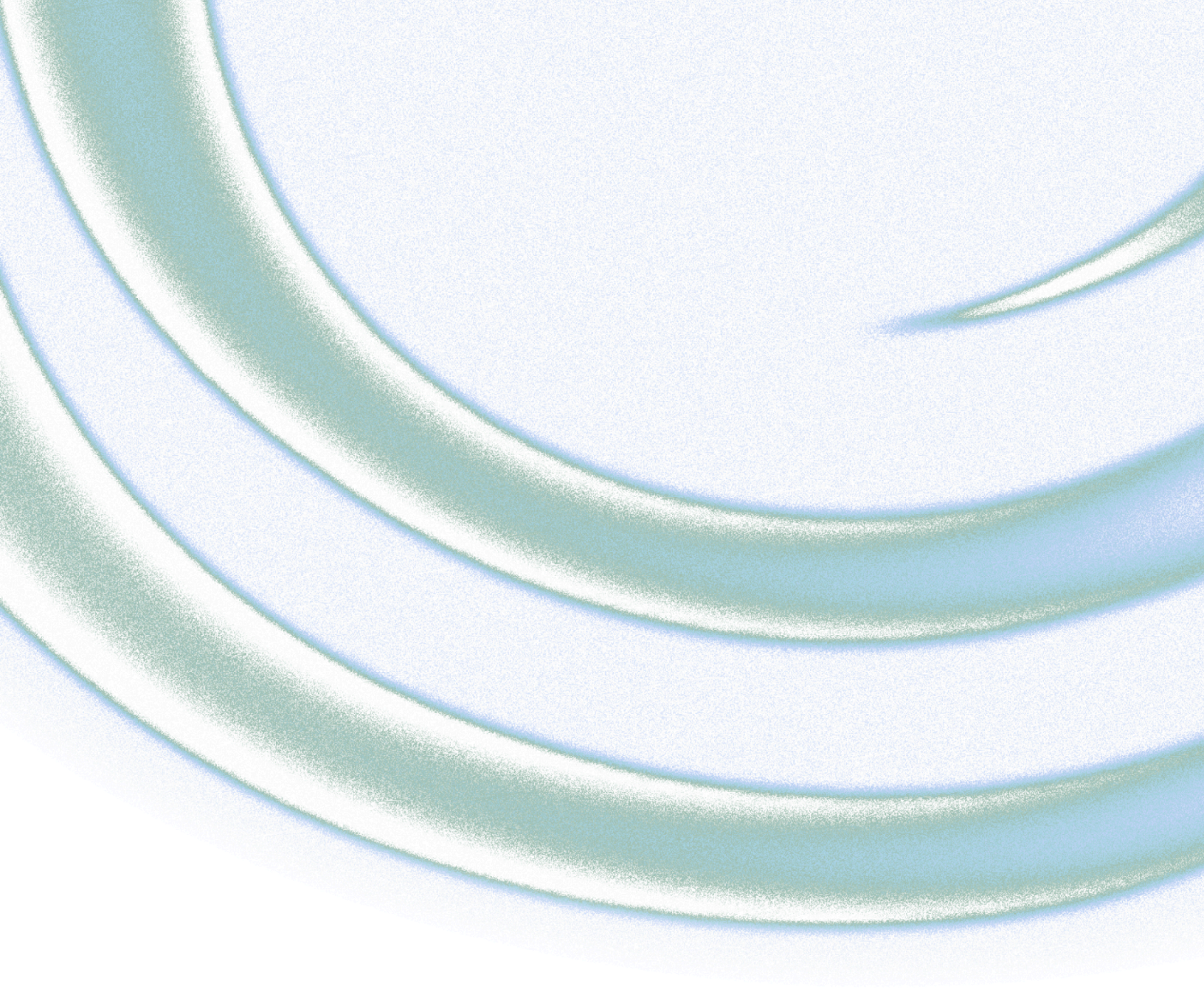

OtterSec

Velocity Protocol v2

Security Assessment

Prepared for
Velocity Exchange

Prepared by
Otter Audits, LLC

Prepared on
September 3rd, 2026

Jinwoo Lee
jin@osec.io

Robert Chen
r@osec.io

Table of Contents

1. Executive Summary 9
1.1. Overview 9
1.2. Key Findings 9
2. Scope 10
3. Findings 11
3.1. Findings Summary 11
4. Advisories 12
OS-VEP-ADV-00 ● Native Fast-Path Trusts Caller-Supplied State Bytes for Hot-Admin Authorization 20
OS-VEP-ADV-01 ● Bulk Order placement Can Drop the Earlier Risk-Increasing Margin Check 23
OS-VEP-ADV-02 ● AMM JIT Can Fill Inside DLOB Matches After AMM Fill Gates Say Unavailable 25
OS-VEP-ADV-03 ● Bid/Ask TWAP Crank Lets Caller-Curated DLOB Depth Set Funding Price 28
OS-VEP-ADV-04 ● Update Perp Bid/Ask TWAP Can Borrow Another User's Keeper Stake Gate 31
OS-VEP-ADV-05 ● Direct Liquidation Accepts a Liability Oracle That Is Stale for Margin 33
OS-VEP-ADV-06 ● Spot Bankruptcy Socializes Loss While Same-Market Revenue Stays Time-Locked 34
OS-VEP-ADV-07 ● Pending IF Fee Sweep Can Force Bankruptcy Socialization 36
OS-VEP-ADV-08 ● Direct Spot Liquidation Accepts a Margin-Invalid Deposit Oracle 39
OS-VEP-ADV-09 ● Positive Unsettled PnL Blocks Liquidation of Larger Riskier Negative PnL 41
OS-VEP-ADV-10 ● Dust Deposit Bricks Spot-Bankruptcy Resolution When Loss Exceeds Deposits 44
OS-VEP-ADV-11 ● IF Unstake-Cancel Penalizes Canceling Staker for Vault Donations 47
OS-VEP-ADV-12 ● IF Request-Remove Freezes Exit Value Before Settling Already-Due Revenue 49
OS-VEP-ADV-13 ● IF Add Accepts a Positive Deposit That Mints Zero Shares 51
OS-VEP-ADV-14 ● Revenue-Settle APR Cap Derived From Live IF-Vault Balance Is Donation-Inflatable 54

OS-VEP-ADV-15	Revenue-Share Sweep Drains PnL Pool Without Reserving Live `net_user_pnl`	57
OS-VEP-ADV-16	Stale Builder Revenue-Share Row Overcharges a Later Non-Builder Order With the Same Order ID	60
OS-VEP-ADV-17	Pyth Lazer Replay Refreshes a Stale Price as Slot-Fresh (No Max-Age Check)	63
OS-VEP-ADV-18	Cross-Market Builder Sidecar Can Sweep The Wrong PnL Pool	66
OS-VEP-ADV-19	Perp Bankruptcy Socialization Double-Credits The AMM Via A Stale last_cumulative_funding_rate	68
OS-VEP-ADV-20	AMM Summary Counts Deferred Revenue Share As Spendable Fee Equity	71
OS-VEP-ADV-21	Unswept Referral Rewards Let New Depositors Dilute Existing Vault Shareholders	73
OS-VEP-ADV-22	Late Vault Entrant Can Capture A Pending Withdrawer's Accrued Reward	75
OS-VEP-ADV-23	Builder-Reward Donation Can Burn A Cancelling Victim's Vault Claim	78
OS-VEP-ADV-24	Unvalidated Vault Denomination Oracle Can Overmint Depositor Shares	81
OS-VEP-ADV-25	Zero Utilization Backdates Interest Onto The First Later Borrower	83
OS-VEP-ADV-26	Unchecked Swap Outputs Turn The Deposit Cap Into A Global Exit And Repayment Lock	84
OS-VEP-ADV-27	Post-Bankruptcy Revenue-Share Credit Escapes Setoff And Overdraws Insurance	87
OS-VEP-ADV-28	Strictly Reducing Swap Trusts An Invalid Oracle In Its Loss Bound	90
OS-VEP-ADV-29	Negative Expiry Price Is Clipped Out Of Margin And Equity	93
OS-VEP-ADV-30	Withdraw Margin Uses Unaccrued Debt From Read-Only Spot Markets	96
OS-VEP-ADV-31	Vault Deposit Mints Shares Before Target-Market Interest Refresh	100
OS-VEP-ADV-32	Vault Withdrawal Requests And Cancellations Use Stale Interest Epochs	102
OS-VEP-ADV-33	IF Cancel Before Revenue Settlement Evades Pending-Window Share Forfeiture	106

OS-VEP-ADV-34	● Reducing DLOB Match Crystallizes A Too-Uncertain Oracle Loss Below The Equity Floor	108
OS-VEP-ADV-35	● Standard Fill Margin Credits A Stale Deposit Into DLOB Bad Debt	112
OS-VEP-ADV-36	● Standard Fill Underprices A Stale Liability Into DLOB Bad Debt	115
OS-VEP-ADV-37	● Bid/Ask TWAP Crank Sets A Third Party's Forced-Close Auction Band	118
OS-VEP-ADV-38	● Standard Fill Uses Unaccrued Spot Debt To Create Bad Debt	122
OS-VEP-ADV-39	● Oracle Spread Check Uses Non-Strict Prices	125
OS-VEP-ADV-40	● Pyth Ownership Check Accepts Invalid Oracles	129
OS-VEP-ADV-41	● Direct Deposit Ignores the Market-Scoped Deposit Pause Bit	130
OS-VEP-ADV-42	● Same-Market transfer-deposit Bypasses Recipient Deposit Admission	133
OS-VEP-ADV-43	● `liquidate_spot_with_swap` Bricked by Stale Account-Index Guard	135
OS-VEP-ADV-44	● Reduce-Only Maker Order Can Overfill When AMM Fulfillment Is Paused	137
OS-VEP-ADV-45	● ReduceOnly Markets Can Still Fill Legacy Non-Reduce-Only Perp Orders	139
OS-VEP-ADV-46	● Unbounded Order Expiry Overflows AMM Fallback Pricing And Aborts Fills	141
OS-VEP-ADV-47	● Trigger-Order Path Uses The Exchange-Pause Guard Instead Of The Fill-Pause Guard	143
OS-VEP-ADV-48	● Swap-Backed Spot Liquidation Bypasses The Max-Pct Liquidation Throttle	146
OS-VEP-ADV-49	● PnL-vs-Spot Liquidations Skip the 5-Minute Spot Oracle TWAP Band	148
OS-VEP-ADV-50	● Spot Bankruptcy Clears the Borrow Using Stale Cumulative Borrow Interest	149
OS-VEP-ADV-51	● Perp PnL Deficit Recap Draws From the Insurance Fund on a Stale PnL Pool	151
OS-VEP-ADV-52	● Transfer Pools Bypasses Destination Market Deposit Caps And Status Gate	154
OS-VEP-ADV-53	● Funding Pause Does Not Stop Spot Interest Accrual Through Public Paths	156

OS-VEP-ADV-54	PnL-vs-Spot Liquidation Can Worsen Account Health When Liquidator Fee Exceeds Buffer	159
OS-VEP-ADV-55	`liquidate_perp_with_fill` Budgets IF/Protocol Fees on a Stale Fee Basis	162
OS-VEP-ADV-56	`liquidate_perp_with_fill` Executes a Real Fill During Global FillPaused	165
OS-VEP-ADV-57	Swap-Backed Spot Liquidation Bypasses Spot Deposit/Withdraw Pauses	167
OS-VEP-ADV-58	Permissionless `set_user_status_to_being_liquidated` Ignores Market-Local Liquidation Pauses	170
OS-VEP-ADV-59	IF Stake Rebase Floors a Pending Unstake Request to Zero and Strands the Stake	172
OS-VEP-ADV-60	IF-Add Auto-Settles Revenue While Global Withdraw-Paused Blocks the Direct Settle Path	175
OS-VEP-ADV-61	Revenue-Pool Deposit Credits at Stale Cumulative Deposit Interest	177
OS-VEP-ADV-62	Direct Revenue-to-IF Settlement Ignores the Market-Scoped SpotOperation::Withdraw Pause	180
OS-VEP-ADV-63	Revenue-Pool Deposit Bypasses the Market-Scoped SpotOperation::Deposit Pause	182
OS-VEP-ADV-64	DLOB Match Quoter Reprices Oracle-Offset Makers Against a Zero Oracle	184
OS-VEP-ADV-65	Market-Scoped AmmImmediateFill Pause Bit Is Defined and Settable but Never Enforced	187
OS-VEP-ADV-66	Permissionless Perp Fee Sweep Under-Reserves Expiry-Price Claims During Settlement	189
OS-VEP-ADV-67	Market-Scoped Settle-PnL Pause Does Not Stop Expired-Position Settlement	193
OS-VEP-ADV-68	TrySettle Soft-Skips a Paused Market's PnL Settle but Still Sweeps Its Revenue Share	195
OS-VEP-ADV-69	Expired-Position Closeout Fee Bypasses the Perp Fee-Ledger Waterfall	198
OS-VEP-ADV-70	Revenue-Share Settlement Credits Any Subaccount of the Recipient Authority	201
OS-VEP-ADV-71	Perp Bankruptcy Resolution Requires Open Interest Even When No Loss Is Socialized	204
OS-VEP-ADV-72	Perp Bankruptcy Socialized Funding Omits Net Unsettled Seed	206

