin_ratio_initial,
10    liquidation_fee,
11    insurance_fund_fee,
12    collateral_haircut
13  );
14  [...]
15 }
```

The mutation path does not settle the reduction at that same precision. In `reduce_liquidated_position`, the same `mark_price` is converted to a B9 balance before `fill_base_and_quote_deltas` computes the quote delta as `size * price`. Because that conversion floors the fixed-point mark to a `u64` price, a long liquidation can realize slightly less quote value and PnL than the sizing formula assumed when the mark contains precision beyond B9.

`clip_size_to_liquidate` adds one B9 base atom before applying lot-size rounding. That adjustment cushions base-size rounding edge cases, but it does not make the execution quote value consistent with the higher-precision mark used by the sizing calculation. The value lost to price flooring is proportional to the liquidated size rather than only to one base atom.

```
packages/perpetuals/sources/clearing_house.move
4088 public fun clip_size_to_liquidate(
4089   size_to_liquidate: u256,
4090   position_base_amount: u256,
4091   lot_size: u64
```

```

4092 ): u64 {
4093     // increment even if division is perfect because of edge cases where 0 < alpha <
4094     // 1E-18
4095     let down_size_to_liquidate = ifixed::to_balance(ifixed::abs(size_to_liquidate),
4096     constants::b9_scaling!()) + 1;
4097     let down_position_base_amount =
4098     ifixed::to_balance(ifixed::abs(position_base_amount), constants::b9_scaling!
4099     ());
4100
4101     // Clip size_to_liquidate to the maximum position's base amount.
4102     // `to_balance` internally calls `abs(base_asset_amount)`
4103     let down_size_to_liquidate = if (lot_size != 1) {
4104         min(
4105             div_ceil(down_size_to_liquidate, lot_size) * lot_size,
4106             down_position_base_amount
4107         )
4108     } else {
4109         // If `lot_size == 1`, the function `div_ceil` is not required because there
4110         // is no possible remainder
4111         min(
4112             down_size_to_liquidate,
4113             down_position_base_amount
4114         )
4115     };
4116
4117     down_size_to_liquidate
4118 }

```

After the partial reduction, `settle_liquidated_position` immediately compares `margin_after` against `margin_req_after` and enters a full-liquidation fallback if the account is still unhealthy. Since `reduce_liquidated_position` recomputes post-reduction health with the original mark while the realized PnL, liquidation fee, and insurance fee are based on the floored execution quote, a precision-only mismatch can leave `margin_after` below `margin_req_after` by dust and cause the remaining base position to be liquidated.

The impact is limited to cases where the partial liquidation should have restored a long account to the initial margin requirement but the floored execution price leaves it just short. In those cases, the liqee can lose the remaining position and pay liquidation and insurance fees on additional notional. If the unnecessary extra liquidation creates or increases bad debt, the insurance fund or downstream debt-handling path may also be affected.

Remediation

Use one liquidation closeout price consistently across sizing, execution, fee calculation, and post-reduction health checks.

Patch

Resolved in [f5c42ad](#).

Force Withdraw Uses Divergent Collateral Prices

ID	Severity	Status
OS-AMP-ADV-08	● Low	✓ Resolved

Description

`start_force_withdraw_session` stores a `collateral_price` in the `WithdrawSession`, and the force path uses that snapshot for market valuation and for converting the withdrawer's pro-rata market value into collateral atoms. However, `process_clearing_house_for_force_withdraw` then calls the perps deallocation path with the oracle object, and that path reads the current collateral oracle again before settling funding into position collateral

```
packages/market-making-vault/sources/vault.move
```

```

1 public(package) fun process_clearing_house_for_force_withdraw<L, C>([...]) {
2     [...]
3     let amount_to_withdraw_balance = ifixed::to_balance(
4         ifixed::div(amount_to_withdraw, withdraw_session.collateral_price),
5         withdraw_session.vault.vault_params.scaling_factor
6     );
7     perps_interface::deallocate_collateral(
8         &mut clearing_house,
9         withdraw_session.vault.account_cap(),
10        &mut withdraw_session.account,
11        base_oracle,
12        collateral_oracle,
13        oracle_config,
14        clock,
15        amount_to_withdraw_balance
16    );
17    [...]
18 }
```

This is inconsistent when the vault position has unsettled funding already reflected in the clearing house's stored cumulative funding rates. The vault's read-only margin helper includes that funding as a USD term using the session price, while perps `take_collateral` later settles the same funding into collateral units using the newer oracle price. After settlement, the vault computes `position_value_after` using the old session price and adds the residual market value to `vault_balance_value`. Finalization then values idle account collateral and computes `coin_out_amount` using the same old price.

Because oracle integration update functions are public once the source value is available, a force withdrawer can build a PTB that starts the force-withdraw session using the old collateral price, updates only the collateral `PriceFeedStorage`, and then processes the market. Moving the vault and account into the `WithdrawSession` does not prevent this oracle-only interleaving.

If the vault is owed funding and the new collateral price is lower, funding settlement mints more collateral units into the position, which the vault then values at the older, higher price, inflating the residual market value. The final payout includes the withdrawer's pro-rata share of this artificial value and may be paid from unrelated idle collateral in the vault account. If idle collateral is insufficient, the same mismatch causes `withdraw_collateral` to abort after all markets have been processed.

Remediation

Use one collateral price for the entire force-withdraw accounting path.

Patch

Resolved in [9e38e4c](#).

Deposits Receive Free Bad-Debt Recovery Option

ID	Severity	Status
OS-AMP-ADV-09	● Low	✓ Resolved

Description

Deposit valuation clips each negative-margin market to zero, while the vault retains the open position and its future recovery potential. New deposits made while the market is clipped receive pro-rata ownership of any later recovery from that existing position at no cost. Existing LPs can be diluted out of the recovery value of underwater positions. The loss is bounded by the eventual recovery from the negative-margin market, but may be material when a large position temporarily falls below zero margin while the vault still holds meaningful idle collateral or other positive assets used for deposit pricing.

Proof of Concept

1. A vault has 2,000 LP shares, 1,801 USDC idle collateral, and a 1 BTC long position with 199 USDC collateral opened at 1,200.
2. BTC falls to 1,000, making the position margin $199 - 200 = -1$ USDC.
3. A depositor starts a deposit session and processes the active clearing house.
4. `get_vault_margin_in_market` returns `0` for the negative-margin position, so `vault_balance_value` is only 1,801 USDC.
5. The depositor adds 1,801 USDC and receives 2,000 LP shares, becoming a 50% LP holder.
6. BTC later returns to 1,200 before the position is liquidated or closed. The position margin recovers to 199 USDC.
7. The depositor can withdraw roughly 50% of the recovered 199 USDC, extracting value from the pre-existing LPs' position.

Remediation

Do not allow new deposits to be priced while any active market has negative margin unless the underwater position is first resolved.

Patch

Resolved in [9e38e4c](#) and [b20dfe9](#).

Settled Markets Remain In Vault Membership

ID	Severity	Status
OS-AMP-ADV-10	● Low	✓ Resolved

Description

`Vault.ch_ids` is the source of truth for markets that must be processed before deposits and withdrawals can finish. Each session copies that vector at creation, and finalization requires the copied list to be empty. Processing a listed market removes its ID from the session list and then calls the normal active-market funding path

Active-market processing cannot handle a closed market because `perpetuals::clearing_house::update_funding` rejects paused markets. Although the perpetuals package provides a closed-market settlement path, including an admin path that can settle any account and deallocate settled collateral back to that account, that call bypasses the market-making-vault wrappers that maintain `Vault.ch_ids`. The vault also exposes no function for closed-market settlement or membership reconciliation.

After the vault account's closed-market position is settled and emptied, the stale clearing-house ID remains in `Vault.ch_ids`. Subsequent deposits, owner withdrawals, and user force withdrawals must snapshot and process that stale ID, but processing reverts on the paused-market check before the session can finish. As a result, users cannot complete withdrawals from the affected vault, and new deposits cannot finalize. Although funds are not directly stolen, stale membership state freezes user exits and vault operations.

Remediation

Give the vault a way to reconcile `Vault.ch_ids` for a market that can no longer be processed.

Patch

Resolved in [9e38e4c](#) and [4704533](#).

Closed Market Withdrawals Use Oracles

ID	Severity	Status
OS-AMP-ADV-11	● Low	✓ Resolved

Description

`withdraw_insurance_fund` computes the withdrawable insurance surplus by valuing the insurance fund with the current collateral oracle price and valuing open interest with the current base oracle price. The function does not check whether the market is closed before doing this calculation, so the same oracle-based path is used even after the clearing house has been closed and settlement prices have been configured.

Closed markets are represented by a `SettlementPrices` dynamic field that stores both `base_price` and `collateral_price`. The market-close and settlement-closeout paths use those stored settlement prices when positions are settled, including when converting funding, PnL, collateral deallocation, and bad debt. This results in inconsistent accounting for closed markets as user settlement is performed at the configured settlement prices, while insurance-fund surplus withdrawal can still be bounded by live oracle prices.

As a result, after a market is closed, the treasury may be able to withdraw an amount of insurance collateral based on oracle prices that no longer match the closed market's settlement basis. If the oracle collateral price differs from the stored collateral settlement price, the computed insurance fund value and surplus can diverge from the value used by the settlement flow, potentially allowing excess withdrawal or incorrectly blocking valid surplus withdrawal.

Remediation

Update `withdraw_insurance_fund` to use the stored `SettlementPrices` values when the clearing house is closed. In the closed-market branch, read `settlement_prices().collateral_price` for insurance-fund collateral valuation and `settlement_prices().base_price` for open-interest valuation instead of querying the collateral and base oracles.

Patch

Resolved in [e141f9d](#).

Market Closure Uses Stale Funding

ID	Severity	Status
OS-AMP-ADV-12	● Low	✓ Resolved

Description

`close_market` closes the market by checking package and registry versions, validating the vendor authority, asserting the market is not already closed, validating the supplied settlement prices, pausing the market, and storing `SettlementPrices`. As shown in `close_market`, this path does not refresh the clearing house market state before settlement is enabled for later closeout processing.

packages/perpetuals/sources/clearing_house.move