OS-VEP-ADV-73	Swap-Backed Spot Liquidation Fees Bypass The Margin-Shortage Cap	209
OS-VEP-ADV-74	Cross-Bankruptcy Resolver Ordering Allocates Limited Quote IF	212
OS-VEP-ADV-75	Protocol Sweep Can Unback Floored Pending IF Bankruptcy Tranche	214
OS-VEP-ADV-76	Equity Breaker Does Not Block Trigger Orders	217
OS-VEP-ADV-77	Delegate Floor Delta Can Move Admin Floor Without Funds	219
OS-VEP-ADV-78	Equity Breaker Does Not Block Generic Spot Swaps	221
OS-VEP-ADV-79	Transfer Perp Position Skips Recipient Equity Floor	223
OS-VEP-ADV-80	AMM-Only Limit Cap Uses A Different Reserve Basis Than Execution	225
OS-VEP-ADV-81	Fill-Triggered Funding Uses Pre-Fill Reserve For Divergence Gate	228
OS-VEP-ADV-82	AMM Override Zero-Fill Suppresses Same-Timestamp Maker TWAP	230
OS-VEP-ADV-83	AMM Scalar Projection Ignores Market Min Order Size K-Down Floor	233
OS-VEP-ADV-84	AMM Quoter Marks Affordability-Rejected Refreshes Fresh	235
OS-VEP-ADV-85	AMM JIT Match Fills Bypass The Per-Fill Reserve Throttle	238
OS-VEP-ADV-86	Bid/Ask TWAP Can Advance Funding In Settlement Markets	240
OS-VEP-ADV-87	Perp Funding Settlement Paths Bypass Funding Pause	242
OS-VEP-ADV-88	Market Funding Pause Does Not Stop Bid/Ask TWAP Mutation	244
OS-VEP-ADV-89	Funding Updates Accept Stale Or Insufficient Oracles	245
OS-VEP-ADV-90	Equity Breaker Does Not Block Liquidator Actions	248
OS-VEP-ADV-91	Oracle Validity Cache Ignores Market Context	250
OS-VEP-ADV-92	PnL Settlement Accepts Stale Or Insufficient Oracles	251
OS-VEP-ADV-93	Prelaunch Oracle Stale Price Reports Zero Delay	253
OS-VEP-ADV-94	Pyth Lazer Confidence Can Be Understated By Caller-Selected Properties	255
OS-VEP-ADV-95	Fee Sweep Can Starve Pending Revenue-Share Claims	257
OS-VEP-ADV-96	Fee Sweep Accepts Stale Margin Insufficient Oracles	260
OS-VEP-ADV-97	SettleRevPool Pause Does Not Stop Revenue-Share Sweep	262

OS-VEP-ADV-98	Signed-Message Taker Orders Bypass The Global Exchange Pause	264
OS-VEP-ADV-99	Delegate Can Shrink Authority Signed-Message Replay PDA	266
OS-VEP-ADV-100	Signed Messages Lack Deployment Domain Separation	269
OS-VEP-ADV-101	Trigger Order Bypasses Market Fill Pause	271
OS-VEP-ADV-102	Trigger Order Accepts Stale Or Uncertain Oracles	273
OS-VEP-ADV-103	Spot Withdraw Pre-Margin TWAP Normalizes Too-Volatile Oracle	276
OS-VEP-ADV-104	Builder Fees Can Be Dropped When The Escrow Row Is Missing, Cleared, Or Stripped By Modify	279
OS-VEP-ADV-105	Self-Approved Builder Fee Bypasses The Initial-Margin Withdrawal Gate	282
OS-VEP-ADV-106	Signed Message TP/SL Sidecars Survive Main Order Soft Skip	285
OS-VEP-ADV-107	Signed-Message IOC Limit Orders Can Be Accepted As Resting Orders	287
OS-VEP-ADV-108	Trigger Order After Expiry Creates Settleable Keeper Reward	289
OS-VEP-ADV-109	Expired Perp Transfer Live Oracle Splits Settlement PnL	291
OS-VEP-ADV-110	A Late Vault Reward After The Final User Exit Becomes Manager Shares	292
OS-VEP-ADV-111	Management-Fee Rebase Leaves Existing Protocol Shares In The Old Denomination	294
OS-VEP-ADV-112	Manager Fee Updates Skip Strategy-Vault Initialization Bounds	296
OS-VEP-ADV-113	Vault Fee Timelock Applies A Newly Raised Management Fee To The Pre-Activation Epoch	298
OS-VEP-ADV-114	Public Vault Rebase Can Strand A Protocol Withdrawal Request	300
OS-VEP-ADV-115	Tokenized Vault Redemption Compares Inconsistent Share Domains	302
OS-VEP-ADV-116	Vault Share Transfer Skips Matured Fee Updates	304
OS-VEP-ADV-117	Protocol Vault Fee Accrual Can Underflow And Freeze Withdrawals	306
OS-VEP-ADV-118	Vault Deposit Can Mint Zero Shares	308
OS-VEP-ADV-119	Vault Profit Share Can Round To Zero Shares	310

OS-VEP-ADV-120	Tokenized Vault Redemption Bricks Future Tokenization	312
OS-VEP-ADV-121	Public Vault Rebase Can Zero Depositor Withdraw Claim	314
OS-VEP-ADV-122	Fee Accrual Rebase Can Freeze Vault Depositor Actions	317
OS-VEP-ADV-123	Full Insurance-Fund Unstake Cancel Can Burn The Staker's Shares	320
OS-VEP-ADV-124	Funding Pre-Gate TWAP Normalizes Too-Volatile Oracle	322
OS-VEP-ADV-125	Spot Swap Pre-Band TWAP Normalizes Price Checks	323
OS-VEP-ADV-126	Liquidation Pre-Band TWAP Normalizes Price Checks	324
OS-VEP-ADV-127	Perp Fill Pre-Band TWAP Normalizes Price Checks	325
OS-VEP-ADV-128	Interest Pause Catch-Up Credits Late Balances For The Entire Paused Interval	326
OS-VEP-ADV-129	Delisting Expiry Price Counts Unmoved Fee-Pool Value	327
OS-VEP-ADV-130	Equity Breaker Blocks Strictly Reducing Spot Swaps And Forces Liquidation Loss	328
OS-VEP-ADV-131	Spot Liabilities Are Absent From The Equity-Floor Metric	329
OS-VEP-ADV-132	Zero Spot-Oracle TWAP Timestamp Collapses The First Price Band	331
OS-VEP-ADV-133	Signerless Tokenized-Vault Rebase Erases Live SPL Backing	332
OS-VEP-ADV-134	Projected Routing Omits Refresh Accounting And Prioritizes A Worse Maker	333
OS-VEP-ADV-135	Same-Slot Oracle Freshness Reuses The Prior Sample's Validity	336
OS-VEP-ADV-136	Expiry Price Solves Against A Cost Basis That Omits Net Unsettled Funding	337
OS-VEP-ADV-137	Cancel-Withdraw Guard Precedes The Sole-Owner Escape Hatch And Forces A Stale-Value Exit	340
OS-VEP-ADV-138	Lending-Yield Carveouts Floor To Zero Under Frequent Permissionless Interest Cranking	342
OS-VEP-ADV-139	Deleting A Sub-Account Permanently Orphans Its Fee-Bearing Builder Rows	344
OS-VEP-ADV-140	Third-Party Escrow Initialization Permanently Suppresses Referrals	347

OS-VEP-ADV-141 Equity-Floor Gates Count Margin-Invalid Assets At Raw Oracle Value 348
OS-VEP-ADV-142 Same-Market Transfer Normalizes Its Liability TWAP Before Margin Validation 352
OS-VEP-ADV-143 Token-Unit Share Transfer Inflates The Recipient's Profit Basis 354
OS-VEP-ADV-144 Invalid-Oracle Dust Suppresses The Authority-Wide Equity Breaker 357
OS-VEP-ADV-145 Pooled Tokenized Cost Basis Lets A Later Tokenizer Strip Existing Holders' Loss Shelter 359
OS-VEP-ADV-146 Unpayable Positive-PnL Dust Can Veto Cross-Market Bankruptcy Resolution 361
OS-VEP-ADV-147 Expiry Solver Spends Earned Revenue-Share Backing On Perp Winners 363
OS-VEP-ADV-148 Per-User Exception Enables Sybil Breaker Bypass 367
OS-VEP-ADV-149 Zero-Token Socialized Deposit Residues Veto Follow-On Bankruptcy 370
OS-VEP-ADV-150 Pyth Ownership Check Accepts Invalid Oracles 374
5. Suggestions 375
OS-VEP-SUG-00 Post-Expiry Cranks Influence The Finalization-Time Settlement TWAP 376
OS-VEP-SUG-01 Third-Party Zero-Slot Referral Escrow Suppresses Rewards 377
OS-VEP-SUG-02 Variable Slot Times Alter Accounting Calculations 379
A. Vulnerability Rating Scale 381
B. Procedure 382

1. Executive Summary

1.1. Overview

Velocity Exchange engaged OtterSec to conduct a security assessment of the `Velocity Protocol v2` program. This assessment was conducted between June 19th and July 20th, 2026. For information on our auditing methodology, refer to Appendix B.

1.2. Key Findings

We produced 154 findings throughout this audit engagement.

Among the most impactful results, we identified that perp-bankruptcy socialization leaves the AMM's cumulative-funding checkpoint stale, causing the residual loss to be counted both as an obligation for surviving traders and as phantom AMM revenue that may later be paid from the PnL pool (● OS-VEP-ADV-19). We also found that an unvalidated vault-denomination oracle can understate NAV and overmint shares to a new depositor, enabling them to withdraw more than they deposited at the expense of existing vault participants (● OS-VEP-ADV-24).

Additionally, unchecked spot-swap outputs can push a market above its daily deposit cap, globally blocking ordinary withdrawals and debt repayments until the condition is administratively corrected or otherwise unwound (● OS-VEP-ADV-26). A malicious reward donation can also exploit rounding during withdrawal cancellation to burn a victim's entire vault-share claim and transfer the resulting value to the remaining depositor (● OS-VEP-ADV-23). Furthermore, rewards swept after a withdrawal request remain excluded from the requester's capped payout despite all requested shares being burned, allowing a later vault entrant to capture value accrued for the withdrawing participant (● OS-VEP-ADV-22).

Finally, stale builder-fee sidecar rows can be reused across markets, charging an unauthorized fee to a later order while paying the builder from the wrong market's PnL pool (● OS-VEP-ADV-18).

2. Scope

`Velocity Protocol v2` is a Solana perpetuals and spot trading protocol.

The source code was delivered to us in a Git repository at `https://github.com/velocity-exchange/velocity-v1`. This audit was performed against commit `9c330b1`.

3. 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-VEP-ADV-*` for advisories and `OS-VEP-SUG-*` for suggestions.