```

984 public fun close_market<VendorKey, Role, T>([...]) {
985     clearing_house.assert_package_version();
986     registry.assert_package_version();
987     registry.assert_admin_or_authorized_assistant_authority_cap(cap);
988     registry.assert_vendor_has_ownership_over_clearing_house(cap,
989         clearing_house.id.as_inner());
989     clearing_house.assert_market_is_not_closed();
990
991     assert_settlement_prices(base_settlement_price, collateral_settlement_price);
992
993     clearing_house.paused = true;
994
995     df::add(
996         &mut clearing_house.id,
997         keys::settlement_prices(),
998         SettlementPrices {
999             base_price: base_settlement_price,
1000             collateral_price: collateral_settlement_price,
1001             deallocate_collateral,
1002             enabled: false,
1003         },
1004     );
1005
1006     events::emit_closed_market(
1007         clearing_house.id.to_inner(),
1008         base_settlement_price,
1009         collateral_settlement_price,
1010         deallocate_collateral,
1011         false,
1012     );
1013 }
```

The later settlement closeout path reads the stored settlement prices and then settles a position using the clearing house market state's cumulative funding rates. Because `close_market` does not call `try_update_funding`, `try_update_twaps`, or `try_update_fundings_and_twaps`, positions closed after market closure may settle funding against the last previously recorded market-state values rather than values updated at closure time. The same stale-state pattern exists in `execute_adl`, where ADL reads cumulative funding rates and uses TWAP-dependent mark-price logic without first invoking the combined funding/TWAP update helper used by ordinary session and collateral deallocation flows.

Remediation

Refresh market funding and TWAP state immediately before any closeout or ADL accounting reads those values. For `execute_adl`, update the clearing house market state before reading cumulative funding rates or calculating the mark price so bad-debt checks, funding settlement, and counterparty reductions use current state.

Patch

Resolved in [e141f9d](#).

Settlement Ordering Blocks Bad-Debt Socialization

ID	Severity	Status
OS-AMP-ADV-13	● Low	✓ Resolved

Description

`close_position_at_settlement_prices` allows account administrators or assistants to close an account's position after a market has been paused and closed. The shared settlement macro settles funding, realizes the full position at the configured settlement price, resets negative collateral into `bad_debt`, and updates market open interest before invoking bad-debt handling. Because open interest is updated as each position closes, solvent long positions can reduce the market's recorded open interest before later short positions are processed.

Bad-debt socialization depends on the market's current open interest when `handle_bad_debt` calls `try_socialize_bad_debt`. That routine aborts if open interest is zero and computes socialized funding using the remaining open interest. If longs close first, settlement can significantly reduce or eliminate the denominator used for socialization. When shorts later generate bad debt, socialization may become impossible or fail the configured threshold, even though the low open-interest state was caused by settlement ordering.

Remediation

Decouple settlement bad-debt socialization from the live open-interest value that is reduced by earlier settlement closes.

Patch

Resolved in [e141f9d](#).

Standalone Stop Orders Fail at Exact Trigger Price

ID	Severity	Status
OS-AMP-ADV-14	● Low	✓ Resolved

Description

Standalone stop orders let the ticket creator choose whether the trigger condition should be evaluated above or below `stop_index_price` through `ge_stop_index_price`. In `place_stop_order_standalone`, the `true` branch accepts equality at the configured stop price, while the `false` branch requires the derived trigger price to be strictly below the configured stop price. This makes the two directions asymmetric and prevents a standalone stop order configured for the lower-price direction from executing when the trigger price is exactly equal to the selected `stop_index_price`.

As a result, an otherwise valid standalone stop order with `ge_stop_index_price` set to `false` can fail at the exact stop threshold and only become executable after the trigger price moves below that threshold.

Remediation

Change the `ge_stop_index_price == false` branch in `place_stop_order_standalone` to accept trigger prices less than or equal to `stop_index_price`.

Patch

Resolved in [e141f9d](#).

Post-Resume Funding Charges Pause Interval

ID	Severity	Status
OS-AMP-ADV-15	● Low	✓ Resolved

Description

Market pause and resume operations in `clearing_house` only toggle the clearing house `paused` flag. While the market is paused, `update_funding` and other active-market flows are blocked by the paused-market check, but resuming the market does not adjust the market state's funding timestamp.

After the market resumes, the next funding update can include the entire elapsed pause interval. The funding updates are allowed once the next funding boundary has passed, and the funding period adjustment is computed from the number of elapsed `funding_frequency_ms` intervals between `funding_last_upd_ms` and the current timestamp. The update then applies the premium TWAP scaled by that adjustment to the cumulative long and short funding rates before storing the current timestamp as the new funding update time.

As a result, a long pause can cause the first post-resume funding update to settle funding for time when the market was inactive, rather than for a bounded post-resume interval. Those cumulative funding rates are later used when positions settle funding, so users may pay or receive funding based on the paused duration.

Remediation

Cap the elapsed funding interval used by `funding_period_adjustment` to at most three funding-frequency intervals.

Patch

Resolved in [e141f9d](#).

Cross-Account Assistant Cap Revocation

ID	Severity	Status
OS-AMP-ADV-16	● Low	✓ Resolved

Description

`revoke_assistant_account_cap` accepts an `assistant_cap_id` supplied by the account admin. The local account logic validates that the admin cap is valid for the supplied `Account<T>`, and assistant caps are normally linked to that account because they are created from its corresponding account admin cap. However, an admin can pass another account's assistant-cap ID into this revocation flow, removing that cap's global registry authorization marker without revoking it from its actual account.

This creates an inconsistent authorization state for the victim assistant cap. The assistant cap object may remain active on its actual account while its global registry marker has been removed. Account-level operations that rely only on the account's `active_assistants` list may still treat the assistant as active, while code paths that consult the registry's authorized multiton-cap marker may reject the same cap as deauthorized. This can unexpectedly disable registry-gated assistant-cap behavior for another account and desynchronize the protocol's two sources of assistant-cap authority.

Remediation

Bind account assistant-cap revocation to the account that owns the assistant cap before removing any registry marker.

Patch

Resolved in [e141f9d](#).

Self-Cancellation Enables Undercollateralized IOC Positions

ID	Severity	Status
OS-AMP-ADV-17	● Low	✓ Resolved

Description

When a taker crosses its own resting order, `process_fill_maker` cancels that order and releases its reserved pending exposure, but allows matching to continue because `taker_size_matched` is zero. An ensuing IOC fill can create live base exposure without posting a replacement order, while `end_session_` compares final health against the stale, pre-cancellation `min_margin_before`.

The `taker_pending_cancelled` guard triggers only when `posted_orders != 0`, and the below-MMR resting-order check likewise does not cover IOC fills. Consequently, `ensure_margin_requirements` may misclassify the session as risk-improving and commit a below-MMR position that can realize bad debt upon liquidation.

Remediation

Require any session with self-cancelled pending exposure and subsequent taker fills to restore full IMR, automatically allocating missing collateral or aborting.

Patch

Resolved in [306d1e4](#).

Pending Orders Let Under-IMR Liquidators Acquire Live Positions

ID	Severity	Status
OS-AMP-ADV-18	● Low	✓ Resolved

Description

`start_session` records the liquidator's live base amount and computes the pre-session margin requirement before liquidation settlement occurs. The stored `min_margin_before` is computed from the current `Position` state, while `position_base_before` captures only the live base amount.

The pre-session requirement includes pending exposure because `margin_requirement` delegates to `abs_net_base`, which takes the maximum absolute exposure after adding pending bids or subtracting pending asks from the live base. Thus, a pending order can therefore contribute to the scalar requirement even when there is no corresponding live position yet. `liquidate` force-cancels the liqee's supplied pending orders before executing liquidation, but the function does not cancel or otherwise isolate the liquidator's own pending orders. The cancellation list is applied to `liqee_account_id`, leaving any pre-existing pending orders on the liquidator position in the risk baseline used by the session.

During session processing, the liquidation leg is then applied to the liquidator's live position before the final margin check. In `process_session_hot_potato`, the liquidator receives the opposite side of the liqee's liquidated base amount through `add_base_to_position`, and liquidation PnL plus fees are settled into collateral. If the resulting position remains below initial margin and `allocate_missing_margin` is false, the end-of-session path does not require new collateral immediately. In the final margin branch, it calls `position::ensure_margin_requirements` with the old `min_margin_before` and the live base amounts before and after the session.

`ensure_margin_requirements` permits an under-IMR session when the new scalar requirement is not greater than the old scalar requirement and the live base direction did not flip, with zero live base before the session treated as acceptable. The comparison does not distinguish pending-only exposure from live exposure, so a pending order that inflated `min_margin_before` can make the liquidation acquisition appear non-increasing.

This creates a path where an under-IMR liquidator can have an old opposite-side pending order, acquire live liquidation exposure, and pass the below-IMR improvement check because the scalar `abs_net_base` requirement remains unchanged. The same pending exposure can serve as the pre-session risk baseline and then remain cancellable after the liquidation has converted the account into a live position.

After the liquidation session, the public `cancel_orders` endpoint lets the account cancel its own pending orders through the normal order-cancellation path. Within `cancel_orders`, the endpoint validates account authority and market state, then delegates to the cancellation macro without performing a margin check in that wrapper. The cancellation macro removes the canceled ask and bid quantities from the position's pending amounts and decrements the pending order count. This updates pending exposure but does not recompute and enforce the initial margin requirement after pending exposure is removed.

The resulting account may retain the acquired live liquidation position while still below IMR and without having allocated the missing collateral that a normal opening trade would require. This weakens the liquidation backstop because the liquidator can collect liquidation fees while posting less than the required initial margin for the live position, and adverse movement before restoration or liquidation could increase losses borne by the insurance fund or bad-debt mechanisms described in the liquidation flow.

Remediation

Do not let pending-only exposure justify acquiring live liquidation exposure below initial margin.

Patch

Resolved in [f5c42ad](#).

Stale Pending Orders Can Be Reposted Into Fresh Under-Margined Exposure

ID	Severity	Status
OS-AMP-ADV-19	● Low	✓ Resolved

Description

Pending orders contribute to a position's initial margin requirement because `margin_requirement` bases the requirement on `abs_net_base`. This requirement is used together with collateral and mark price to compute the position's margin state.

```
packages/perpetuals/sources/position.move
```

```

754 public fun margin_requirement(
755     position: &Position,
756     mark_price: u256,
757     margin_ratio: u256,
758 ): u256 {
759     let abs_net_base = abs_net_base(position);
760     let abs_net_base_value = ifixed::mul(abs_net_base, mark_price);
761
762     ifixed::mul(abs_net_base_value, margin_ratio)
763 }
```

`abs_net_base` includes both live base exposure and pending bid or ask quantities when determining the larger absolute exposure. This means canceling a pending order can reduce the computed margin requirement even if no trade occurs.