3.1. Findings Summary

Severity	Count
Critical	0
High	41
Medium	109
Low	1
Informational	3

4. Advisories

Here, we present a technical analysis of the 151 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-VEP-ADV-00	High	Resolved	Native Fast-Path Trusts Caller-Supplied State Bytes for Hot-Admin Authorization
OS-VEP-ADV-01	High	Resolved	Bulk Order placement Can Drop the Earlier Risk-Increasing Margin Check
OS-VEP-ADV-02	High	Resolved	AMM JIT Can Fill Inside DLOB Matches After AMM Fill Gates Say Unavailable
OS-VEP-ADV-03	High	Resolved	Bid/Ask TWAP Crank Lets Caller-Curated DLOB Depth Set Funding Price
OS-VEP-ADV-04	High	Resolved	Update Perp Bid/Ask TWAP Can Borrow Another User's Keeper Stake Gate
OS-VEP-ADV-05	High	Resolved	Direct Liquidation Accepts a Liability Oracle That Is Stale for Margin
OS-VEP-ADV-06	High	Resolved	Spot Bankruptcy Socializes Loss While Same-Market Revenue Stays Time-Locked
OS-VEP-ADV-07	High	Resolved	Pending IF Fee Sweep Can Force Bankruptcy Socialization
OS-VEP-ADV-08	High	Resolved	Direct Spot Liquidation Accepts a Margin-Invalid Deposit Oracle
OS-VEP-ADV-09	High	Resolved	Positive Unsettled PnL Blocks Liquidation of Larger Riskier Negative PnL
OS-VEP-ADV-10	High	Resolved	Dust Deposit Bricks Spot-Bankruptcy Resolution When Loss Exceeds Deposits
OS-VEP-ADV-11	High	Resolved	IF Unstake-Cancel Penalizes Canceling Staker for Vault Donations
OS-VEP-ADV-12	High	Resolved	IF Request-Remove Freezes Exit Value Before Settling Already-Due Revenue
OS-VEP-ADV-13	High	Resolved	IF Add Accepts a Positive Deposit That Mints Zero Shares
OS-VEP-ADV-14	High	Resolved	Revenue-Settle APR Cap Derived From Live IF-Vault Balance Is Donation-Inflatable
OS-VEP-ADV-15	High	Resolved	Revenue-Share Sweep Drains PnL Pool Without Reserving Live `net_user_pnl`

ID	Severity	Status	Description
OS-VEP-ADV-16	High	Resolved	Stale Builder Revenue-Share Row Overcharges a Later Non-Builder Order With the Same Order ID
OS-VEP-ADV-17	High	Resolved	Pyth Lazer Replay Refreshes a Stale Price as Slot-Fresh (No Max-Age Check)
OS-VEP-ADV-18	High	Resolved	Cross-Market Builder Sidecar Can Sweep The Wrong PnL Pool
OS-VEP-ADV-19	High	Resolved	Perp Bankruptcy Socialization Double-Credits The AMM Via A Stale last_cumulative_funding_rate
OS-VEP-ADV-20	High	Resolved	AMM Summary Counts Deferred Revenue Share As Spendable Fee Equity
OS-VEP-ADV-21	High	Resolved	Unswept Referral Rewards Let New Depositors Dilute Existing Vault Shareholders
OS-VEP-ADV-22	High	Resolved	Late Vault Entrant Can Capture A Pending Withdrawer's Accrued Reward
OS-VEP-ADV-23	High	Resolved	Builder-Reward Donation Can Burn A Cancelling Victim's Vault Claim
OS-VEP-ADV-24	High	Resolved	Unvalidated Vault Denomination Oracle Can Over-mint Depositor Shares
OS-VEP-ADV-25	High	Resolved	Zero Utilization Backdates Interest Onto The First Later Borrower
OS-VEP-ADV-26	High	Resolved	Unchecked Swap Outputs Turn The Deposit Cap Into A Global Exit And Repayment Lock
OS-VEP-ADV-27	High	Resolved	Post-Bankruptcy Revenue-Share Credit Escapes Setoff And Overdraws Insurance
OS-VEP-ADV-28	High	Resolved	Strictly Reducing Swap Trusts An Invalid Oracle In Its Loss Bound
OS-VEP-ADV-29	High	Resolved	Negative Expiry Price Is Clipped Out Of Margin And Equity
OS-VEP-ADV-30	High	Resolved	Withdraw Margin Uses Unaccrued Debt From Read-Only Spot Markets
OS-VEP-ADV-31	High	Resolved	Vault Deposit Mints Shares Before Target-Market Interest Refresh
OS-VEP-ADV-32	High	Resolved	Vault Withdrawal Requests And Cancellations Use Stale Interest Epochs
OS-VEP-ADV-33	High	Resolved	IF Cancel Before Revenue Settlement Evades Pending-Window Share Forfeiture
OS-VEP-ADV-34	High	Resolved	Reducing DLOB Match Crystallizes A Too-Uncertain Oracle Loss Below The Equity Floor

ID	Severity	Status	Description
OS-VEP-ADV-35	High	✓ Resolved	Standard Fill Margin Credits A Stale Deposit Into DLOB Bad Debt
OS-VEP-ADV-36	High	✓ Resolved	Standard Fill Underprices A Stale Liability Into DLOB Bad Debt
OS-VEP-ADV-37	High	✓ Resolved	Bid/Ask TWAP Crank Sets A Third Party's Forced-Close Auction Band
OS-VEP-ADV-38	High	✓ Resolved	Standard Fill Uses Unaccrued Spot Debt To Create Bad Debt
OS-VEP-ADV-39	High	✓ Resolved	Oracle Spread Check Uses Non-Strict Prices
OS-VEP-ADV-40	High	✓ Resolved	Pyth Ownership Check Accepts Invalid Oracles
OS-VEP-ADV-41	Medium	✓ Resolved	Direct Deposit Ignores the Market-Scoped Deposit Pause Bit
OS-VEP-ADV-42	Medium	✓ Resolved	Same-Market transfer-deposit Bypasses Recipient Deposit Admission
OS-VEP-ADV-43	Medium	✓ Resolved	`liquidate_spot_with_swap` Bricked by Stale Account-Index Guard
OS-VEP-ADV-44	Medium	✓ Resolved	Reduce-Only Maker Order Can Overfill When AMM Fulfillment Is Paused
OS-VEP-ADV-45	Medium	✓ Resolved	ReduceOnly Markets Can Still Fill Legacy Non-Reduce-Only Perp Orders
OS-VEP-ADV-46	Medium	✓ Resolved	Unbounded Order Expiry Overflows AMM Fallback Pricing And Aborts Fills
OS-VEP-ADV-47	Medium	✓ Resolved	Trigger-Order Path Uses The Exchange-Pause Guard Instead Of The Fill-Pause Guard
OS-VEP-ADV-48	Medium	✓ Resolved	Swap-Backed Spot Liquidation Bypasses The Max-Pct Liquidation Throttle
OS-VEP-ADV-49	Medium	✓ Resolved	PnL-vs-Spot Liquidations Skip the 5-Minute Spot Oracle TWAP Band
OS-VEP-ADV-50	Medium	✓ Resolved	Spot Bankruptcy Clears the Borrow Using Stale Cumulative Borrow Interest
OS-VEP-ADV-51	Medium	✓ Resolved	Perp PnL Deficit Recap Draws From the Insurance Fund on a Stale PnL Pool
OS-VEP-ADV-52	Medium	✓ Resolved	Transfer Pools Bypasses Destination Market Deposit Caps And Status Gate
OS-VEP-ADV-53	Medium	✓ Resolved	Funding Pause Does Not Stop Spot Interest Accrual Through Public Paths
OS-VEP-ADV-54	Medium	✓ Resolved	PnL-vs-Spot Liquidation Can Worsen Account Health When Liquidator Fee Exceeds Buffer

ID	Severity	Status	Description
OS-VEP-ADV-55	Medium	Resolved	`liquidate_perp_with_fill` Budgets IF/Protocol Fees on a Stale Fee Basis
OS-VEP-ADV-56	Medium	Resolved	`liquidate_perp_with_fill` Executes a Real Fill During Global FillPaused
OS-VEP-ADV-57	Medium	Resolved	Swap-Backed Spot Liquidation Bypasses Spot Deposit/Withdraw Pauses
OS-VEP-ADV-58	Medium	Ack'd	Permissionless `set_user_status_to_being_liquidated` Ignores Market-Local Liquidation Pauses
OS-VEP-ADV-59	Medium	Resolved	IF Stake Rebase Floors a Pending Unstake Request to Zero and Strands the Stake
OS-VEP-ADV-60	Medium	Resolved	IF-Add Auto-Settles Revenue While Global WithdrawPaused Blocks the Direct Settle Path
OS-VEP-ADV-61	Medium	Resolved	Revenue-Pool Deposit Credits at Stale Cumulative Deposit Interest
OS-VEP-ADV-62	Medium	Resolved	Direct Revenue-to-IF Settlement Ignores the Market-Scoped SpotOperation::Withdraw Pause
OS-VEP-ADV-63	Medium	Resolved	Revenue-Pool Deposit Bypasses the Market-Scoped SpotOperation::Deposit Pause
OS-VEP-ADV-64	Medium	Resolved	DLOB Match Quoter Reprices Oracle-Offset Makers Against a Zero Oracle
OS-VEP-ADV-65	Medium	Resolved	Market-Scoped AmmImmediateFill Pause Bit Is Defined and Settable but Never Enforced
OS-VEP-ADV-66	Medium	Resolved	Permissionless Perp Fee Sweep Under-Reserves Expiry-Price Claims During Settlement
OS-VEP-ADV-67	Medium	Resolved	Market-Scoped Settle-PnL Pause Does Not Stop Expired-Position Settlement
OS-VEP-ADV-68	Medium	Resolved	TrySettle Soft-Skips a Paused Market's PnL Settle but Still Sweeps Its Revenue Share
OS-VEP-ADV-69	Medium	Resolved	Expired-Position Closeout Fee Bypasses the Perp Fee-Ledger Waterfall
OS-VEP-ADV-70	Medium	Resolved	Revenue-Share Settlement Credits Any Subaccount of the Recipient Authority
OS-VEP-ADV-71	Medium	Resolved	Perp Bankruptcy Resolution Requires Open Interest Even When No Loss Is Socialized
OS-VEP-ADV-72	Medium	Resolved	Perp Bankruptcy Socialized Funding Omits Net Unsettled Seed
OS-VEP-ADV-73	Medium	Resolved	Swap-Backed Spot Liquidation Fees Bypass The Margin-Shortage Cap

ID	Severity	Status	Description
OS-VEP-ADV-74	Medium	Resolved	Cross-Bankruptcy Resolver Ordering Allocates Limited Quote IF
OS-VEP-ADV-75	Medium	Resolved	Protocol Sweep Can Unback Floored Pending IF Bankruptcy Tranche
OS-VEP-ADV-76	Medium	Resolved	Equity Breaker Does Not Block Trigger Orders
OS-VEP-ADV-77	Medium	Resolved	Delegate Floor Delta Can Move Admin Floor Without Funds
OS-VEP-ADV-78	Medium	Resolved	Equity Breaker Does Not Block Generic Spot Swaps
OS-VEP-ADV-79	Medium	Resolved	Transfer Perp Position Skips Recipient Equity Floor
OS-VEP-ADV-80	Medium	Resolved	AMM-Only Limit Cap Uses A Different Reserve Basis Than Execution
OS-VEP-ADV-81	Medium	Resolved	Fill-Triggered Funding Uses Pre-Fill Reserve For Divergence Gate
OS-VEP-ADV-82	Medium	Resolved	AMM Override Zero-Fill Suppresses Same-Timestamp Maker TWAP
OS-VEP-ADV-83	Medium	Resolved	AMM Scalar Projection Ignores Market Min Order Size K-Down Floor
OS-VEP-ADV-84	Medium	Resolved	AMM Quoter Marks Affordability-Rejected Refreshes Fresh
OS-VEP-ADV-85	Medium	Resolved	AMM JIT Match Fills Bypass The Per-Fill Reserve Throttle
OS-VEP-ADV-86	Medium	Resolved	Bid/Ask TWAP Can Advance Funding In Settlement Markets
OS-VEP-ADV-87	Medium	Resolved	Perp Funding Settlement Paths Bypass Funding Pause
OS-VEP-ADV-88	Medium	Resolved	Market Funding Pause Does Not Stop Bid/Ask TWAP Mutation
OS-VEP-ADV-89	Medium	Ack'd	Funding Updates Accept Stale Or Insufficient Oracles
OS-VEP-ADV-90	Medium	Resolved	Equity Breaker Does Not Block Liquidator Actions
OS-VEP-ADV-91	Medium	Resolved	Oracle Validity Cache Ignores Market Context
OS-VEP-ADV-92	Medium	Resolved	PnL Settlement Accepts Stale Or Insufficient Oracles
OS-VEP-ADV-93	Medium	Resolved	Prelaunch Oracle Stale Price Reports Zero Delay
OS-VEP-ADV-94	Medium	Resolved	Pyth Lazer Confidence Can Be Understated By Caller-Selected Properties
OS-VEP-ADV-95	Medium	Resolved	Fee Sweep Can Starve Pending Revenue-Share Claims
OS-VEP-ADV-96	Medium	Ack'd	Fee Sweep Accepts Stale Margin Insufficient Oracles

ID	Severity	Status	Description
OS-VEP-ADV-97	Medium	Resolved	SettleRevPool Pause Does Not Stop Revenue-Share Sweep
OS-VEP-ADV-98	Medium	Resolved	Signed-Message Taker Orders Bypass The Global Exchange Pause
OS-VEP-ADV-99	Medium	Resolved	Delegate Can Shrink Authority Signed-Message Replay PDA
OS-VEP-ADV-100	Medium	Ack'd	Signed Messages Lack Deployment Domain Separation
OS-VEP-ADV-101	Medium	Resolved	Trigger Order Bypasses Market Fill Pause
OS-VEP-ADV-102	Medium	Resolved	Trigger Order Accepts Stale Or Uncertain Oracles
OS-VEP-ADV-103	Medium	Resolved	Spot Withdraw Pre-Margin TWAP Normalizes Too-Volatile Oracle
OS-VEP-ADV-104	Medium	Resolved	Builder Fees Can Be Dropped When The Escrow Row Is Missing, Cleared, Or Stripped By Modify
OS-VEP-ADV-105	Medium	Resolved	Self-Approved Builder Fee Bypasses The Initial-Margin Withdrawal Gate
OS-VEP-ADV-106	Medium	Resolved	Signed Message TP/SL Sidecars Survive Main Order Soft Skip
OS-VEP-ADV-107	Medium	Resolved	Signed-Message IOC Limit Orders Can Be Accepted As Resting Orders
OS-VEP-ADV-108	Medium	Resolved	Trigger Order After Expiry Creates Settleable Keeper Reward
OS-VEP-ADV-109	Medium	Resolved	Expired Perp Transfer Live Oracle Splits Settlement PnL
OS-VEP-ADV-110	Medium	Resolved	A Late Vault Reward After The Final User Exit Becomes Manager Shares
OS-VEP-ADV-111	Medium	Resolved	Management-Fee Rebase Leaves Existing Protocol Shares In The Old Denomination
OS-VEP-ADV-112	Medium	Resolved	Manager Fee Updates Skip Strategy-Vault Initialization Bounds
OS-VEP-ADV-113	Medium	Resolved	Vault Fee Timelock Applies A Newly Raised Management Fee To The Pre-Activation Epoch
OS-VEP-ADV-114	Medium	Resolved	Public Vault Rebase Can Strand A Protocol Withdrawal Request
OS-VEP-ADV-115	Medium	Resolved	Tokenized Vault Redemption Compares Inconsistent Share Domains
OS-VEP-ADV-116	Medium	Resolved	Vault Share Transfer Skips Matured Fee Updates

ID	Severity	Status	Description
OS-VEP-ADV-117	Medium	Resolved	Protocol Vault Fee Accrual Can Underflow And Freeze Withdrawals
OS-VEP-ADV-118	Medium	Resolved	Vault Deposit Can Mint Zero Shares
OS-VEP-ADV-119	Medium	Resolved	Vault Profit Share Can Round To Zero Shares
OS-VEP-ADV-120	Medium	Resolved	Tokenized Vault Redemption Bricks Future Tokenization
OS-VEP-ADV-121	Medium	Resolved	Public Vault Rebase Can Zero Depositor Withdraw Claim
OS-VEP-ADV-122	Medium	Resolved	Fee Accrual Rebase Can Freeze Vault Depositor Actions
OS-VEP-ADV-123	Medium	Resolved	Full Insurance-Fund Unstake Cancel Can Burn The Staker's Shares
OS-VEP-ADV-124	Medium	Resolved	Funding Pre-Gate TWAP Normalizes Too-Volatile Oracle
OS-VEP-ADV-125	Medium	Resolved	Spot Swap Pre-Band TWAP Normalizes Price Checks
OS-VEP-ADV-126	Medium	Resolved	Liquidation Pre-Band TWAP Normalizes Price Checks
OS-VEP-ADV-127	Medium	Resolved	Perp Fill Pre-Band TWAP Normalizes Price Checks
OS-VEP-ADV-128	Medium	Resolved	Interest Pause Catch-Up Credits Late Balances For The Entire Paused Interval
OS-VEP-ADV-129	Medium	Resolved	Delisting Expiry Price Counts Unmoved Fee-Pool Value
OS-VEP-ADV-130	Medium	Resolved	Equity Breaker Blocks Strictly Reducing Spot Swaps And Forces Liquidation Loss
OS-VEP-ADV-131	Medium	Resolved	Spot Liabilities Are Absent From The Equity-Floor Metric
OS-VEP-ADV-132	Medium	Resolved	Zero Spot-Oracle TWAP Timestamp Collapses The First Price Band
OS-VEP-ADV-133	Medium	Resolved	Signerless Tokenized-Vault Rebase Erases Live SPL Backing
OS-VEP-ADV-134	Medium	Resolved	Projected Routing Omits Refresh Accounting And Prioritizes A Worse Maker
OS-VEP-ADV-135	Medium	Resolved	Same-Slot Oracle Freshness Reuses The Prior Sample's Validity
OS-VEP-ADV-136	Medium	Resolved	Expiry Price Solves Against A Cost Basis That Omits Net Unsettled Funding
OS-VEP-ADV-137	Medium	Resolved	Cancel-Withdraw Guard Precedes The Sole-Owner Escape Hatch And Forces A Stale-Value Exit

ID	Severity	Status	Description
OS-VEP-ADV-138	Medium	Resolved	Lending-Yield Carveouts Floor To Zero Under Frequent Permissionless Interest Cranking
OS-VEP-ADV-139	Medium	Resolved	Deleting A Sub-Account Permanently Orphans Its Fee-Bearing Builder Rows
OS-VEP-ADV-140	Medium	Resolved	Third-Party Escrow Initialization Permanently Suppresses Referrals
OS-VEP-ADV-141	Medium	Resolved	Equity-Floor Gates Count Margin-Invalid Assets At Raw Oracle Value
OS-VEP-ADV-142	Medium	Resolved	Same-Market Transfer Normalizes Its Liability TWAP Before Margin Validation
OS-VEP-ADV-143	Medium	Resolved	Token-Unit Share Transfer Inflates The Recipient's Profit Basis
OS-VEP-ADV-144	Medium	Resolved	Invalid-Oracle Dust Suppresses The Authority-Wide Equity Breaker
OS-VEP-ADV-145	Medium	Resolved	Pooled Tokenized Cost Basis Lets A Later Tokenizer Strip Existing Holders' Loss Shelter
OS-VEP-ADV-146	Medium	Resolved	Unpayable Positive-PnL Dust Can Veto Cross-Market Bankruptcy Resolution
OS-VEP-ADV-147	Medium	Resolved	Expiry Solver Spends Earned Revenue-Share Backing On Perp Winners
OS-VEP-ADV-148	Medium	Resolved	Per-User Exception Enables Sybil Breaker Bypass
OS-VEP-ADV-149	Medium	Resolved	Zero-Token Socialized Deposit Residues Veto Follow-On Bankruptcy
OS-VEP-ADV-150	Low	Resolved	Pyth Ownership Check Accepts Invalid Oracles

Native Fast-Path Trusts Caller-Supplied State Bytes for Hot-Admin Authorization

ID	Severity	Status
OS-VEP-ADV-00	High	Resolved

Description

The program entrypoint contains a native fast path for instructions prefixed with `0xFFFFFFFF`, and this dispatch happens before the normal Anchor entrypoint is invoked. As shown in `program_entry`, discriminator `0` routes to the MM-oracle native handler and discriminator `1` routes to the AMM-spread-adjustment native handler, so these calls do not receive Anchor account validation before reaching the native code.

programs/velocity/src/lib.rs

```rust
pub fn program_entry<'info>(
    program_id: &'info Pubkey,
    accounts: &'info [AccountInfo<'info>],
    data: &'info [u8],
) -> anchor_lang::solana_program::entrypoint::ProgramResult {
    if let [0xFF, 0xFF, 0xFF, 0xFF, discriminator, ref payload @ ..] = data {
        match *discriminator {
            0 => Ok(handle_update_mm_oracle_native(accounts, payload)?),
            1 => Ok(handle_update_amm_spread_adjustment_native(
                accounts, payload,
            )?),
            _ => Err([...]),
        }
    } else {
        // Fallback to anchor generated entry
        entry(program_id, accounts, data)
    }
}
```

Native pre-Anchor dispatch for hot-admin opcodes

The AMM-spread-adjustment native handler authenticates the signer by reading `hot_amm_spread_adjust` from fixed bytes in the caller-supplied state account, then directly casts the first account's data as a `PerpMarket` and writes `amm.amm_spread_adjustment`. As shown in `handle_update_amm_spread_adjustment_native`, the handler does not verify that the state account is the canonical Velocity state PDA, that the state or market accounts are owned by the Velocity program, or that either account has the expected discriminator before using those raw bytes.

programs/velocity/src/vlp/amm/admin.rs

```rust
pub fn handle_update_amm_spread_adjustment_native(
    accounts: &[AccountInfo],
    data: &[u8],
) -> Result<()> {
    // Accounts: [0] perp_market (mut), [1] signer, [2] state.
    // hot_amm_spread_adjust lives at bytes 392..424 of State (after disc).
    let signer_account = &accounts[1];
    #[cfg(not(feature = "anchor-test"))]
    {
        let state = &accounts[2].data.borrow();
        let mut hot_amm_spread_adjust = [0u8; 32];
        hot_amm_spread_adjust.copy_from_slice(&state[392..424]);
        let hot_key =
        anchor_lang::prelude::Pubkey::new_from_array(hot_amm_spread_adjust);
        assert!(
            signer_account.is_signer && *signer_account.key == hot_key,
            "signer must match state.hot_amm_spread_adjust, signer: {}, expected: {}",
            signer_account.key,
            hot_key
        );
    }
    let mut perp_market_data = accounts[0].data.borrow_mut();
    let perp_market: &mut PerpMarket =
        bytemuck::from_bytes_mut(&mut perp_market_data[8..8 +
        std::mem::size_of::<PerpMarket>()]);
    perp_market.amm.amm_spread_adjustment = data[0] as i8;

    Ok(())
}
```

Caller-supplied state bytes authorize direct PerpMarket mutation

Because the trusted hot-admin key is derived from bytes supplied through the account list, a public caller can make the signer comparison succeed by providing state-shaped bytes containing the caller's signing key at the checked offset. The same native dispatch also exposes the MM-oracle update path described in the finding, whose sibling handler reads hot-admin and feature-flag values from fixed offsets, reads the current slot from the supplied clock account bytes, and writes `mm_oracle_price`, `mm_oracle_slot`, and `mm_oracle_sequence_id` after sequence, slot-gap, and step-cap checks.

This grants the native hot-admin writes to callers that can supply matching account data rather than to the intended configured hot-admin role. For the spread-adjustment opcode, attacker-controlled negative values may reduce both AMM spreads down to the code-enforced floor of `1`, while positive values may widen spreads and reduce AMM availability. For the MM-oracle opcode,

accepted updates feed the market's MM-oracle price data, which is used by AMM refresh, order fill, funding, and liquidation-adjacent oracle logic in the linked repository context.

Remediation

Before reading the fixed offsets, require the supplied state account to be the canonical `velocity_state` PDA for crate::ID, owned by crate::ID, and ≥ State::SIZE, and verify its discriminator; require `accounts[0]` to be a Velocity-owned PerpMarket with the expected discriminator, and require the MM-oracle clock account to be the real clock sysvar. Prefer `AccountLoader<State>::try_from` or a shared native-auth helper so future native handlers can't skip these checks.

Patch

Resolved in PR#141.

Bulk Order placement Can Drop the Earlier Risk-Increasing Margin Check

ID	Severity	Status
OS-VEP-ADV-01	High	Resolved

Description

The bulk `place_orders` entrypoint accepts a batch of up to 32 order parameters and processes them sequentially. For each perp order, it constructs a new `PlaceOrderOptions` value with `risk_increasing` initialized to false and `enforce_margin_check` set only when the loop is processing the final order.

Inside `place_perp_order`, one allowed no-op path occurs before a new order is placed or any placement margin check can run. If the computed expiration timestamp is already in the past, the function logs the skipped order and returns successfully, as shown in the early timestamp check.

programs/velocity/src/controller/orders.rs

```rust
pub fn place_perp_order([...]) -> VelocityResult {
    [...]
    let max_ts = match params.max_ts {
        Some(max_ts) => max_ts,
        None => match params.order_type {
            OrderType::Market | OrderType::Oracle => now.safe_add(
                30_i64.max(
                    (auction_duration.safe_div(2)?)
                        .cast::<i64>()?
                        .safe_add(10_i64)?,
                ),
            )?,
            _ => 0_i64,
        },
    };
    if max_ts != 0 && max_ts < now {
        msg!("max_ts ({}) < now ({}), skipping order", max_ts, now);
        return Ok(());
    }
    [...]
}
```

Expired orders can return successfully before the placement margin check.