```
packages/perpetuals/sources/position.move
```

```

766 public fun abs_net_base(position: &Position): u256 {
767     let base_asset_amount = position.base_asset_amount;
768     let bids_net_abs = ifixed::abs(
769         ifixed::add(base_asset_amount, position.bids_quantity)
770     );
771     let asks_net_abs = ifixed::abs(
772         ifixed::sub(base_asset_amount, position.asks_quantity)
773     );
774
775     ifixed::max(bids_net_abs, asks_net_abs)
776 }
```

At session start, `start_session` settles funding, records the taker's base amount, and computes the pre-session margin and minimum margin requirement before order matching or cleanup. During limit-order execution, the matching loop calls `process_fill_maker` before posting any remaining taker size. Expired maker orders and maker orders owned by the taker are made unfillable, and their pending amounts can be removed from the maker position, and if the maker

is also the taker this cleanup affects the same position whose pre-session margin baseline was already saved. If the taker's remaining size is not filled and the order type permits posting, `execute_limit_order` posts that residual size and records the posted amount in the session summary.

At session end, `process_post` applies the posted residual size to the taker's pending-order accounting before the final margin calculation. The final check then compares the post-session margin state against the original pre-cleanup baseline.

`ensure_margin_requirements` permits a below-IMR position to proceed when the current minimum margin requirement does not exceed the saved requirement, the base position did not flip, and margin did not decrease. Because the saved requirement can still include an expired or self-crossed pending order that was canceled earlier in the same session, a user may replace stale non-executable pending exposure with a fresh posted order of similar size while remaining below initial margin. This could leave live executable maker exposure on an account that would have failed the same post if the stale order had been canceled in a separate step first.

Remediation

Do not let cleanup-only pending reductions serve as margin budget for new posts in the same session. Compute a post-cleanup/pre-post margin baseline and compare newly posted exposure against that baseline.

Patch

Resolved in [f5c42ad](#).

6. Suggestions

Here, we present a discussion of the 20 suggestions we identified during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Severity	Status	Description
OS-AMP-SUG-00	● Informational	✓ Resolved	Default Minimum Order Bypasses Registry
OS-AMP-SUG-01	● Informational	✓ Resolved	Funding Receivables Can Hide ADL Bad Debt
OS-AMP-SUG-02	● Informational	✓ Resolved	Insurance Fee Pre-Transfer Can Block Bad-Debt Liquidations
OS-AMP-SUG-03	● Informational	✓ Resolved	Liquidation Fees Can Exceed Maintenance Margin
OS-AMP-SUG-04	● Informational	✓ Resolved	Resume Has No Cancel-Only Grace
OS-AMP-SUG-05	● Informational	✗ To-do	Vendor Assistants Can Escalate To Perpetuals Vendor Admin
OS-AMP-SUG-06	● Informational	✗ To-do	Socialized Bad Debt Can Be Underpaid After Collateral Price Moves
OS-AMP-SUG-07	● Informational	✗ To-do	Asynchronous Funding Conversion Enables Collateral Arbitrage
OS-AMP-SUG-08	● Informational	✗ To-do	Realized PnL FX Arbitrage
OS-AMP-SUG-09	● Informational	✓ Resolved	Cleanup-Only Standalone Stops Can Allocate Collateral
OS-AMP-SUG-10	● Informational	✓ Resolved	Cleanup-Only Stop Executions Can Burn Tickets Without Any Fill
OS-AMP-SUG-11	● Informational	✓ Resolved	Empty TWAP Attempts Can Allocate User Collateral
OS-AMP-SUG-12	● Informational	✓ Resolved	TWAP Fresh Chunks Bypass Execution Cap
OS-AMP-SUG-13	● Informational	✓ Resolved	Stop Executors Can Add Uncommitted Priority Taker Fees
OS-AMP-SUG-14	● Informational	✓ Resolved	TWAP Executors Can Force Priority Taker Fees
OS-AMP-SUG-15	● Informational	✓ Resolved	Stop Order Executions Bypass Vault Pending Order Cap
OS-AMP-SUG-16	● Informational	○ Ack'd	Frozen Oracles Remain Temporarily Consumable
OS-AMP-SUG-17	● Informational	○ Ack'd	Manipulated Book Marks Underfund New Positions
OS-AMP-SUG-18	● Informational	○ Ack'd	Stale Pre-Fill Marks Permit Maker Insolvency
OS-AMP-SUG-19	● Informational	○ Ack'd	Endpoint Quote Manipulation Distorts Funding Intervals

Default Minimum Order Bypasses Registry

ID	Severity	Status
OS-AMP-SUG-00	● Informational	✓ Resolved

Description

Market creation validates the explicit market creation inputs in `assert_market_parameters` but does not validate the risk-limit defaults that are filled in later. After those checks, `create_market_params` constructs the `MarketParams` object and assigns the per-market `min_order_usd_value` from the fixed one-dollar constant. Because this happens after `assert_market_parameters`, market creation can persist that default even when the registry's configured lower bound for minimum order value is higher.

The update path enforces a stricter invariant for the same field. In `set_risk_limit_params`, the selected or existing `min_order_usd_value` is checked against both `registry_config.up_min_order_usd_value()` and `registry_config.low_min_order_usd_value()` before the risk-limit fields are written, so the creation path and update path can disagree about whether the same value is valid.

The registry configuration permits the lower and upper minimum-order bounds to be changed as long as they remain nonnegative and ordered. Since `set_config` does not require the lower bound to remain at or below the one-dollar creation default, operators can configure a lower bound that the hard-coded creation default would not satisfy.

packages/perpetuals/sources/registry.move

```

1 public fun set_config<ADMIN_OR_ASSISTANT>([...]) {
2     [...]
3 };
4     assert!(
5         ifixed::greater_than_eq(low_min_order_usd_value, 0) &&
6         ifixed::greater_than_eq(up_min_order_usd_value, 0) &&
7         ifixed::less_than_eq(low_min_order_usd_value, up_min_order_usd_value),
8         invalid_min_order_usd_value_bounds!(),
9     );
10    [...]
11 }
```

Registry configuration allows the lower bound to be raised

Runtime limit-order placement then relies on the stored market parameter rather than rechecking the registry. In `place_limit_order`, after matching, any residual posted size is compared against the `min_order_usd_value` read from the clearing house's `market_params`, so a newly created market with the stale one-dollar default can continue using that value for posted-order admission.

The helper `assert_order_value` computes the posted order value from the residual size and the session mark price, then compares that value only to the per-market `min_order_usd_value`

supplied by the caller. It does not consult the registry bounds, which makes the stored market parameter the effective runtime threshold.

packages/perpetuals/sources/clearing_house.move

```

4678 fun assert_order_value(
4679     size_posted: u64,
4680     index_price: u256,
4681     min_order_usd_value: u256
4682 ) {
4683     // Check order value based on index price and abort if less than threshold
4684     let order_value = ifixed::mul(
4685         ifixed::from_balance(size_posted, constants::b9_scaling!()),
4686         index_price
4687     );
4688
4689     assert!(
4690         order_value >= min_order_usd_value,
4691         order_usd_value_too_low!()
4692     );
4693 }

```

Posted order value is compared to the per-market threshold

This lower threshold matters because the book midpoint is derived only from the current best ask and best bid keys. In `book_price`, order size is not part of the midpoint calculation, so orders that satisfy the under-bounded per-market threshold may still define the book-derived price consumed by the wider accounting paths described in the finding, including TWAP and funding-related flows.

packages/perpetuals/sources/orderbook.move

```

132 public fun book_price(book: &Orderbook): Option<u64> {
133     let (no_asks, no_bids) = (
134         map::is_empty(asks(book)), map::is_empty(bids(book))
135     );
136     if (no_bids || no_asks) option::none()
137     else {
138         let min_ask = order_id::price_ask(map::min_key(asks(book)));
139         let max_bid = order_id::price_bid(map::min_key(bids(book)));
140         let price = (min_ask + max_bid) / 2;
141         option::some(price)
142     }
143 }

```

Book price is computed from best bid and ask only

Remediation

Validate the creation-time `min_order_usd_value` default against the same registry lower and upper bounds enforced by `set_risk_limit_params` before storing `MarketParams`.

Patch

Resolved in [f5c42ad](#).

Funding Receivables Can Hide ADL Bad Debt

ID	Severity	Status
OS-AMP-SUG-01	● Informational	✓ Resolved

Description

`execute_adl` is the permissioned automated deleveraging path for a bad-debt position, but its initial bad-debt predicate is evaluated before the function settles the bad-debt account's funding or cancels supplied pending orders. The function computes `margin` with `compute_margin_with_fundings` and returns immediately when that preview margin is positive, as shown in the initial eligibility check.

```
packages/perpetuals/sources/adl.move
1 public fun execute_adl<T>([...]) {
2     [...]
3     let (margin, _min_margin) = bad_debt_position.compute_margin_with_fundings(
4         collateral_price,
5         mark_price,
6         0, // min margin ratio, only affects _min_margin
7         mkt_funding_rate_long,
8         mkt_funding_rate_short,
9         collateral_haircut
10    );
11    // Allowing adl on 0 margin allows force canceling open orders on an otherwise empty
    position
12    if (ifixed::greater_than(margin, 0)) return;
13    [...]
14 }
```

The early-return branch is the specific gate that can stop the rest of the ADL state transition. If the early return is not taken, the same ADL flow proceeds to `force_cancel_orders` and only then borrows the bad-debt position mutably to call `settle_position_funding`. This ordering means a positive pre-settlement preview prevents both the pending-order cleanup and the funding settlement that would otherwise be part of the ADL state transition.

```
packages/perpetuals/sources/adl.move
1 public fun execute_adl<T>([...]) {
2     [...]
3     let bad_debt_position = clearing_house.borrow_mut_position(bad_debt_account_id);
4     bad_debt_position.settle_position_funding(
5         collateral_price,
6         mkt_funding_rate_long,
7         mkt_funding_rate_short,
8         &ch_id,
```

```

9      bad_debt_account_id,
10    );
11    [... ]
12  }

```

The order-cancel portion is also reached only after the preview predicate. The preview helper used by the ADL predicate calculates unrealized funding separately from collateral. It first computes the effective USD value of the existing collateral, then adds unrealized PnL and the funding value as independent signed terms, so a funding receivable contributes to the preview without first being converted into collateral.

`effective_collateral_usd_value` applies `collateral_haircut` only when the position's current collateral is non-negative. After actual funding settlement, a received funding amount increases `position.collateral` in collateral units, so subsequent margin accounting discounts the resulting positive collateral balance through this helper instead of valuing the receivable as an unhaircutted USD term.

packages/perpetuals/sources/position.move

```

676 fun effective_collateral_usd_value(
677     position: &Position,
678     collateral_price: u256,
679     collateral_haircut: u256
680 ): u256 {
681     let collateral_usd = ifixed::mul(position.collateral, collateral_price);
682     // Haircuts discount positive collateral only. Applying the factor to
683     // negative collateral would make debt look smaller.
684     if (
685         collateral_haircut != 0 &&
686         !ifixed::is_neg(position.collateral)
687     ) {
688         ifixed::mul(
689             collateral_usd,
690             ifixed::sub(constants::one_fixed!(), collateral_haircut)
691         )
692     } else collateral_usd
693 }

```

This accounting mismatch can allow a position with funding receivable to appear solvent in the ADL preview even though its settled margin would be negative once that receivable is converted into positive, haircut-discounted collateral. In that case, `execute_adl` may return before closing the bad-debt position, before settling its funding, and before canceling pending orders. The `clearing_house::settle_position_funding` public entrypoint also requires the market to be unpaused, while `execute_adl` checks only that the market is not closed, so this difference is

most relevant when the permissioned ADL route is expected to operate during a paused-market incident.

Remediation

Make the ADL bad-debt predicate use the same accounting as the ADL state transition.

Patch

Resolved in [f5c42ad](#).

Insurance Fee Pre-Transfer Can Block Bad-Debt Liquidations

ID	Severity	Status
OS-AMP-SUG-02	● Informational	✓ Resolved

Description

Liquidation accounting first computes the liqee's realized PnL and deducts both liquidation fees and insurance-fund fees from the liqee collateral. In `reduce_liquidated_position`, if that fee-inclusive collateral update leaves the position negative, the position collateral is reset and the deficit is returned as `bad_debt`.

packages/perpetuals/sources/clearing_house.move