`place_perp_order` also returns successfully for a `TryPostOnly` post-only failure before reaching the margin check, and otherwise performs the placement margin check only when `options.enforce_margin_check` is true.

Because the batch caller creates fresh options for every iteration, a risk-increasing result from an earlier order is not accumulated into the final order's margin check. If the final order is

not risk-increasing, the final check is called with `risk_increasing` false; if the final order is an allowed no-op such as an expired order or `TryPostOnly` failure, the successful early return can skip the post-batch placement margin check entirely. The distinction is material because `meets_place_order_margin_requirement` uses initial margin only when its `risk_increasing` argument is true, and otherwise uses maintenance margin.

This can admit resting risk-increasing perp orders through the bulk placement path under a weaker threshold than the single-order placement path would require. If such an order later matches, the fill path still performs a margin check, but the non-liquidation fill logic uses `MarginRequirementType::Fill` for non-position-decreasing taker orders and maintenance margin for position-decreasing orders, rather than the initial margin requirement used for risk-increasing placement.

Remediation

Accumulate batch-level risk across all orders: create one mutable `PlaceOrderOptions` outside the loop (or a separate `batch_risk_increasing` boolean), OR each `place_perp_order` result into it, and enforce the final margin check with the accumulated value after all orders are processed. The final post-batch check must live outside `place_perp_order` (or otherwise run even when the final order is an allowed no-op).

Patch

Resolved in PR#135.

AMM JIT Can Fill Inside DLOB Matches After AMM Fill Gates Say Unavailable

ID	Severity	Status
OS-VEP-ADV-02	High	✓ Resolved

Description

`fill_perp_order` derives a single `amm_is_available` decision before routing a perpetual fill. That decision starts from the global AMM pause state and is then combined with the market-level `amm_can_fill_order` result, which is the control point for AMM fill eligibility in this path.

`determine_perp_fulfillment_methods` receives `amm_is_available`, but applies it only when adding standalone AMM fulfillment methods. Resting DLOB maker matches are still pushed into the fulfillment list when the taker crosses the maker, even if the earlier AMM availability decision was false.

programs/velocity/src/math/fulfillment.rs

```
pub fn determine_perp_fulfillment_methods([...]) ->
VelocityResult<Vec<PerpFulfillmentMethod>> {
    [...]
    for (maker_key, maker_order_index, maker_price) in maker_orders_info.iter() {
        let taker_crosses_maker = match limit_price {
            Some(taker_price) => do_orders_cross(maker_direction, *maker_price,
            taker_price),
            None => true,
        };
        if !taker_crosses_maker {
            break;
        }

        if amm_is_available {
            let maker_better_than_amm = match order.direction {
                PositionDirection::Long => *maker_price <= amm_price,
                PositionDirection::Short => *maker_price >= amm_price,
            };

            if !maker_better_than_amm {
                fulfillment_methods.push(PerpFulfillmentMethod::AMM(Some(*maker_price)))
                amm_price = *maker_price;
            }
        }

        fulfillment_methods.push(PerpFulfillmentMethod::Match(
            *maker_key,
            maker_order_index.cast()?,
            *maker_price,
        ));
```

```
29
30          if fulfillment_methods.len() > 6 {
31              break;
32          }
33      }
34      if amm_is_available { [...] }
35      [...]
36  }
```

determine_perp_fulfillment_methods gates standalone AMM methods but still adds DLOB matches.

When a `Match` fulfillment method is executed, `fulfill_perp_order_step` does not receive or re-check the `amm_is_available` flag. Instead, the match branch constructs an `AmmJitQuoter` from the match context and includes it alongside the DLOB order quoter, allowing the AMM to participate as a JIT maker during the same DLOB match.

`AmmJitQuoter::commit_fill` applies AMM swap math and writes the resulting reserve and `base_asset_amount_with_amm` changes back to the AMM. As a result, a JIT slice produced inside a DLOB match is not only quoted; it can mutate AMM state.

programs/velocity/src/vlp/amm/quoter.rs

```
996  impl<'a> QuoterCommit for AmmJitQuoter<'a> {
997      fn commit_fill(&mut self, _ctx: &QuoteContext, fill: &QuoterFill) ->
         VelocityResult<()> {
998          if fill.base_filled == 0 {
999              return Ok(());
1000         }
1001         let direction = Self::swap_direction(fill.side);
1002         let swap =
1003             amm_controller::calculate_base_swap_output(self.amm, fill.base_filled,
                direction)?;
1004         self.amm.base_asset_reserve = swap.new_base_asset_reserve;
1005         self.amm.quote_asset_reserve = swap.new_quote_asset_reserve;
1006
1007         // Same as AmmQuoter: with_amm tracks USERS' net position with AMM as
1008         // counterparty. Taker Long → users' net long grows → += base.
1009         let signed_base = fill.base_filled as i128;
1010         let delta = match fill.side {
1011             PositionDirection::Long => signed_base,
1012             PositionDirection::Short => -signed_base,
1013         };
1014         self.amm.base_asset_amount_with_amm =
1015             self.amm.base_asset_amount_with_amm.safe_add(delta)?;
1016
1017         // The fill moved the curve reserves; re-derive the cached ask/bid
1018         // spread reserves off the (unchanged) cached spreads so the cache
```

```
1019            // dashboards read stays consistent — mirrors master's post-swap
1020            // `update_spread_reserves`.
1021            crate::vlp::amm::math::spread::refresh_cached_spread_reserves(self.amm)?;
1022            Ok(())
1023        }
1024    }
```

`AmmJitQuoter::commit_fill` mutates AMM reserves.

This bypasses the intended effect of AMM fill gates for matched DLOB fills. A taker or filler that crosses a resting maker can still cause the vAMM to trade as a JIT maker when standalone AMM fills have been made unavailable by the global AMM pause, a per-market AMM fill pause, drawdown retreat, or the other checks folded into `amm_can_fill_order`.

Remediation

Carry the AMM availability decision into the `Match` branch and set `max_jit_base=0` / omit `AmmJitQuoter` when `amm_is_available` is false, or enforce the same pause/drawdown/volatility/oracle checks in `AmmJitQuoter` construction.

Patch

Resolved in PR#182.

Bid/Ask TWAP Crank Lets Caller-Curated DLOB Depth Set Funding Price

ID	Severity	Status
OS-VEP-ADV-03	High	Resolved

Description

The `update_perp_bid_ask_twap` path derives mark TWAP inputs from DLOB maker liquidity supplied through the instruction's remaining accounts. Before the handler estimates bid and ask prices from that depth, `filter_bids_asks_by_oracle_divergence` applies only a lower bound to bids and an upper bound to asks: bids that are too low are removed and asks that are too high are removed, but high bids and low asks are not symmetrically rejected, as shown in `filter_bids_asks_by_oracle_divergence`.

programs/velocity/src/math/orders.rs

```rust
pub fn filter_bids_asks_by_oracle_divergence(
    bids: Vec<Level>,
    asks: Vec<Level>,
    oracle_price: i64,
    max_divergence_percent: u64,
) -> VelocityResult<(Vec<Level>, Vec<Level>)> {
    let oracle_abs = oracle_price.unsigned_abs().max(1);
    let min_bid_price: u64 = oracle_abs
        .cast::<u128>()?
        .safe_mul((100 - max_divergence_percent).min(100).cast()?)?
        .safe_div(100)?
        .cast()?;
    let max_ask_price: u64 = oracle_abs
        .cast::<u128>()?
        .safe_mul((100 + max_divergence_percent).cast()?)?
        .safe_div(100)?
        .cast()?;

    let filtered_bids: Vec<Level> = bids
        .into_iter()
        .filter(|level| level.price >= min_bid_price)
        .collect();
    let filtered_asks: Vec<Level> = asks
        .into_iter()
        .filter(|level| level.price <= max_ask_price)
        .collect();

    Ok((filtered_bids, filtered_asks))
}
```

One-sided oracle divergence filtering

After loading the caller-provided maker accounts, the handler estimates bid and ask prices at the market's funding-rate depth, folds those estimates into `market_stats.update_mark_twap_crank`, and then invokes `controller::funding::update_funding_rate` in the same instruction. The highlighted call in `handle_update_perp_bid_ask_twap` shows that the funding writer runs immediately after the DLOB-based mark TWAP update, using the same market state and timestamp.

programs/velocity/src/instructions/keeper.rs

```
pub fn handle_update_perp_bid_ask_twap<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    let funding_paused =
        state.funding_paused()? ||
        perp_market.is_operation_paused(PerpOperation::UpdateFunding);
    controller::funding::update_funding_rate(
        perp_market.market_index,
        perp_market,
        &mut oracle_map,
        now,
        slot,
        &state.oracle_guard_rails,
        funding_paused,
        None,
    )?;
    [...]
}
```

DLOB TWAP crank immediately calls funding update

Inside `update_funding_rate`, the funding calculation refreshes the mark TWAP again through `update_mark_twap_with_amm_bid_ask` and derives the funding rate from the spread between that returned mark TWAP and the oracle TWAP.tThe earlier crank path writes `last_mark_price_twap_ts = now` because the funding call uses the same `now`, the shared TWAP helper has zero elapsed time and may preserve the DLOB-influenced mark TWAP as the funding input.

The same funding call then converts the resulting funding rate into long and short funding deltas and mutates `cumulative_funding_rate_long` and `cumulative_funding_rate_short`. The manipulated mark input can therefore affect persistent cumulative funding state and the emitted `FundingUpdated` event.

programs/velocity/src/controller/funding.rs

```
pub fn update_funding_rate([...]) -> VelocityResult<bool> {
    [...]
    // Apply cum-rate updates BEFORE dispatching FundingUpdated so the AMM
    // settles against post-update values (same as user positions settle
    // against current cum rates).
```

```
        market.cumulative_funding_rate_long = market
            .cumulative_funding_rate_long
            .safe_add(funding_rate_long_value)?;
        market.cumulative_funding_rate_short = market
            .cumulative_funding_rate_short
            .safe_add(funding_rate_short_value)?;

        // ---- AMM-side settlement: a fresh quoter settles the AMM's own funding
        // PnL from the post-update cum-rate deltas and runs its eager k-update,
        // emitting `AmmCurveChanged` itself when the curve moves. ----
        let event = MarketEvent::FundingUpdated {[...]};
        let event_ctx = QuoteContext {[...]};
        AmmQuoter::for_amm(&mut market.amm).on_market_event(&event_ctx, &event)?;
        [...]
    }
```

Funding updates cumulative long and short rates

A keeper satisfying the handler's keeper-stat checks can therefore choose the maker accounts used for the DLOB depth, place depth that passes the one-sided oracle filter, and crank at a funding boundary so the curated mark TWAP feeds the funding update. Thus, funding can be steered in favor of one side of the market, including the negative-funding direction, subject to the protocol's funding clamps and profitability checks.

Remediation

Decouple the funding update from the caller-curated DLOB TWAP update (or compute funding's mark input before any DLOB mutation), require a canonical maker set / remove DLOB depth from the public funding crank, apply symmetric oracle bands, and bind `keeper_stats.authority` to the signer.

Patch

Resolved in PR#416.

Update Perp Bid/Ask TWAP Can Borrow Another User's Keeper Stake Gate

ID	Severity	Status
OS-VEP-ADV-04	High	✓ Resolved

Description

`UpdatePerpBidAskTwap` accepts `keeper_stats` and `authority` as separate accounts. The account context requires `authority` to be a signer, but it does not bind the loaded `UserStats` account to that signer with a `has_one = authority` constraint as shown in `UpdatePerpBidAskTwap`.

programs/velocity/src/instructions/keeper.rs

```
#[derive(Accounts)]
pub struct UpdatePerpBidAskTwap<'info> {
    pub state: AccountLoader<'info, State>,
    #[account(mut)]
    pub perp_market: AccountLoader<'info, PerpMarket>,
    /// CHECK: checked in `update_funding_rate` ix constraint
    pub oracle: UncheckedAccount<'info>,
    pub keeper_stats: AccountLoader<'info, UserStats>,
    pub authority: Signer<'info>,
}
```

Unbound keeper stats and signer accounts

`handle_update_perp_bid_ask_twap` then loads the caller-supplied `keeper_stats` and uses it for the two keeper gates: `can_update_bid_ask_twap()` and the minimum IF stake amount. After those checks pass, the same handler continues into oracle-derived market stat refresh, bid and ask TWAP calculation, mark TWAP update, and funding-rate update, without comparing the `UserStats` authority to `ctx.accounts.authority`.