```

1  public(package) fun reduce_liquidated_position([...]): (u256, u256, bool, u256, u256,
2    [...]
3    // Update collateral for liqee
4    let insurance_fund_fees = ifixed::mul(insurance_fund_fee, quote_asset_delta);
5    let liquidation_fees = ifixed::mul(liquidation_fee, quote_asset_delta);
6    let total_fees = ifixed::add(liquidation_fees, insurance_fund_fees);
7    let collateral_change = ifixed::sub(pnl_to_realize, total_fees);
8    position.add_to_collateral_usd(
9      collateral_change,
10     collateral_price
11   );
12   [...]
13 }
```

`settle_liquidated_position` then aggregates the outputs from one or two calls to `reduce_liquidated_position`. Even when the resulting `bad_debt` is nonzero, it still borrows the market vault and transfers the full calculated insurance-fund fee from `vault.collateral_balance` into `vault.insurance_fund_balance`, which the adjacent comment already notes is redundant in bad-debt scenarios.

packages/perpetuals/sources/clearing_house.move

```

1  fun settle_liquidated_position<T>([...]) {
2    [...]
3    // In case of bad debt, the insurance fund will repay back `insurance_fund_fees`
4    // too,
5    // making this transfer redundant in those (hopefully rare) scenarios.
6    let vault = borrow_mut_market_vault(&mut hot_potato.clearing_house);
7    transfer_from_vault_to_insurance_fund<T>(
8      vault,
9      ifixed::div(insurance_fund_fees, hot_potato.collateral_price),
10     scaling_factor
11   )
12 }
```

```

10     );
11     [...]
12 }

```

The transfer helper enforces that the collateral side of the vault already contains the full fee amount before moving it to the insurance-fund bucket. Because `settle_liquidated_position` passes the USD-denominated insurance fee divided by the current collateral price, a sharp collateral-price decline can make the required collateral-unit transfer larger than the current collateral bucket balance. This ordering matters because `execute_liquidation` calls `settle_liquidated_position` before it checks `hot_potato.session_summary.bad_debt` and invokes `handle_bad_debt`. If the insurance-fee transfer aborts inside settlement, execution never reaches the bad-debt branch.

packages/perpetuals/sources/clearing_house.move

```

1  fun execute_liquidation<T>([...]) {
2      [...]
3      // If bad debt occurred, handle it immediately so any later session
4      // Fills settle against the bumped funding rates before changing side.
5      if (hot_potato.session_summary.bad_debt != 0) {
6          handle_bad_debt<T>(
7              &mut hot_potato.clearing_house,
8              hot_potato.session_summary.bad_debt,
9              hot_potato.mark_price,
10             hot_potato.collateral_price,
11             hot_potato.session_summary.is_liqee_long,
12             ch_id
13         );
14     };
15 }

```

`handle_bad_debt` is the path that compares the insurance-fund balance against the bad debt, transfers available insurance funds back into the vault, and socializes any uncovered residual. The pre-transfer assertion can therefore block a liquidation before the code reaches the path that appears intended to cover the deficit from `vault.insurance_fund_balance` or socialize the remainder.

As a result, some liquidations with bad debt may be unexecutable even when the separate insurance-fund bucket has enough collateral to cover the deficit. This is possible for volatile collateral when the collateral price drops sharply after a trader opened near the allowed initial margin: the insurance fee converts into many collateral units, while the liquidation cannot proceed far enough to use the insurance fund for the resulting bad debt.

Remediation

Avoid requiring `vault.collateral_balance` to prepay insurance-fund fees when the liquidation has produced bad debt.

Patch

Resolved in [f5c42ad](#).

Liquidation Fees Can Exceed Maintenance Margin

ID	Severity	Status
OS-AMP-SUG-03	● Informational	✓ Resolved

Description

Market parameter validation accepts liquidation fee settings by comparing the sum of `liquidation_fee` and `insurance_fund_fee` only against `margin_ratio_initial`. In `assert_market_parameters`, market creation first validates that the maintenance margin ratio is below the initial margin ratio, then checks fee bounds, and then calls the IMR-based liquidation-fee helper without comparing the same fee sum to `margin_ratio_maintenance`.

The helper used by creation enforces that the combined liquidation and insurance-fund fees are strictly below `margin_ratio_initial`. Because the helper receives only `margin_ratio_initial`, it cannot enforce the stronger invariant needed for accounts that become liquidatable at the maintenance margin threshold.

packages/perpetuals/sources/market.move

```

1704 public(package) fun assert_liquidation_fees_against_imr(
1705     margin_ratio_initial: u256,
1706     liquidation_fee: u256,
1707     insurance_fund_fee: u256,
1708 ) {
1709     assert!(
1710         ifixed::less_than(
1711             ifixed::add(liquidation_fee, insurance_fund_fee),
1712             margin_ratio_initial,
1713         ),
1714         liquidation_fees_exceed_initial_margin_ratio!()
1715     );
1716 }
```

The same invariant is applied when fee parameters are updated. `set_fee_params` resolves optional fee inputs, validates the registry-level fee caps, and then calls the same IMR-based helper using the existing initial margin ratio before storing the new liquidation and insurance-fund fee rates. Margin-ratio proposals reuse this check from the opposite direction. `create_margin_ratios_proposal` validates the proposed margin-ratio pair, reads the market's current liquidation and insurance-fund fee rates, and checks those fees against the proposed initial margin ratio, but not against the proposed maintenance margin ratio.

Liquidation eligibility is based on the maintenance margin ratio instead. In `settle_liquidated_position`, the liqee's position is liquidatable when its computed margin is below the requirement calculated with `margin_ratio_maint`, so a position can be eligible for liquidation while still having positive margin below IMR but near MMR.

```
packages/perpetuals/sources/clearing_house.move
```

```

1 fun settle_liquidated_position<T>([...]) {
2     [...]
3     // Abort if position can't be liquidated
4     let (margin_before, min_margin_before) = position.compute_margin_and_requirement(
5         hot_potato.collateral_price,
6         mark_price,
7         margin_ratio_maint,
8         collateral_haircut
9     );
10    assert!(
11        ifixed::less_than(margin_before, min_margin_before),
12        position_above_mmr!()
13    );
14    [...]
15 }
```

Once liquidation proceeds, `reduce_liquidated_position` deducts both the liquidation fee and the insurance-fund fee from the liqee's collateral by subtracting their sum from realized PnL before updating collateral. If the accepted fee sum is greater than the maintenance-margin buffer available at the liquidation threshold, a positive-margin liqee may be driven into negative collateral by the fee debit.

```
packages/perpetuals/sources/clearing_house.move
```

```

1 public(package) fun reduce_liquidated_position([...]): (u256, u256, bool, u256, u256,
2     [...]) {
3     // Update collateral for liqee
4     let insurance_fund_fees = ifixed::mul(insurance_fund_fee, quote_asset_delta);
5     let liquidation_fees = ifixed::mul(liquidation_fee, quote_asset_delta);
6     let total_fees = ifixed::add(liquidation_fees, insurance_fund_fees);
7     let collateral_change = ifixed::sub(pnl_to_realize, total_fees);
8     position.add_to_collateral_usd(
9         collateral_change,
10        collateral_price
11    );
12    [...]
13 }
```

The liquidation path then treats negative collateral as bad debt by resetting the position collateral and returning the negative amount as `bad_debt`. The surrounding liquidation flow handles that bad debt immediately with insurance funds when available, or socializes the remaining amount through the market state when insurance funds are insufficient. This means configurations accepted by market creation, fee updates, or margin-ratio proposals may allow liquidator and

insurance-fund fees to exceed the margin condition that made the account liquidatable, shifting the shortfall into the protocol's bad-debt mechanisms.

```
packages/perpetuals/sources/clearing_house.move
```

```

1 public(package) fun reduce_liquidated_position(...): (u256, u256, bool, u256, u256,
  u256, u256, u256, u256, u256) {
2     [...]
3     // Insurance funds will be used to complete liquidation if needed.
4     // To use IFs, position's collateral needs to be negative (indicating bad debt).
5     // A process to socialize debt is executed if required.
6     let mut bad_debt = 0;
7     if (ifixed::is_neg(position.collateral())) {
8         bad_debt = position.reset_collateral();
9     };
10    [...]
11 }
```

Remediation

Validate the combined `liquidation_fee` and `insurance_fund_fee` against `margin_ratio_maintenance` everywhere the relevant fee or margin tuple can be created or changed.

Patch

Resolved in [f5c42ad](#).

Resume Has No Cancel-Only Grace

ID	Severity	Status
OS-AMP-SUG-04	● Informational	✓ Resolved

Description

The market pause flow uses a single `paused` state for both emergency halt behavior and later reactivation. In `admin_pause_market`, the package pause guardian path authorizes the pause and returns if the market is already paused, otherwise it records the paused state without recording a pause epoch, disabling only selected operations, or distinguishing cancellation from trading.

packages/perpetuals/sources/clearing_house.move

```

891 public fun admin_pause_market<T>(
892     clearing_house: &mut ClearingHouse<T>,
893     cap: &AuthorityCap<PACKAGE, PAUSE_GUARDIAN>,
894     registry: &Registry,
895 ) {
896     clearing_house.assert_package_version();
897     registry.assert_package_version();
898     registry.assert_multiton_authority_cap_is_authorized(cap);
899     if (clearing_house.paused) return;
900
901     clearing_house.paused = true;
902 }
```

The vendor-scoped `pause_market` path follows the same state model after confirming the vendor owns the clearing house. As shown in `pause_market`, the vendor pause also only moves the market into the shared paused state, so both pause entrypoints converge on the same boolean state.

packages/perpetuals/sources/clearing_house.move

```

947 public fun pause_market<VendorKey, T>(
948     clearing_house: &mut ClearingHouse<T>,
949     cap: &AuthorityCap<VENDOR<VendorKey>, PAUSE_GUARDIAN>,
950     registry: &Registry,
951 ) {
952     clearing_house.assert_package_version();
953     registry.assert_package_version();
954     registry.assert_multiton_authority_cap_is_authorized(cap);
955     registry.assert_vendor_has_ownership_over_clearing_house(cap,
956         clearing_house.id.as_inner());
957     if (clearing_house.paused) return;
958
959     clearing_house.paused = true;
960 }
```

While that state is active, account owners cannot use the normal cancellation path to remove resting orders. `cancel_orders` validates the account authority and package version, then gates cancellation on the market not being paused before it reaches the internal order cancellation routine.

```
packages/perpetuals/sources/clearing_house.move
```

```
1968 public fun cancel_orders<T, Role>(
1969     clearing_house: &mut ClearingHouse<T>,
1970     cap: &AuthorityCap<ACCOUNT, Role>,
1971     account: &Account<T>,
1972     order_ids: vector<u128>,
1973 ) {
1974     account.assert_authority_cap_is_valid(cap);
1975     clearing_house.assert_package_version();
1976     clearing_house.assert_market_is_not_paused();
1977
1978     clearing_house.cancel_orders_!(
1979         account.account_id(),
1980         &order_ids,
1981     );
1982 }
```

`try_cancel_orders` is also paused-gated before it inspects the orderbook or updates pending-order accounting. In `try_cancel_orders`, the function can return per-order cancellation results only after the same unpaused assertion succeeds, so it does not provide a cancel-only escape path during a pause.

```
packages/perpetuals/sources/clearing_house.move
```

```
1 public fun try_cancel_orders<T, Role>(
2     clearing_house: &mut ClearingHouse<T>,
3     cap: &AuthorityCap<ACCOUNT, Role>,
4     account: &Account<T>,
5     order_ids: &vector<u128>,
6 ): vector<bool> {
7     account.assert_authority_cap_is_valid(cap);
8     clearing_house.assert_package_version();
9     clearing_house.assert_market_is_not_paused();
10     [...]
11 }
```

The same paused state also blocks users from starting a trading session. In `start_session`, the session hot potato is created only after the function verifies that the market is not paused, so taker order placement and cancellation both become available again through normal public flows only once the paused flag is cleared. Resume does not introduce an intermediate cancellation-

only period for package pause guardians. As shown in `admin_resume_market`, the function rejects closed markets, returns if the market is not currently paused, and otherwise flips the same paused state back to active trading without invalidating or revalidating orders that were posted before the pause.

packages/perpetuals/sources/clearing_house.move

```

905 public fun admin_resume_market<T>(  

906     clearing_house: &mut ClearingHouse<T>,  

907     cap: &AuthorityCap<PACKAGE, PAUSE_GUARDIAN>,  

908     registry: &Registry,  

909 ) {  

910     clearing_house.assert_package_version();  

911     registry.assert_package_version();  

912     registry.assert_multiton_authority_cap_is_authorized(cap);  

913     clearing_house.assert_market_is_not_closed();  

914  

915     if (!clearing_house.paused) return;  

916  

917     clearing_house.paused = false;  

918 }

```

The vendor-scoped `resume_market` path mirrors that behavior after the vendor ownership check. Once either resume path clears the paused state, the code read around order execution shows normal sessions can again walk opposite-side orderbook entries and fill non-expired, otherwise valid maker orders at their stored order prices. This creates a race after maintenance or incident response when external prices moved during the pause: resting-order owners were unable to cancel while paused, while the first post-resume actions can include both owner cancellations and adverse fills against stale orders.

packages/perpetuals/sources/clearing_house.move

```

962 public fun resume_market<VendorKey, T>(  

963     clearing_house: &mut ClearingHouse<T>,  

964     cap: &AuthorityCap<VENDOR<VendorKey>, PAUSE_GUARDIAN>,  

965     registry: &Registry,  

966 ) {  

967     clearing_house.assert_package_version();  

968     registry.assert_package_version();  

969     registry.assert_multiton_authority_cap_is_authorized(cap);  

970     registry.assert_vendor_has_ownership_over_clearing_house(cap,  

971         clearing_house.id.as_inner());  

971     clearing_house.assert_market_is_not_closed();  

972  

973     if (!clearing_house.paused) return;  

974 }

```

```
975 |      clearing_house.paused = false;  
976 |  }
```

Remediation

Introduce a safer resume state for paused markets.

Patch

Resolved in [f5c42ad](#).

Vendor Assistants Can Escalate To Perpetuals Vendor Admin

ID	Severity	Status
OS-AMP-SUG-05	● Informational	✗ To-do

Description

`registry::register_vendor` creates the Perpetuals vendor root for a `VendorKey`, but its input authority is an upstream vendor cap parameterized as `ADMIN_OR_ASSISTANT` while its return value is the internal Perpetuals `AuthorityCap<VENDOR<VendorKey>, ADMIN>`. As shown in `register_vendor`, the function initializes the vendor clearing-house dynamic field and then returns the newly created internal vendor admin cap to the caller.

packages/perpetuals/sources/registry.move

```

899 public fun register_vendor<VendorKey, ADMIN_OR_ASSISTANT>(
900     registry: &mut Registry,
901     _: &AuthorityCap<vendor::authority::VENDOR<VendorKey>, ADMIN_OR_ASSISTANT>,
902     _: &mut VendorMetadata<VendorKey>,
903 ): AuthorityCap<VENDOR<VendorKey>, ADMIN> {
904     registry.assert_package_version();
905     perpetuals::authority::assert_is_admin_or_assistant<ADMIN_OR_ASSISTANT>();
906
907     df::add(
908         &mut registry.id,
909         keys::vendor_clearing_house_key<VendorKey>(),
910         vector<ID>[],
911     );
912
913     perpetuals::authority::create_vendor_admin_cap(&mut registry.id)
914 }
```

The role gate in `register_vendor` checks only that the upstream role is either admin or assistant. An upstream assistant therefore satisfies the check even though the handoff mints the full internal Perpetuals vendor admin cap rather than an internal assistant cap.

This crosses the intended role boundary. In the upstream vendor package, `ASSISTANT` is a subset role created by the vendor admin. Perpetuals treats that subset role as sufficient to mint the internal full-admin cap, so revoking or withholding the upstream vendor admin no longer protects the vendor's Perpetuals namespace after the assistant performs first registration.

Remediation

Do not let an upstream assistant mint the internal Perpetuals vendor admin cap on its own.

Socialized Bad Debt Can Be Underpaid After Collateral Price Moves

ID	Severity	Status
OS-AMP-SUG-06	● Informational	✗ To-do

Description

Liquidation can create a collateral-denominated deficit on the liquidated account. In `reduce_liquidated_position`, the account's PnL and liquidation-related fees are applied through the liquidation session's `collateral_price`. If the resulting collateral balance is negative, the code resets the position balance and records the absolute value as `bad_debt`, as shown in the liquidation accounting flow.

packages/perpetuals/sources/clearing_house.move

```

1 public(package) fun reduce_liquidated_position([...]): (u256, u256, bool, u256, u256,
  u256, u256, u256, u256) {
2     [...]
3     // Insurance funds will be used to complete liquidation if needed.
4     // To use IFs, position's collateral needs to be negative (indicating bad debt).
5     // A process to socialize debt is executed if required.
6     let mut bad_debt = 0;
7     if (ifixed::is_neg(position.collateral())) {
8         bad_debt = position.reset_collateral();
9     };
10    [...]
11 }

```

`reset_collateral` confirms that this returned value is the absolute value of the old position collateral balance after the position collateral is set to zero. The deficit being handed to bad debt handling is therefore denominated in collateral units.

packages/perpetuals/sources/position.move

```

1 /// Reset position's collateral and returns the absolute value
2 /// of the old collateral amount (amount of debt)
3 public(package) fun reset_collateral(position: &mut Position): u256 {
4     let collateral = position.collateral;
5     position.collateral = 0;
6     ifixed::abs(collateral)
7 }

```

Reset returns the absolute collateral deficit.

When the insurance fund cannot cover the full collateral deficit, `handle_bad_debt` transfers the available insurance fund collateral and converts only the uncovered remainder into `amount_to_socialize` by multiplying by the liquidation session's collateral price. This changes the

residual from a collateral-unit shortfall into a USD-denominated value before it is applied through funding.

`try_socialize_bad_debt` then divides that USD amount by market open interest and records it in the cumulative funding rate for the side opposite the liquidated position through `market::add_bad_debt_to_market`. The charged positions are therefore not debited for collateral in this path; they inherit a funding-rate delta that will be settled later.

packages/perpetuals/sources/clearing_house.move

```

1  fun try_socialize_bad_debt([...]) {
2      [...]
3      // If the amount of bad debt is below the thresholds, proceed with socializing
4      // losses, otherwise abort and handle with ADL.
5      // Checks:
6      // - Notional bad debt amount below `max_bad_debt`
7      // - Bad debt amount relative to `open_interest * mark_price` below
8      //   `max_socialize_losses_mr_decrease`
9      let social_fundings = ifixed::div(amount_to_socialize, open_interest);
10     assert!(ifixed::less_than_eq(amount_to_socialize, max_bad_debt),
11         bad_debt_notional_above_threshold!());
12     assert!(ifixed::less_than_eq(
13         ifixed::div(social_fundings, mark_price),
14         max_socialize_losses_mr_decrease
15     ),
16         bad_debt_socialization_above_threshold!());
17     [...]
18 }

```

The converted amount is applied as a funding-rate delta.

The later settlement path in `settle_position_funding` converts the funding numerator back into collateral units by dividing by the `collateral_price` supplied to that settlement call. Public funding settlement, session start, deallocation, and settlement-price flows read or use the collateral price at their own execution time, so if the collateral price changes after bad debt is socialized, the number of collateral units collected from charged positions can differ from the uncovered collateral units originally reset. In particular, collateral appreciation can cause the charged side to pay fewer collateral units than `bad_debt - insurance_fund_value`, leaving the vault with less collateral than the reset deficit required. Collateral depreciation can over-collect relative to that original unit deficit.

Remediation

Keep the socialized bad debt obligation collateral-denominated, or bind its USD conversion to the liquidation-time price through settlement.

Asynchronous Funding Conversion Enables Collateral Arbitrage

ID	Severity	Status
OS-AMP-SUG-07	● Informational	✗ To-do

Description

Market funding is accumulated as cumulative long and short funding rates derived from the premium TWAP and elapsed funding period. In `update_fundings`, the market state records the funding-rate delta, but the update does not record or bind the delta to a collateral conversion price.

The position funding direction is determined from the difference between the current and prior cumulative funding rate and the negated base asset amount in `unrealized_funding`. This makes a positive funding-rate delta payable by one side and receivable by the opposite side, while still expressing the transfer as a USD-denominated funding amount before collateral conversion.

```
packages/perpetuals/sources/position.move
```

```

1084 public fun unrealized_funding(
1085     cum_funding_rate_now: u256,
1086     cum_funding_rate_before: u256,
1087     base_asset_amount: u256,
1088 ): u256 {
1089     if (cum_funding_rate_now == cum_funding_rate_before) return 0;
1090
1091     ifixed::mul(
1092         ifixed::sub(cum_funding_rate_now, cum_funding_rate_before),
1093         ifixed::neg(base_asset_amount),
1094     )
1095 }
```

Funding direction derived from rate delta and position size.

The optimized `settle_position_funding` path reconstructs the applicable rate delta from the position snapshot and multiplies it by the absolute base size to produce a funding numerator. This establishes the USD funding amount independently for each position settlement, before any collateral-unit conversion occurs.

The same optimized settlement then divides the funding numerator by the `collateral_price` supplied to that individual call. The receiver branch floors the collateral units credited, while the payer branch ceils the collateral units debited, so the conversion basis is the settlement-time collateral price for that specific account rather than a market-wide price fixed when the funding delta accrued.

The reference settlement path has the same accounting design.

`settle_position_funding_reference` first calculates the funding amount, then applies it to collat-

eral through a helper that receives the current `collateral_price`, confirming that the optimized implementation is not merely diverging from the reference model.