The `can_update_bid_ask_twap` helper only derives its result from the `UserStats` pause bit for `UpdateBidAskTwap`. It does not authenticate the signer or establish that the stats row belongs to the signer, so it cannot compensate for the missing binding in the instruction account context.

programs/velocity/src/state/user.rs

```
pub fn can_update_bid_ask_twap(&self) -> bool {
    let update_mark_twap_paused = UserStatsPausedOperations::is_operation_paused(
        self.paused_operations,
        UserStatsPausedOperations::UpdateBidAskTwap,
    );
    !update_mark_twap_paused
}
```

Pause helper does not authenticate the stats authority

Because the pause and stake checks are evaluated against whichever `UserStats` account is supplied, a public signer may pass an unpaused, sufficiently staked stats row owned by another user and reach the TWAP and funding mutation path without that owner signing. This makes the staked-keeper gate borrowable and weakens the economic prerequisite intended to restrict access to the funding-manipulation surface.

Remediation

Bind `keeper_stats` to the signer (`has_one = authority`, or a handler check `keeper_stats.load()?.authority == authority.key()`).

Patch

Resolved in PR#213.

Direct Liquidation Accepts a Liability Oracle That Is Stale for Margin

ID	Severity	Status
OS-VEP-ADV-05	High	Resolved

Description

The oracle validity policy distinguishes liquidation from margin calculation. In `is_oracle_valid_for_action`, `VelocityAction::MarginCalc` rejects oracle states that include `TooUncertain` and `StaleForMargin`, while `VelocityAction::Liquidate` only rejects `NonPositive` and `TooVolatile`. This means an oracle state that is invalid for margin can still be considered valid for liquidation, as shown by the action-specific validity checks.

The direct spot liquidation path uses that broader liquidation policy for both sides of the liquidation. In `liquidate_spot`, the asset market and the liability market are refreshed through `update_spot_market_and_check_validity` with `Some(VelocityAction::Liquidate)`, and the liability price returned from that refresh is then carried into the subsequent liquidation sizing flow.

By contrast, withdraw-style value release explicitly checks the margin calculation's liability-oracle aggregate before allowing the user to proceed while liabilities remain outstanding. In `meets_withdraw_margin_requirement`, a user with a positive margin requirement or outstanding liabilities must have `calculation.all_liability_oracles_valid`, otherwise the path returns `InvalidOracle`.

The later `liquidate_spot` transfer block applies the liquidation result after the sizing and price-band checks. It reduces the user's liability, credits insurance and protocol fees, assigns the liability transfer to the liquidator as a borrow, and transfers the user's asset deposit exposure to the liquidator. This is the point where the liquidation calculation becomes a balance movement.

As a result, a public liquidator may execute spot collateral and liability transfers using a liability oracle state that the protocol treats as invalid for ordinary margin-based value release. The code still enforces the five-minute TWAP divergence band before the transfer, but that check only limits recent price divergence and does not make `StaleForMargin` or `TooUncertain` valid under the stricter margin policy.

Remediation

Align the oracle validity requirements used for liability-side liquidation with the requirements used by margin calculation. In direct `liquidate_spot`, and in the analogous direct perp liquidation path, either reject liability oracles that `VelocityAction::MarginCalc` would reject or require the post-calculation `all_liability_oracles_valid` flag before applying liability, collateral, perp exposure, or liquidation-fee transfers.

Patch

Resolved in PR#391.

Spot Bankruptcy Socializes Loss While Same-Market Revenue Stays Time-Locked

ID	Severity	Status
OS-VEP-ADV-06	High	Resolved

Description

`resolve_spot_bankruptcy` resolves a bankrupt borrow by comparing the borrow amount only against the already-settled insurance fund vault balance. The function contains an explicit placeholder for a market insurance fund draw before social loss, but the value used for the insurance payment is derived from the vault balance passed by the caller rather than from the affected `SpotMarket.revenue_pool`.

programs/velocity/src/controller/liquidation.rs

```
pub fn resolve_spot_bankruptcy([...]) -> VelocityResult<u64> {
    [...]
    // todo: add market's insurance fund draw attempt here (before social loss)
    // subtract 1 so insurance_fund_vault_balance always stays >= 1
    let if_payment =
    borrow_amount.min(insurance_fund_vault_balance.saturating_sub(1).cast()?);
    let loss_to_socialize = borrow_amount.safe_sub(if_payment)?;
    [...]
}
```

Any unpaid portion is then treated as `loss_to_socialize` and applied through a reduction of `cumulative_deposit_interest`. In `resolve_spot_bankruptcy`, the market accounting records the remaining loss after the insurance fund payment, but there is no intervening draw from the same market's revenue pool before depositor claims are reduced.

The keeper handler does attempt to settle market revenue into the insurance fund vault before invoking `resolve_spot_bankruptcy`, then reloads the spot market and insurance fund vault balances and passes the insurance fund vault amount into the controller. This means the bankruptcy path can use revenue only after it has successfully moved into the vault through the ordinary settlement path.

That ordinary settlement path is time-gated by `on_the_hour_update` in `attempt_settle_revenue_to_insurance_fund`, and `settle_revenue_to_insurance_fund` further caps the amount settled when insurance fund stakers exist. As a result, market revenue that remains in `SpotMarket.revenue_pool` because the timer is not due or the cap binds is not used as first-loss coverage during spot bankruptcy.

programs/velocity/src/controller/insurance.rs

```rust
pub fn settle_revenue_to_insurance_fund([...]) -> VelocityResult<u64> {
    [...]
    if spot_market.insurance_fund.user_shares > 0 {
        // only allow MAX_APR_PER_REVENUE_SETTLE_TO_INSURANCE_FUND_VAULT or 1/10th of
        revenue pool to be settled
        let capped_apr_amount = insurance_vault_amount
            .cast::<u128>()?
            .safe_mul(MAX_APR_PER_REVENUE_SETTLE_TO_INSURANCE_FUND_VAULT)?
            .safe_div(PERCENTAGE_PRECISION)?
            .safe_div(
                ONE_YEAR
                    .safe_div(spot_market.insurance_fund.revenue_settle_period.cast()?)?
                    .max(1),
            )?;
        let capped_token_pct_amount = token_amount.safe_div(10)?;
        token_amount = capped_token_pct_amount.min(capped_apr_amount);
    }
    [...]
}
```

Consequently, depositors may absorb spot bad debt even when the same market already has IF-designated revenue recorded in its revenue pool. Because the revenue pool is maintained as a spot balance inside the market's aggregate deposit accounting, reducing cumulative deposit interest socializes the bankruptcy across deposit claims instead of prioritizing that earmarked revenue as first-loss cover.

Remediation

Update the spot bankruptcy resolution path so market-local IF-designated revenue is consumed before depositor interest is reduced. This can be implemented by drawing from `SpotMarket.revenue_pool` inside `resolve_spot_bankruptcy` before computing the socialized loss, or by adding a bankruptcy-specific revenue settlement path that is not blocked by the normal revenue-settle timer or staker settlement cap.

Patch

Resolved in PR#238.

Pending IF Fee Sweep Can Force Bankruptcy Socialization

ID	Severity	Status
OS-VEP-ADV-07	High	✓ Resolved

Description

The `sweep_perp_market_fees` instruction is exposed through the keeper path and its account context requires the state, perp market, quote spot market, and oracle accounts, but no authority signer. As shown in the `SweepPerpMarketFees` account context, the constraints bind the market, quote spot market, and oracle identity, but they do not restrict who may invoke the sweep.

programs/velocity/src/instructions/keeper.rs

```
3485 pub struct SweepPerpMarketFees<'info> {
3486     pub state: AccountLoader<'info, State>,
3487     #[account(
3488         mut,
3489         seeds = [b"perp_market", perp_market_index.to_le_bytes().as_ref()],
3490         bump,
3491         has_one = oracle @ ErrorCode::InvalidOracle,
3492     )]
3493     pub perp_market: AccountLoader<'info, PerpMarket>,
3494     /// The perp market's quote spot market (enforced by the PDA derivation)
3495     #[account(
3496         mut,
3497         seeds = [b"spot_market",
            perp_market.load()?.quote_spot_market_index.to_le_bytes().as_ref()],
3498         bump
3499     )]
3500     pub spot_market: AccountLoader<'info, SpotMarket>,
3501     /// CHECK: must be `perp_market.oracle` (enforced by `has_one` above)
3502     pub oracle: UncheckedAccount<'info>,
3503 }
```

Public sweep account context without an authority signer

The sweep then computes available pnl-pool tokens by reserving only positive `net_user_pnl` plus the configured buffer before draining fee ledger balances. In `sweep_market_fees`, the insurance-fund carveout is transferred from the perp market `pnl_pool` into the quote spot market `revenue_pool`, and the corresponding `pending_if_fee` counter is consumed, so the pending tranche is no longer visible to later bankruptcy resolution.

Perp bankruptcy resolution treats unswept `pending_if_fee` as the first tranche of loss coverage before drawing from the shared insurance fund. The `resolve_perp_bankruptcy` flow consumes that counter-only tranche against the bankrupt account's negative quote asset amount, meaning the fee value remains in the pnl pool instead of being moved as tokens during the bankruptcy path.

programs/velocity/src/controller/liquidation.rs

```rust
pub fn resolve_perp_bankruptcy([...]) -> VelocityResult<u64> {
    [...]
    // Tranche 1: the market's own in-transit insurance fees (`pending_if_fee`)
    // are consumed BEFORE the shared IF vault is tapped. Counter-only: the
    // pending claim and the forgiven loss are both claims on future pnl-pool
    // inflows, so canceling one against the other needs no token movement —
    // the fee value that would have swept to the revenue pool stays in the
    // pnl pool backing the counterparties this spares from socialization.
    let pending_if_payment: u128 = {
        let mut perp_market = perp_market_map.get_ref_mut(&market_index)?;

        let pending_if_payment = loss
            .unsigned_abs()
            .min(perp_market.fee_ledger.pending_if_fee);

        if pending_if_payment > 0 {
            perp_market
                .fee_ledger
                .consume_pending_if(pending_if_payment)?;
            msg!("bankruptcy pending_if_fee tranche: {}", pending_if_payment);
        }

        pending_if_payment
    };
    let loss_after_pending = loss.safe_add(pending_if_payment.cast::<i128>()?)?;
    [...]
}
```

Bankruptcy consumes unswept pending IF fees first

If the pending IF-fee tranche was already swept, the bankruptcy path proceeds with a larger remaining loss. The later socialization branch in `resolve_perp_bankruptcy` applies any uncovered loss through cumulative funding-rate updates and records total social loss, so clearing `pending_if_fee` before bankruptcy can move a loss that would have been absorbed by the market-specific pending tranche into the shared insurance-fund or social-loss path.

The bankruptcy wrapper attempts to settle quote-market revenue into the insurance fund before resolving the bankruptcy, but `attempt_settle_revenue_to_insurance_fund` only transfers revenue when the revenue-settle timer permits it. When the timer is not due, swept pending IF fees remain in the revenue pool rather than being restored as available bankruptcy coverage, leaving the resolver to operate without the first-loss tranche that existed before the sweep. Thus, an unprivileged keeper can make a bankruptcy strictly worse for counterparties and the shared insurance fund.

Remediation

Prevent the public perp-market fee sweep from consuming `pending_if_fee` while unresolved perp bankruptcy losses could use it as first-loss coverage. The sweep should reserve pending bankruptcy-loss offsets before transferring the IF carveout to the quote revenue pool, or bankruptcy resolution should be able to consume an equivalent recently swept, settlement-reserved IF-fee bucket before drawing shared insurance or socializing loss.

Patch

Resolved in PR#397.

Direct Spot Liquidation Accepts a Margin-Invalid Deposit Oracle

ID	Severity	Status
OS-VEP-ADV-08	High	Resolved

Description

Velocity applies oracle validity by action, and the policy used for liquidation is less restrictive than the policy used for margin calculation. As shown in `is_oracle_valid_for_action`, `VelocityAction::MarginCalc` rejects `TooUncertain` and `StaleForMargin` oracle states, while `VelocityAction::Liquidate` rejects only `NonPositive` or `TooVolatile` oracle states. This means an oracle that the protocol records as invalid for margin can still pass the liquidation-specific validity check.

The direct spot liquidation path refreshes the user's deposit market using `VelocityAction::Liquidate`, then reads the accepted asset oracle price from the same price data. In `liquidate_spot`, this asset-side price and market metadata are captured before the margin calculation and before the liquidation transfer amounts are computed, so a positive but margin-invalid deposit oracle can enter the liquidation flow as the asset price.