```
packages/perpetuals/sources/position.move
1 public fun settle_position_funding_reference(
2     position: &mut Position,
3     collateral_price: u256,
4     mkt_funding_rate_long: u256,
5     mkt_funding_rate_short: u256,
6     ch_id: &ID,
7     account_id: u64,
8 ) {
9     let funding = calculate_position_funding_internal(
10         position,
11         mkt_funding_rate_long,
12         mkt_funding_rate_short
13     );
14
15     let collateral_after = if (funding != 0) {
16         add_to_collateral_usd(position, funding, collateral_price)
17     } else {
18         position.collateral
19     };
20     [...]
21 }
```

Reference settlement using collateral-price conversion.

The helper used by the reference implementation converts USD funding into collateral units by dividing by the provided collateral price and adding the result to the position collateral balance. This preserves the same dependency on the collateral price supplied at the time the individual account is settled.

The public clearing-house entry point `settle_position_funding` reads the current collateral oracle price, reads the current market cumulative funding rates, and settles the selected `account_id` with those values. Because the entry point takes an arbitrary account identifier and no account capability is shown in this signature, separate accounts can be settled at different times with different collateral oracle prices. After a position is settled, its cumulative funding snapshots are advanced to the market cumulative funding rates. The settled account therefore will not be charged or credited again for the same funding interval, even if the matched counter-account settles the same interval later at a different collateral price.

This makes the funding transfer path-dependent in collateral units. A trader with equal and opposite positions in two accounts could settle the receiving side when collateral is cheaper and the paying side when collateral is more expensive, causing the two position collateral balances to increase in aggregate for the same USD funding interval. The withdrawal path in

`withdraw_free_collateral_to_account_balance` debits the internal position collateral and transfers collateral from the market vault when free collateral is deallocated.

```
packages/perpetuals/sources/clearing_house.move

1 fun process_post([...]) {
2   [...]
3   ): u64 {
4     [...]
5     // Reduce position's collateral. If the vault cannot pay, the transaction aborts
6     // and this debit is rolled back.
7     position.sub_from_collateral(
8       ifixed::from_balance(amount, scaling_factor)
9     );
10
11    if (amount != 0) {
12      withdraw_vault_collateral(
13        borrow_mut_market_vault(clearing_house),
14        account_balance,
15        amount
16      );
17    };
18
19    amount
```

Free collateral withdrawal from the market vault.

The session deallocation path also converts free collateral into a balance amount, subtracts it from the position, and withdraws that amount from the market vault to the account. As a result, any surplus internal collateral created by settling opposite sides at different collateral prices may become withdrawable when the affected positions have free collateral.

Remediation

Use a single market-wide conversion basis for each funding delta instead of each position's settlement-time collateral price.

Realized PnL FX Arbitrage

ID	Severity	Status
OS-AMP-SUG-08	● Informational	✗ To-do

Description

Trading PnL is realized in USD but posted to the position's collateral balance using the collateral oracle price cached in the account's current session. The `SessionHotPotato` stores `collateral_price` when the session begins, and `process_session_hot_potato` later destructures and reuses that cached value for fill settlement and margin accounting. For maker fills, the matched maker order updates base and quote through `add_base_to_position`, computes realized PnL net of maker and integrator fees, and immediately passes that signed USD collateral change into `add_to_collateral_usd` with the session-local collateral price, as shown in `process_fill_maker`.

The taker settlement path follows the same conversion basis. In `process_fill_taker`, the session summary's filled ask and bid amounts are applied to the taker's position, the two realized PnL components are summed, taker and integrator fees are subtracted, and the resulting USD-denominated value is converted into collateral units using the same `collateral_price` argument for that session.

The conversion helper is the common root of the non-zero-sum behavior in collateral units. `add_to_collateral_usd` divides the signed USD amount by the supplied collateral price and adds the result to `position.collateral`, so two offsetting realized USD PnL amounts can produce different collateral-unit deltas if they are settled in different sessions with different collateral prices.

```
packages/perpetuals/sources/position.move
```

```

147 public(package) fun add_to_collateral_usd(
148     position: &mut Position,
149     fixed_usd: u256,
150     collateral_price: u256
151 ): u256 {
152     let fixed = ifixed::div(fixed_usd, collateral_price);
153     position.collateral = ifixed::add(position.collateral, fixed);
154     position.collateral

```

Because the conversion basis is selected independently by each account's closeout session, controlled accounts can separate the profitable and losing legs of an otherwise offsetting position set. If the profitable leg is realized while collateral is cheaper and the offsetting loss is realized later while collateral is more expensive, the two accounts can have zero aggregate realized USD PnL while retaining a positive aggregate collateral-unit balance from the different USD-to-collateral divisors. The remaining exposure is hedged through another controlled account between the two closeouts, so the strategy does not require sustained directional exposure.

After the winning account is flat, the credited collateral can become withdrawable.

`compute_free_collateral` returns all positive collateral when the position has no pending orders and no base amount, and it uses the computed free collateral to subtract it from the position and withdraw vault collateral into the account when `deallocate_free_collateral` is enabled.

packages/perpetuals/sources/position.move

```

1 public fun compute_free_collateral(
2     position: &Position,
3     collateral_price: u256,
4     mark_price: u256,
5     margin_ratio: u256,
6     collateral_haircut: u256
7 ): u256 {
8     // If position is empty, just return all positive collateral as free.
9     if (
10         position.pending_orders == 0 &&
11         position.base_asset_amount == 0
12     ) return ifixed::max(0, position.collateral);
13     [...]
14 }
```

This can let a user extract collateral units from the market vault even though the controlled accounts have no net USD trading profit and no final open base exposure. The excess comes from vault liquidity that backs internal balances, so repeated cycles could leave the vault unable to satisfy later withdrawals or push the shortage into the protocol's loss-handling mechanisms once the imbalance is realized.

Remediation

Use one market-wide conversion basis for both sides of an economic PnL cycle instead of converting each realized leg at that account session's collateral price.

Cleanup-Only Standalone Stops Can Allocate Collateral

ID	Severity	Status
OS-AMP-SUG-09	● Informational	✓ Resolved

Description

In `place_stop_order_standalone`, the standalone stop execution path removes the `StopOrderTicket` from the account before the delegated order is placed. This means the ticket is no longer available once execution proceeds into the stop-order path, even though the later order session may not produce an economic fill or posted order. It then deletes the removed ticket after validating the executor and before checking the encrypted standalone order details and stop trigger conditions. In the ticket-deletion step, the one-shot stop ticket is consumed before the delegated limit or market order result is known.

For standalone limit stops, `place_stop_order_standalone` forwards the committed order parameters into the clearing-house limit-order path, including the selected order type and the `reduce_only` flag. This includes IOC stop-limit orders, so a non-reduce-only IOC stop can enter normal limit-order matching through this branch. After placing the delegated order, `place_stop_order_standalone` finalizes the session by passing the inverse of `reduce_only` as the missing-margin allocation flag. As a result, standalone stop-limit tickets with `reduce_only = false` enable automatic missing-margin allocation during `end_session`.

In `execute_limit_order`, IOC handling sets the remaining taker size to zero before the post-order branch runs. If matching only cleaned an expired, self-trading, or otherwise non-executable maker order with no taker fill, this IOC behavior prevents any residual stop size from being posted to the book. `session_has_activity` treats any maker event as session activity, without requiring nonzero filled base or a posted order. `end_session` uses this predicate to reject empty sessions, so a cleanup-only maker event can allow finalization to continue even when the taker summary has no fill and no posted order.

```
packages/perpetuals/sources/clearing_house.move
```

```

594 fun session_has_activity(
595     session_summary: &SessionSummary,
596     maker_events: &vector<events::FilledMakerOrder>,
597     liqee_account_id: &Option<u64>,
598 ): bool {
599     option::is_some(liqee_account_id)
600     || session_summary.posted_orders != 0
601     || !vector::is_empty(maker_events)
602 }
```

During session finalization, the missing-margin branch is based on the account's current margin shortfall and the `allocate_missing_margin` flag, not on whether the current delegated order produced a fill or posted order. When the account is below initial margin and allocation is enabled,

collateral is added to the position and transferred from the user's account balance into the clearing-house vault.

Together, these behaviors can commit a standalone stop-limit execution that only cleans a maker order, consumes the stop ticket, does not fill the user's stop order, does not post a residual order, and still moves free account collateral into the market vault. The impact is limited to collateral availability and exposure for an account that already has an under-initial-margin position and sufficient account collateral.

Remediation

For standalone stop-limit execution, gate automatic missing-margin allocation on an economic result from the delegated order.

Patch

Resolved in [f5c42ad](#).

Cleanup-Only Stop Executions Can Burn Tickets Without Any Fill

ID	Severity	Status
OS-AMP-SUG-10	● Informational	✓ Resolved

Description

The SLTP stop-order execution path in `place_stop_order_sltp` removes the `StopOrderTicket` from the account before the delegated order is placed. It then deletes the removed ticket through `delete_after_execution!` before any limit or market order is submitted. Because the ticket has already been consumed at this point, later execution that commits as a session leaves no active stop ticket behind.

The standalone stop-order path follows the same structure in `place_stop_order_standalone`. After asserting that the account has the requested ticket, it removes the `StopOrderTicket` before checking the executor and before starting the clearing-house session. It also deletes the ticket with `delete_after_execution!` before verifying the standalone order details, checking the trigger price condition, or placing the delegated order. This makes ticket preservation depend on later abort behavior rather than on whether the stop order actually had economic effect.

For SLTP orders, a `SessionHotPotato` is started, and either a reduce-only limit order or a market order is placed before `end_session` is called. In the limit-order case, this means the already-deleted ticket is only protected if the delegated session aborts before commit. Standalone stop orders similarly place a limit or market order through the session and then call `end_session`. For non-reduce-only standalone orders, the session may allocate missing margin, but the relevant issue is that the order session is finalized after the stop ticket has already been consumed.

`end_session` rejects empty sessions through `session_has_activity`, and that helper treats any non-empty `maker_events` vector as activity. The broader `process_session_hot_potato` flow also only processes taker-side fills when the session summary contains nonzero filled base, so maker-event presence and economic taker fill are not equivalent.