The liquidation margin calculation does track whether deposit oracles were valid, but the direct liquidation gate shown in `liquidate_spot` does not consult that flag. It checks only the numeric cross-margin result through `meets_cross_margin_requirement()` for users not already being liquidated and `can_exit_cross_margin_liquidation()` for users already in liquidation. The surrounding margin calculation code updates `all_deposit_oracles_valid`, while the margin calculation state methods used here compare only collateral and requirement values.

programs/velocity/src/controller/liquidation.rs

```
pub fn liquidate_spot([...]) -> VelocityResult {
    [...]
    if !user.is_cross_margin_being_liquidated()
        && margin_calculation.meets_cross_margin_requirement()
    {
        msg!("margin calculation: {:?}", margin_calculation);
        return Err(ErrorCode::SufficientCollateral);
    } else if user.is_cross_margin_being_liquidated()
        && margin_calculation.can_exit_cross_margin_liquidation()?
    {
        user.exit_cross_margin_liquidation();
        return Ok(());
    }
    [...]
}
```

Liquidation predicates rely on numeric margin status without checking deposit oracle validity.

The accepted asset price is later used to bound the liability transfer implied by the user's deposit and to compute the asset transfer for the selected liability transfer. In `liquidate_spot`, both sizing calculations consume the same `asset_price` obtained after the liquidation-scoped oracle check. It then performs the 5-minute TWAP divergence checks after these calculations and can enter cross-margin bankruptcy when the liability transfer does not cover the margin shortage and the user is bankrupt.

As a result, a stale or too-uncertain but still positive deposit oracle that remains within the later TWAP divergence band may cause direct spot liquidation to treat collateral as lower-value than ordinary margin rules would allow. In that state, the account may be considered liquidatable based on the numeric margin result, collateral transfer sizing may use the depressed asset price, and residual borrow may proceed into the existing spot bankruptcy path, where the controller pays what it can from the insurance fund vault and socializes remaining loss through spot market accounting.

Remediation

For direct spot liquidation, enforce the same deposit-side oracle validity required for margin calculation before using the asset price. This can be done by aligning the `VelocityAction::Liquidate` policy with `VelocityAction::MarginCalc` for deposit oracles, or by explicitly rejecting the liquidation when the computed `margin_calculation.all_deposit_oracles_valid` flag is false before liquidatability checks, transfer sizing, and the bankruptcy transition.

Patch

Resolved in PR#391.

Positive Unsettled PnL Blocks Liquidation of Larger Riskier Negative PnL

ID	Severity	Status
OS-VEP-ADV-09	High	Resolved

Description

`liquidate_perp_pnl_for_deposit` is intended to transfer a user's negative settled perp PnL to the liquidator in exchange for the user's spot deposit, but it first rejects liquidation of a riskier contract tier while safer liabilities remain. The tier guard depends on `calculate_user_safest_position_tiers`, which treats every non-available perp position as an outstanding perp liability. As shown in `calculate_user_safest_position_tiers`, the perp loop only checks `is_available()` before folding the market's `contract_tier` into the safest liability tier; it does not distinguish negative unsettled PnL from a positive PnL claim.

programs/velocity/src/math/margin.rs

```
pub fn calculate_user_safest_position_tiers(
    user: &User,
    perp_market_map: &PerpMarketMap,
    spot_market_map: &SpotMarketMap,
) -> VelocityResult<(AssetTier, ContractTier)> {
    [...]
    for spot_position in user.spot_positions.iter() {
        if spot_position.is_available() || spot_position.balance_type ==
        SpotBalanceType::Deposit {
            continue;
        }
        let spot_market = spot_market_map.get_ref(&spot_position.market_index)?;
        safest_tier_spot_liablity = min(safest_tier_spot_liablity,
        spot_market.asset_tier);
    }
    for market_position in user.perp_positions.iter() {
        if market_position.is_available() {
            continue;
        }
        let market = &perp_market_map.get_ref(&market_position.market_index)?;
        safest_tier_perp_liablity = min(safest_tier_perp_liablity,
        market.contract_tier);
    }
    Ok((safest_tier_spot_liablity, safest_tier_perp_liablity))
}
```

Safest-tier calculation includes all non-available perp positions.

A zero-base perp row with unsettled PnL is non-available regardless of whether the PnL is positive or negative. The `PerpPosition` availability helpers show that `has_unsettled_pnl()` returns true whenever `base_asset_amount == 0` and `quote_asset_amount != 0`, so a positive `quote_asset_amount` makes the row non-available even though it is not a liability owed by the user.

programs/velocity/src/state/user.rs

```rust
pub fn is_available(&self) -> bool {
    !self.is_open_position()
        && !self.has_open_order()
        && !self.has_unsettled_pnl()
        && self.isolated_position_scaled_balance == 0
        && !self.is_being_liquidated()
}

pub fn is_open_position(&self) -> bool {
    self.base_asset_amount != 0
}
pub fn has_open_order(&self) -> bool {
    self.open_orders != 0 || self.open_bids != 0 || self.open_asks != 0
}
pub fn margin_requirement_for_open_orders(&self) -> VelocityResult<u128> {
    self.open_orders
        .cast::<u128>()?
        .safe_mul(OPEN_ORDER_MARGIN_REQUIREMENT)
}
pub fn has_unsettled_pnl(&self) -> bool {
    self.base_asset_amount == 0 && self.quote_asset_amount != 0
}
```

Positive zero-base unsettled PnL makes a perp position non-available.

In the liquidation path, the target perp position is separately required to have `base_asset_amount == 0` and negative unsettled PnL, so the route is operating on an actual perp debt. After entering liquidation and canceling orders, however, it computes the user's safest outstanding tiers and compares the target market's `contract_tier` against those tiers, as shown in `liquidate_perp_pnl_for_deposit`. If the user also has a positive-PnL claim in a safer A/B-tier market, that positive row can become the safest perp "liability" used for this comparison.

programs/velocity/src/controller/liquidation.rs

```rust
pub fn liquidate_perp_pnl_for_deposit([...]) -> VelocityResult {
    [...]
    let (safest_tier_spot_liability, safest_tier_perp_liability) = liquidation_mode
        .calculate_user_safest_position_tiers(user, perp_market_map, spot_market_map)?;
    let is_contract_tier_violation =
        !(contract_tier.is_as_safe_as(&safest_tier_perp_liability,
        &safest_tier_spot_liability));
    [...]
}
```

Liquidation compares the target contract tier against computed safest tiers.

When the target negative PnL is in a `Speculative` or `HighlySpeculative` market, the safer positive-PnL row can make `is_contract_tier_violation` true. The same liquidation function then returns `TierViolationLiquidatingPerpPnl` before calculating or applying the PnL and deposit transfers, preventing the available spot deposit from reducing the larger riskier negative PnL.

programs/velocity/src/controller/liquidation.rs

```
pub fn liquidate_perp_pnl_for_deposit([...]) -> VelocityResult {
    [...]
    if is_contract_tier_violation {
        msg!(
            "liquidating contract tier={:?} pnl is riskier than outstanding {:?} &
            {:?}",
            contract_tier,
            safest_tier_perp_liability,
            safest_tier_spot_liability
        );
        return Err(ErrorCode::TierViolationLiquidatingPerpPnl);
    }
    [...]
}
```

Tier violation aborts the PnL-for-deposit liquidation.

For an undercollateralized account whose remaining asset is a spot deposit and whose debt is negative perp PnL, this can block the intended liquidation path whenever the account also holds positive unsettled PnL in a safer market. If that positive PnL cannot be settled because the safer market's PnL pool is insufficient, the positive row may continue to block liquidation of the riskier debt, leaving the deposit unavailable to cover the negative PnL and increasing the chance that the unresolved loss proceeds to bankruptcy, insurance fund loss, or socialization.

Remediation

Update `calculate_user_safest_position_tiers` so perp positions are included in the safest perp liability tier only when they represent genuine user liabilities. In particular, for zero-base unsettled-PnL positions, include the market tier only when `quote_asset_amount < 0` and ignore positive unsettled PnL claims.

Patch

Resolved in PR#244.

Dust Deposit Bricks Spot-Bankruptcy Resolution When Loss Exceeds Deposits

ID	Severity	Status
OS-VEP-ADV-10	High	Resolved

Description

The spot-bankruptcy socialization helper only treats a market with exactly zero deposits as having nothing to haircut. As shown in `calculate_cumulative_deposit_interest_delta_to_resolve_bankruptcy`, any nonzero `total_deposits` causes the helper to derive the deposit-interest reduction from the uncovered borrow amount divided by total deposits, without capping the result to the market's current `cumulative_deposit_interest`.

programs/velocity/src/math/liquidation.rs

```rust
pub fn calculate_cumulative_deposit_interest_delta_to_resolve_bankruptcy(
    borrow: u128,
    spot_market: &SpotMarket,
) -> VelocityResult<u128> {
    let total_deposits = get_token_amount(
        spot_market.deposit_balance,
        spot_market,
        &SpotBalanceType::Deposit,
    )?;
    // No depositors to haircut: nothing to socialize against.
    if total_deposits == 0 {
        return Ok(0);
    }
    spot_market
        .cumulative_deposit_interest
        .safe_mul(borrow)?
        .safe_div_ceil(total_deposits)
}
```

Uncapped deposit-interest delta helper

`resolve_spot_bankruptcy` first limits the insurance-fund contribution so the vault keeps a one-unit remainder, then treats the remaining borrow as `loss_to_socialize`. The controller passes that remaining loss directly into the helper, as shown in the `resolve_spot_bankruptcy` excerpt, so the size of the computed deposit-interest delta is driven by the post-insurance uncovered loss rather than by the amount of deposits available to absorb it.

programs/velocity/src/controller/liquidation.rs

```rust
pub fn resolve_spot_bankruptcy([...]) -> VelocityResult<u64> {
    [...]
    // subtract 1 so insurance_fund_vault_balance always stays >= 1
    let if_payment =
    borrow_amount.min(insurance_fund_vault_balance.saturating_sub(1).cast()?);
    let loss_to_socialize = borrow_amount.safe_sub(if_payment)?;
    let cumulative_deposit_interest_delta =
        calculate_cumulative_deposit_interest_delta_to_resolve_bankruptcy(
            loss_to_socialize,
            spot_market_map.get_ref(&market_index)?.deref(),
        )?;
    [...]
}
```

Uncovered loss passed into the helper

When the uncovered loss exceeds total deposits, the helper can return a delta greater than the current cumulative deposit interest. The later `resolve_spot_bankruptcy` update subtracts that delta from `cumulative_deposit_interest` with checked arithmetic, so an oversized delta reverts the resolution instead of zeroing the deposit interest and continuing to record the socialized loss.

programs/velocity/src/controller/liquidation.rs

```rust
pub fn resolve_spot_bankruptcy([...]) -> VelocityResult<u64> {
    [...]
    {
        [...]
        spot_market.cumulative_deposit_interest = spot_market
            .cumulative_deposit_interest
            .safe_sub(cumulative_deposit_interest_delta)?;
        [...]
    }
    [...]
}
```

Checked subtraction of the computed delta

The affected controller path is reached through `handle_resolve_spot_bankruptcy`, gated only by `solvency_repair_not_paused`. This creates a denial-of-service condition for thin or abandoned spot markets: a market with zero deposits would follow the zero-deposit branch in the helper, but any positive deposit makes the helper compute an uncapped haircut. If the bankrupt borrow left after the capped insurance-fund payment is larger than total deposits, repeated resolution attempts can revert at the cumulative-interest subtraction, leaving the bankrupt user and bad-debt accounting unresolved until the dust deposit is removed or enough insurance-fund balance is available that the remaining loss no longer exceeds deposits.

Remediation

Clamp the deposit-interest delta returned for spot-bankruptcy socialization to at most the market's current `cumulative_deposit_interest`, or equivalently cap the portion of `loss_to_socialize` charged to depositors at total deposits. If the uncovered loss exceeds all deposits, explicitly handle the residual by reducing deposit interest to zero while still recording the full social loss and completing bankruptcy resolution.

Patch

Resolved in PR#246.

IF Unstake-Cancel Penalizes Canceling Staker for Vault Donations

ID	Severity	Status
OS-VEP-ADV-11	High	✓ Resolved

Description

The IF unstake cancel path can penalize the canceling staker based on the live insurance fund vault balance rather than only the value recorded when the unstake request was created. The public cancel entrypoint uses the `RequestRemoveInsuranceFundStake` account context, which requires the stake authority as signer, and then checks only that the stake belongs to the requested market and that a pending withdraw request exists before passing the current IF vault token balance into `cancel_request_remove_insurance_fund_stake`, as shown in `handle_cancel_request_remove_insurance_fund_stake`.

The request path stores the pending share count and freezes the request-time vault value in `last_withdraw_request_value`. That value is computed from the requested IF shares, the then-current total IF shares, and the then-current vault amount, then capped below the vault balance in `request_remove_insurance_fund_stake`.