```
packages/perpetuals/sources/clearing_house.move
```

```

594 fun session_has_activity(
595     session_summary: &SessionSummary,
596     maker_events: &vector<events::FilledMakerOrder>,
597     liqee_account_id: &Option<u64>,
598 ): bool {
599     option::is_some(liqee_account_id)
600     || session_summary.posted_orders != 0
601     || !vector::is_empty(maker_events)
602 }
```

Expired maker orders can still create maker events without filling the taker. In `process_fill_maker`, the code sets the allowed maker size to zero for expired maker orders, cancels the maker order,

returns a zero matched taker size, and still constructs a `FilledMakerOrder` event, which makes `maker_events` non-empty even though the taker fill fields remain zero.

For immediate-or-cancel limit orders, `execute_limit_order` explicitly clears the remaining taker size after matching. If the only matched entries were expired makers that matched zero size, this prevents any residual taker size from being posted after the cleanup. It then drops the matched prefix of orderbook entries when `last_order_id_matched` is set. In the cleanup-only case, this removes stale maker orders while leaving the stop order with no taker fill and no posted fallback order.

```
packages/perpetuals/sources/clearing_house.move
1 fun execute_limit_order<T>([...]): (u64, Option<u128>) {
2   [...]
3   // Update size for IMMEDIATE_OR_CANCEL orders
4   if (order_type == constants::immediate_or_cancel!()) {
5     size = 0;
6   };
7
8   // Drop all the orders that have been filled
9   if (last_order_id_matched > 0) {
10    map::batch_drop(map, last_order_id_matched, include_last_order_id);
11  };
12  [...]
13 }
```

This results in a committed stop-order execution whose observable session activity is stale maker cleanup rather than a fill or post by the stop order. A user's stop ticket can therefore be permanently consumed even though the stop order did not execute economically, leaving the user without that active stop protection.

Remediation

Defer irreversible stop-ticket deletion until after the delegated session summary proves that the stop order had the intended economic effect.

Patch

Resolved in [f5c42ad](#).

Empty TWAP Attempts Can Allocate User Collateral

ID	Severity	Status
OS-AMP-SUG-11	● Informational	✓ Resolved

Description

The normal session finalizer rejects sessions that did not produce activity, while the package-only empty-session finalizer forwards to the same processing routine with empty sessions allowed. In `clearing_house::end_session` and `clearing_house::end_session_allow_empty`, the public path passes `false` for empty-session handling, but the TWAP-only path passes `true` into the shared finalizer.

packages/perpetuals/sources/clearing_house.move

```

1 public fun end_session<T>(
2     hot_potato: SessionHotPotato<T>,
3     account: &mut Account<T>,
4     allocate_missing_margin: bool,
5     deallocate_free_collateral: bool,
6 ): (ClearingHouse<T>, SessionSummary) {
7     [...]
8     let (ch, session_summary) = hot_potato.process_session_hot_potato(
9         account,
10        allocate_missing_margin,
11        deallocate_free_collateral,
12        false
13    );
14
15    (ch, session_summary)
16 }
```

Inside `process_session_hot_potato`, the empty-session flag is only used to bypass the activity assertion. Because the empty-session bypass does not skip the later collateral path, an empty finalization can still allocate missing margin. When the position is below initial margin and `allocate_missing_margin` is true, the finalizer computes the missing collateral amount, adds it to the position, and transfers that amount from the supplied `Account` collateral balance into the clearing-house market vault.

The TWAP executor path calls `end_session_allow_empty` with `allocate_missing_margin` set to the inverse of `new_details.reduce_only`. As a result, a non-reduce-only TWAP attempt uses the empty-session finalizer while still authorizing the missing-margin allocation branch.

packages/perpetuals/sources/twap_orders.move

```

1 public fun execute<T>([...]): (SessionSummary, Coin<SUI>, ClearingHouse<T>) {
2     [...]
```

```

3      let (clearing_house, session_summary) = hot_potato.end_session_allow_empty(
4          account,
5          !new_details.reduce_only,
6          false
7      );
8      [...]
9  }

```

`twap_orders::execute` creates the session for the account associated with the ticket, validates the caller as a ticket executor, and derives the fresh and total execution amounts before placing the order. This means an authorized executor can drive the session finalization path for the user's account through the TWAP ticket flow.

packages/perpetuals/sources/twap_orders.move

```

1  public fun execute<T>([...]): (SessionSummary, Coin<SUI>, ClearingHouse<T>) {
2      [...]
3      // Place order
4      let mut hot_potato = clearing_house.start_session_(
5          account.account_id(),
6          oracle_config,
7          base_oracle,
8          collateral_oracle,
9          ctx.gas_price() > ctx.reference_gas_price(),
10         new_details.integrator_info,
11         clock,
12     );
13     let limit_price = derive_twap_limit_price!(&hot_potato, new_details);
14     [...]
15 }

```

The TWAP order placed during execution is an immediate-or-cancel limit order. The surrounding clearing-house logic for immediate-or-cancel orders sets any unfilled size to zero before posting, so when no executable opposing liquidity exists within the derived limit price, the session may have no fill and no posted order.

packages/perpetuals/sources/twap_orders.move

```

1  public fun execute<T>([...]): (SessionSummary, Coin<SUI>, ClearingHouse<T>) {
2      [...]
3      hot_potato.place_limit_order(
4          new_details.side,
5          amount,
6          limit_price,
7          constants::immediate_or_cancel!(), // TWAP orders are always
          IMMEDIATE_OR_CANCEL

```

```

8         option::none(),
9         new_details.reduce_only,
10        option::none(),
11    );
12
13    let (clearing_house, session_summary) = hot_potato.end_session_allow_empty(
14        account,
15        !new_details.reduce_only,
16        false
17    );
18    [...]
19 }

```

After the empty session is finalized, `twap_orders::execute` still applies the TWAP execution result using the session summary's filled amounts. This happens after the possible collateral allocation has already been processed by the clearing-house finalizer.

packages/perpetuals/sources/twap_orders.move

```

1 public fun execute<T>([...]): (SessionSummary, Coin<SUI>, ClearingHouse<T>) {
2     [...]
3     ticket.apply_twap_execution_result!(
4         new_details,
5         session_summary.base_filled_bid(),
6         session_summary.base_filled_ask(),
7         account_id,
8         fresh_amount,
9         amount,
10        clock.timestamp_ms(),
11    );
12    [...]
13 }

```

The result application updates scheduling before checking whether any base amount was actually filled. Therefore, a zero-fill non-reduce-only TWAP attempt can both consume scheduled TWAP capacity and move free account collateral into the market vault when the position is below initial margin, even though the attempt made no execution progress.

Remediation

Do not allow empty delegated TWAP sessions to perform account-balance collateral transfers.

Patch

Resolved in [f5c42ad](#).

TWAP Fresh Chunks Bypass Execution Cap

ID	Severity	Status
OS-AMP-SUG-12	● Informational	✓ Resolved

Description

`TWAPOrderDetails` defines `max_one_execution_amount_bps` as a per-execution limit that applies after any carried backlog is included. The helper `max_one_execution_amount` converts that basis-point value into an absolute amount as a fraction of the total TWAP size, as shown in `max_one_execution_amount`.

```
packages/perpetuals/sources/twap_orders.move
```

```
207 fun max_one_execution_amount(new_details: &TWAPOrderDetails): u64 {
208     (new_details.size * new_details.max_one_execution_amount_bps) / 10_000
209 }
```

The resolver first derives the fresh portion of the next attempt from the executor-provided amount, the unscheduled fresh remainder, and the small-tail merge rule. In `resolve_fresh_and_total_execution_amount`, this fresh portion is only clipped to the remaining fresh amount, or expanded to the whole fresh remainder when the final tail is small enough to merge into the current execution.

The per-execution maximum is then used only to determine how much previously scheduled but unfilled backlog may be added to the fresh amount. If the fresh amount is already greater than or equal to the maximum, `resolve_fresh_and_total_execution_amount` adds no backlog but still returns the fresh amount as the total execution amount, so the computed total is not checked against the cap shown in `max_one_execution_amount`. The amount uncertainty check does not close this gap because it compares the caller-requested amount against `target_chunk_amount`, not against `max_one_execution_amount`. As shown in `assert_amount_within_uncertainty`, a request near the nominal chunk can pass even when the configured per-execution maximum is lower than that nominal chunk.

```
packages/perpetuals/sources/twap_orders.move
```

```
910 public(package) fun assert_amount_within_uncertainty(
911     new_details: &TWAPOrderDetails,
912     desired_amount: u64,
913 ) {
914     // TODO: Replace with `std::u64::mul_div!`.
915     //
916     let allowed_uncertainty = (
917         (new_details.target_chunk_amount() as u128) *
918         (new_details.amount_uncertainty_bps as u128)
919     ) / 10_000;
920 }
```

```

921 |         assert!(
922 |             (desired_amount.diff(new_details.target_chunk_amount()) as u128)
923 |                 <= allowed_incertainity,
924 |             twap_order_amount_uncertainty_violated!()
925 |         );

```

In the execution path, this resolved total is used as the order size. After `assert_twap_order_can_be_executed` validates the caller-provided amount, `execute` calls `resolve_fresh_and_total_execution_amount` and passes the returned total amount directly into `place_limit_order`. As a result, the user-configured per-attempt maximum may be ineffective for normal fresh chunks and for small-tail merges, allowing a valid executor request to schedule a taker order larger than `max_one_execution_amount`.

Remediation

After resolving the fresh amount and any backlog, compute the final total execution amount and assert that it is less than or equal to `new_details.max_one_execution_amount()` before returning it to `execute`. Also validate new TWAP details so `target_chunk_amount` does not exceed `max_one_execution_amount`.

Patch

Resolved in [f5c42ad](#) and [d25c768](#).

Stop Executors Can Add Uncommitted Priority Taker Fees

ID	Severity	Status
OS-AMP-SUG-13	● Informational	✓ Resolved

Description

Stop-order execution delegates placement to an address listed on the ticket, then verifies the supplied encrypted details before opening a clearing-house session. In the SLTP path, the verified hash covers the stop-order fields and IntegratorInfo, but the execution path derives the priority-gas mode from the executor transaction context when calling `start_session_`, as shown in `place_stop_order_sltp`.

packages/perpetuals/sources/stop_orders.move

```

1 public fun place_stop_order_sltp<T>([...]): (SessionSummary, Coin<SUI>,
  ClearingHouse<T>) {
2     [...]
3     // Place stop order
4     let mut hot_potato = clearing_house.start_session_(
5         account.account_id(),
6         oracle_config,
7         base_oracle,
8         collateral_oracle,
9         ctx.gas_price() > ctx.reference_gas_price(),
10        integrator_info,
11        clock,
12    );
13    [...]
14 }
```

The standalone stop-order path follows the same pattern. `place_stop_order_standalone` starts the trading session using the executor transaction gas-price comparison rather than a value committed in the stop-order details.

packages/perpetuals/sources/stop_orders.move

```

1 public fun place_stop_order_standalone<T>([...]): (SessionSummary, Coin<SUI>,
  ClearingHouse<T>) {
2     [...]
3     // Place stop order
4     let mut hot_potato = clearing_house.start_session_(
5         account.account_id(),
6         oracle_config,
7         base_oracle,
8         collateral_oracle,
9         ctx.gas_price() > ctx.reference_gas_price(),
```

```

10     integrator_info,
11     clock,
12 );
13 [...]
14 }

```

`start_session_` stores the supplied priority-gas boolean in the `SessionHotPotato`, and session finalization uses that value when processing any taker fill. As shown in the clearing-house settlement flow, a session with taker-filled size resolves the configured priority taker fee when `uses_priority_gas_price` is true and passes that value into taker fill processing, where it is added to the taker fee before being multiplied by filled volume and debited from the position collateral.

packages/perpetuals/sources/clearing_house.move

```

1  fun process_session_hot_potato<T>([...]): (ClearingHouse<T>, SessionSummary) {
2      [...]
3      // Process size matched by taker
4      let (
5          taker_fees,
6          taker_open_interest,
7          taker_integrator_fees
8      ) = if (
9          session_summary.base_filled_ask != 0 ||
10         session_summary.base_filled_bid != 0
11     ) {
12         let priority_taker_fee = if (uses_priority_gas_price) {
13             market::resolve_priority_taker_fee(priority_taker_fee_config)
14         } else 0;
15         process_fill_taker(
16             position,
17             collateral_price,
18             account_id,
19             &session_summary,
20             taker_fee,
21             priority_taker_fee,
22             &integrator_info,
23             &ch_id,
24             mark_price
25         )
26     } else (0, 0, 0);
27     [...]
28 }

```

The priority taker fee is a configurable market parameter. The default market-parameter constructor initializes it to 10 bps, and the validation shown in `assert_priority_taker_fee` only

bounds a configured value by the registry's maximum absolute taker fee. Because the stop-order hash inputs do not include a priority-fee preference or cap, the user can commit `IntegratorInfo` but cannot commit whether this additional taker fee is acceptable for delegated stop execution.

```
packages/perpetuals/sources/market.move
```

```
1639 fun assert_priority_taker_fee(
1640     registry_config: &RegistryConfig,
1641     priority_taker_fee: Option<u256>
1642 ) {
1643     if (option::is_none(&priority_taker_fee)) return;
1644
1645     let priority_taker_fee = *option::borrow(&priority_taker_fee);
1646     assert!(
1647         ifixed::greater_than_eq(priority_taker_fee, 0) &&
1648         ifixed::less_than_eq(priority_taker_fee,
1649             registry_config.max_abs_taker_fee()),
1650         invalid_priority_taker_fee!()
1651     );
1652 }
```

As a result, an authorized stop executor can execute an otherwise valid stop order with a transaction gas price above the reference gas price and cause the user's taker fill to include the configured priority taker fee. The marginal executor choice is independent of the user's filled notional, while the resulting fee is charged against the user's position collateral when the stop order produces a taker fill.

Remediation

Do not leave the priority-fee decision to the stop-order executor.

Patch

Resolved in [f5c42ad](#).

TWAP Executors Can Force Priority Taker Fees

ID	Severity	Status
OS-AMP-SUG-14	● Informational	✓ Resolved

Description

`TWAPOrderDetails` defines the order fields that a user commits for TWAP execution, but it does not include a field that records whether the user permits priority-gas execution or caps the additional priority taker fee. The ticket validation relies on a hash of the `TWAPOrderDetails` BCS bytes. Because the priority-fee consent is not part of that structure, it is also not part of the committed hash used to validate the details supplied during execution.

```
packages/perpetuals/sources/twap_orders.move
```

```
198 fun compute_hash(order_snapshot: &TWAPOrderDetails): vector<u8> {
199     sui::hash::blake2b256(&std::bcs::to_bytes(order_snapshot))
200 }
```

In `twap_orders::execute`, the caller must be one of the ticket executors and the supplied details must pass the ticket's TWAP execution checks, but the clearing-house session is started with `uses_priority_gas_price` derived from the executor transaction's gas price comparison. This means an authorized executor can submit an otherwise valid TWAP chunk with a transaction gas price above the reference gas price and set the session's priority flag without changing the committed TWAP details.

When the clearing-house session is settled, filled taker activity branches on the cached `uses_priority_gas_price` flag and resolves a configured priority taker fee when the flag is set. The settlement logic then passes that priority fee into taker fill processing for the account associated with the session. `calculate_taker_fee_breakdown` adds any nonzero priority taker fee to the market taker fee before multiplying the resulting taker fee by the filled taker volume. As a result, the executor-controlled priority flag can increase the taker fees charged for a filled TWAP execution.

The resulting fee impact is realized in `process_fill_taker`, which applies the calculated taker fees and integrator fees against the taker's position collateral after settling taker PnL. Therefore, although the executor cannot alter committed order parameters such as side, size, slippage, or integrator information, they can cause the user's position to pay the additional priority taker fee on each filled TWAP chunk by choosing priority gas for the execution transaction.

Remediation

Do not leave the priority-fee decision to the TWAP executor.

Patch

Resolved in [f5c42ad](#).

Stop Order Executions Bypass Vault Pending Order Cap

ID	Severity	Status
OS-AMP-SUG-15	● Informational	✓ Resolved

Description

The vault exposes a per-position pending-order cap through `max_pending_orders_per_position`. Direct limit-order placement enforces this cap after order placement by calling `assert_pending_orders_within_limit`, which rejects any position whose `pending_order_count()` exceeds the configured vault limit.

The stop-order execution paths do not apply the same post-placement check. Both SL/TP and standalone stop-order execution call the perpetuals stop-order module and then only update vault clearing-house membership, leaving any newly posted resting limit order unchecked. Because stop-limit execution can post resting orders, this omission allows a position to exceed the cap enforced by direct `place_limit_order` and `end_perpetuals_session` flows.

A vault position may accumulate more pending orders than the configured limit permits. This undermines the cap intended to bound order-management complexity, increases cancellation and maintenance costs, and may affect force-withdraw liveness because all pending orders must be canceled before assets can be settled.

Remediation

Apply the same pending-order cap check used by direct order placement after each stop-order execution path that can create or update a position.

Patch

Resolved in [9e38e4c](#).

Frozen Oracles Remain Temporarily Consumable

ID	Severity	Status
OS-AMP-SUG-16	● Informational	○ Ack'd

Description

Freezing a `PriceFeedStorage` stops further oracle updates but does not revoke read access or make `price_feed(...)` fail. Consequently, both iPerps' `price!` macro and the vault's `get_price` can continue consuming the last stored `(price, timestamp_ms)` from the frozen oracle.

```
packages/perpetuals/sources/market.move
```

```
1 macro fun price([...]): u256 {
2     let (price, timestamp_ms) = price_feed::price_and_timestamp_ms(
3         price_feed_storage::price_feed($oracle, $source_id),
4     );
5     [...]
6 }
```

```
packages/market-making-vault/sources/vault.move
```

```
1 public(package) fun start_deposit_session<L, C>([...]): DepositSession<L, C> {
2     [...]
3     let collateral_price = get_price(
4         collateral_oracle,
5         clock,
6         vault.vault_params.collateral_source_id,
7         vault.vault_params.collateral_pfs_tolerance
8     );
9     [...]
10 }
```

Thus, consumption remains possible until the unchanged timestamp exceeds `oracle_tolerance`, so freezing is not an immediate circuit breaker and leaves a bounded stale-price usage window.

Remediation

Ensuring all consumers reject frozen prices regardless of timestamp freshness.

Patch

Aftermath acknowledged this issue and stated that affected values would remain valid only for the short oracle-tolerance window. For perpetuals specifically, they noted that if the oracle is frozen, the corresponding perpetuals market should likely also be frozen. For other domains, they stated that this behavior will be explicitly documented.

Manipulated Book Marks Underfund New Positions

ID	Severity	Status
OS-AMP-SUG-17	● Informational	○ Ack'd

Description

it is possible to temporarily depress the order-book midpoint and call `update_twaps`, contaminating the spread TWAP used to derive the session mark price. `start_session_` caches this manipulated mark, and `end_session_` uses it to calculate the position's margin requirement and automatic collateral allocation. Because the snapshot is not revalidated after the fair book is restored, the protocol may allocate materially less collateral than the position requires at an unmanipulated mark. The newly opened position can therefore become immediately insolvent, shifting the resulting bad debt to the insurance fund or market counterparties.

Remediation

Use an oracle-anchored manipulation-resistant mark for margin allocation, and recompute final position health against a fresh post-session mark before committing.

Patch

Aftermath acknowledged this issue.

Stale Pre-Fill Marks Permit Maker Insolvency

ID	Severity	Status
OS-AMP-SUG-18	● Informational	○ Ack'd

Description

`process_fill_maker` validates maker solvency using the mark price cached at session start, which incorporates the pre-fill order-book price. Executing the fill may remove or resize the best order and shift the book-derived mark price adversely to the maker.

Because neither `process_fill_maker` nor `end_session_` recomputes the mark and revalidates the already-mutated maker position, the stale check can incorrectly accept the fill. The maker may therefore finish insolvent or without sufficient liquidation-fee reserve, exposing the market to bad debt.

Remediation

Recompute the mark from the projected post-fill order book and retain the maker fill only if the maker remains solvent with sufficient liquidation-fee reserve at that updated mark.

Patch

Aftermath acknowledged this issue.

Endpoint Quote Manipulation Distorts Funding Intervals

ID	Severity	Status
OS-AMP-SUG-19	● Informational	○ Ack'd

Description

Cancelling a best-priced order immediately changes the cached bid or ask and thus the order-book midpoint, without first checkpointing `MarketState.premium_twap`. This makes it possible to manipulate the endpoint quote immediately before a scheduled `update_funding` call. `market::update_twap` assigns this single `price_now` observation the entire elapsed `since_last` weight, effectively treating it as though it prevailed throughout the preceding interval.

The distorted premium TWAP is then incorporated into the cumulative funding indexes used to settle trader positions. This enables brief order-book manipulation to create artificial funding transfers between longs and shorts without maintaining the manipulated price over time.

Remediation

Checkpoint the premium TWAP using the pre-change book price before every best-quote mutation, so endpoint quotes cannot retroactively represent the elapsed interval.

Patch

Aftermath acknowledged this issue.

Appendix A.

Vulnerability rating scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the general findings.

Severity	Description
● Critical	Vulnerabilities that immediately result in a loss of availability, consensus divergence, or exploitable memory corruption with minimal preconditions.
● High	Vulnerabilities that may result in consensus divergence or denial of service but require specific transaction or network conditions to exploit.
● Medium	Vulnerabilities that may result in incorrect behavior or state divergence under specific configurations or edge cases.
● Low	Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.
● Informational	Best practices to mitigate future security risks. These are classified as general findings.

Resolution status is one of ✓ Resolved, ○ Ack'd, or ✗ To-do.

Appendix B.

Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process