On cancel, the controller first applies rebasing using the current vault balance, then calls `calculate_if_shares_lost` with that same live balance. If the math reports lost shares, the controller decreases the canceling staker's IF shares and reduces both `total_shares` and `user_shares`, emits an unstake-cancel record with amount zero, and clears the pending withdraw request. The excerpted cancel controller does not transfer vault tokens to the canceling staker while doing this share reduction.

programs/velocity/src/controller/insurance.rs

```
pub fn cancel_request_remove_insurance_fund_stake([...]) -> VelocityResult {
    [...]
    let if_shares_lost =
        calculate_if_shares_lost(insurance_fund_stake, spot_market,
        insurance_vault_amount)?;

    insurance_fund_stake.decrease_if_shares(if_shares_lost, spot_market)?;

    spot_market.insurance_fund.total_shares = spot_market
        .insurance_fund
        .total_shares
        .safe_sub(if_shares_lost)?;

    spot_market.insurance_fund.user_shares = spot_market
        .insurance_fund
        .user_shares
        .safe_sub(if_shares_lost)?;
    [...]
```

```
18 }
```

Cancel controller burns computed lost shares and clears the request

The share-loss calculation compares the current value of the pending shares against the frozen request-time value. If the current value is higher, it recomputes how many shares would represent the frozen value using `total_shares - n_shares` and `vault_balance - last_withdraw_request_value` because this uses the live vault balance, an unsolicited SPL transfer into the IF vault can inflate the denominator used during cancel. When the non-pending residual share base is small, the integer share conversion may round the retained pending shares down substantially, potentially to zero, so `n_shares - new_n_shares` is burned from the canceling staker.

Under the ordering described in the finding, a staker that retains the residual IF shares can wait for a victim's signed cancel, donate tokens directly into the IF vault before that cancel executes, and cause the cancel to burn the victim's pending shares. Because the vault tokens remain in the IF vault while the victim's shares and the global user share totals are reduced, the remaining shares represent a larger claim on the vault balance.

Remediation

Change the cancel path so unsolicited vault-balance increases cannot create `if_shares_lost` for the canceling staker. In practice, compute cancel restoration from request-time share accounting, or cap share loss to protocol-recognized IF losses rather than increases in the raw SPL vault balance. The cancel operation should restore or preserve the pending shares unless there is an accounted protocol loss since the request, and direct donations to the IF vault should not reduce the canceling staker's `if_shares`, `total_shares`, or `user_shares`.

Patch

Resolved in PR#254.

IF Request-Remove Freezes Exit Value Before Settling Already-Due Revenue

ID	Severity	Status
OS-VEP-ADV-12	High	✓ Resolved

Description

The insurance-fund add flow settles due revenue before minting stake shares, but the request-remove flow does not perform the same settlement before calculating the exiting stake's pending withdrawal value. In `handle_request_remove_insurance_fund_stake`, the handler validates the request, converts the requested token amount into IF shares using the current insurance-fund vault balance, and then calls the controller request function without first attempting to settle revenue into the insurance fund.

The controller records the pending unstake as both a share count and a request-time token value. As shown in `request_remove_insurance_fund_stake`, `last_withdraw_request_value` is derived from the current insurance vault amount and capped below the vault balance, so the request freezes the exiting staker's token-denominated value before any later revenue settlement can increase the value of those shares.

When the unstaking period has elapsed, `remove_insurance_fund_stake` recomputes the current token value of the requested shares but caps the actual withdrawal at the previously recorded `last_withdraw_request_value`. If the insurance-fund vault increased after the request, the current share value can exceed the frozen request value, and the highlighted removal logic pays only the lower amount.

programs/velocity/src/controller/insurance.rs

```
pub fn remove_insurance_fund_stake([...]) -> VelocityResult<u64> {
    [...]
    let amount = if_shares_to_vault_amount(
        n_shares,
        spot_market.insurance_fund.total_shares,
        insurance_vault_amount,
    )?;
    let _if_shares_lost =
        calculate_if_shares_lost(insurance_fund_stake, spot_market,
        insurance_vault_amount)?;
    let withdraw_amount = amount.min(insurance_fund_stake.last_withdraw_request_value);
    [...]
}
```

Removal caps payout at the stale request-time value.

The public revenue-settlement instruction can move due revenue from the spot market vault into the insurance-fund vault once the configured settlement period has elapsed and withdrawals are not paused. The handler calls the insurance controller settlement routine and then transfers the resulting token amount into the insurance-fund vault; the surrounding controller code indicates

that, once staker shares exist, settled revenue accrues as share-price appreciation rather than through newly minted shares.

As a result, already-due revenue settled between a staker's request and removal may remain in the insurance-fund vault instead of being paid to the exiting shares.

Remediation

Settle already-due revenue before recording an IF withdraw request (mirroring the add path), or define the pending unstake as a fixed share count rather than capping the remove payout at a stale request-time token value that predates a legitimate settlement.

Patch

Resolved in PR#252.

IF Add Accepts a Positive Deposit That Mints Zero Shares

ID	Severity	Status
OS-VEP-ADV-13	High	✓ Resolved

Description

The insurance fund add flow accepts a positive staking amount, computes the depositor's shares from the insurance fund vault balance that already exists before the user's tokens are transferred, and then receives the user's tokens afterward. In `handle_add_insurance_fund_stake`, the value passed into the insurance controller is the current vault account amount, so a prior increase to the vault balance is included in the share-price calculation for the incoming deposit.

programs/velocity/src/instructions/if_staker.rs

```rust
pub fn handle_add_insurance_fund_stake<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    controller::insurance::add_insurance_fund_stake(
        amount,
        ctx.accounts.insurance_fund_vault.amount,
        insurance_fund_stake,
        user_stats,
        spot_market,
        clock.unix_timestamp,
        false,
    )?;
    controller::token::receive(
        &ctx.accounts.token_program,
        &ctx.accounts.user_token_account,
        &ctx.accounts.insurance_fund_vault,
        &ctx.accounts.authority,
        amount,
        &mint,
        if spot_market.has_transfer_hook() {
            Some(remaining_accounts_iter)
        } else {
            None
        },
    )?;
    [...]
}
```

Insurance fund add computes shares before receiving the user's tokens.

The controller then derives `n_shares` for the add and applies that value to the staker and market accounting without rejecting the case where a positive deposit mints zero shares. In `add_insurance_fund_stake`, the no-staker bootstrap only seeds shares when total shares are zero;

once any shares already exist, the proportional conversion result is used directly for the depositor.

The share conversion in `vault_amount_to_if_shares` calculates the new shares as a proportion of the current vault balance and total shares. Because this proportional math can round down, a sufficiently inflated vault balance relative to the victim's deposit and existing total shares can make a positive deposit produce `0` shares.

programs/velocity/src/math/insurance.rs

```rust
pub fn vault_amount_to_if_shares(
    amount: u64,
    total_if_shares: u128,
    insurance_fund_vault_balance: u64,
) -> VelocityResult<u128> {
    // relative to the entire pool + total amount minted
    let n_shares = if insurance_fund_vault_balance > 0 {
        // assumes total_if_shares != 0 (in most cases) for nice result for user

        get_proportion_u128(
            amount.cast::<u128>()?,
            total_if_shares,
            insurance_fund_vault_balance.cast::<u128>()?,
        )?
    } else {
        // must be case that total_if_shares == 0 for nice result for user
        validate!(
            total_if_shares == 0,
            ErrorCode::InvalidIFSharesDetected,
            "assumes total_if_shares == 0",
        )?;

        amount.cast::<u128>()?
    };
    Ok(n_shares)
}
```

Share conversion uses proportional vault accounting.

Thus, an attacker that already holds the residual insurance fund shares can donate into the vault before a victim's add so that the victim's positive deposit is under-minted or mints no shares at all. The victim's tokens still enter the vault after the accounting update, increasing the vault value backing the existing shares rather than giving the victim proportional ownership.

The ordinary remove flow redeems existing insurance fund shares against the vault amount at withdrawal time. As shown in `remove_insurance_fund_stake`, the withdrawal amount is derived from the requested shares, total shares, and current vault balance, so the enlarged vault balance

can accrue to the attacker-held shares subject to the normal request and unstaking flow. The impact is a direct value-transfer primitive against insurance fund depositors when the attacker can obtain favorable ordering before the victim add and already controls existing shares.

programs/velocity/src/controller/insurance.rs

```
pub fn remove_insurance_fund_stake([...]) -> VelocityResult<u64> {
    [...]
    let amount = if_shares_to_vault_amount(
        n_shares,
        spot_market.insurance_fund.total_shares,
        insurance_vault_amount,
    )?;
    let _if_shares_lost =
        calculate_if_shares_lost(insurance_fund_stake, spot_market,
        insurance_vault_amount)?;
    let withdraw_amount = amount.min(insurance_fund_stake.last_withdraw_request_value);
    insurance_fund_stake.decrease_if_shares(n_shares, spot_market)?;
    insurance_fund_stake.cost_basis = insurance_fund_stake
        .cost_basis
        .safe_sub(withdraw_amount.cast()?)?;
    spot_market.insurance_fund.total_shares =
        spot_market.insurance_fund.total_shares.safe_sub(n_shares)?;
    spot_market.insurance_fund.user_shares =
        spot_market.insurance_fund.user_shares.safe_sub(n_shares)?;
    [...]
}
```

Removal redeems existing shares against the vault balance.

Remediation

Reject positive IF adds that would mint `n_shares == 0`, and add a caller-supplied minimum-shares (slippage) argument so a donation/front-run cannot silently zero out a depositor's minted shares.

Patch

Resolved in PR#390.

Revenue-Settle APR Cap Derived From Live IF-Vault Balance Is Donation-Inflatable

ID	Severity	Status
OS-VEP-ADV-14	High	Resolved

Description

The public revenue-settlement instruction reads the current token balances from the spot-market vault and the insurance-fund vault, then passes the live `insurance_fund_vault.amount` into `settle_revenue_to_insurance_fund`. As shown in `handle_settle_revenue_to_insurance_fund`, the instruction enforces the market index and settlement timing, but the APR-cap input is still the mutable token-account balance observed at settlement time.

programs/velocity/src/instructions/keeper.rs

```rust
pub fn handle_settle_revenue_to_insurance_fund<'c: 'info, 'info>(
    ctx: Context<'info, SettleRevenueToInsuranceFund<'info>>,
    spot_market_index: u16,
) -> Result<()> {
    [...]
    let spot_vault_amount = ctx.accounts.spot_market_vault.amount;
    let insurance_vault_amount = ctx.accounts.insurance_fund_vault.amount;
    let clock = Clock::get()?;
    let now = clock.unix_timestamp;
    let time_until_next_update = math::helpers::on_the_hour_update(
        now, spot_market.insurance_fund.last_revenue_settle_ts,
        spot_market.insurance_fund.revenue_settle_period,
    )?;
    validate!(time_until_next_update == 0,[...])?;
    // uses proportion of revenue pool allocated to insurance fund
    let token_amount = controller::insurance::settle_revenue_to_insurance_fund(
        spot_vault_amount,
        insurance_vault_amount,
        spot_market,
        now,
        true,
    )?;
    [...]
}
```

Public settlement passes the live insurance fund vault balance

Inside `settle_revenue_to_insurance_fund`, the cap applied when `user_shares` is nonzero is derived from `insurance_vault_amount` and then bounded by one tenth of the available revenue-pool amount. Because the finding states that the insurance fund vault can receive a direct SPL token donation, this makes the same-period APR denominator donation-inflatable: an incumbent staker

can increase the live vault balance immediately before an eligible settlement, raising the APR cap toward the separate percentage-of-revenue-pool bound.

programs/velocity/src/controller/insurance.rs

```rust
pub fn settle_revenue_to_insurance_fund([...]) -> VelocityResult<u64> {
    [...]
    if spot_market.insurance_fund.user_shares > 0 {
        // only allow MAX_APR_PER_REVENUE_SETTLE_TO_INSURANCE_FUND_VAULT or 1/10th of
        revenue pool to be settled
        let capped_apr_amount = insurance_vault_amount
            .cast::<u128>()?
            .safe_mul(MAX_APR_PER_REVENUE_SETTLE_TO_INSURANCE_FUND_VAULT)?
            .safe_div(PERCENTAGE_PRECISION)?
            .safe_div(
                ONE_YEAR
                    .safe_div(spot_market.insurance_fund.revenue_settle_period.cast()?)?
                    .max(1),
            )?;
        let capped_token_pct_amount = token_amount.safe_div(10)?;
        token_amount = capped_token_pct_amount.min(capped_apr_amount);
    }
    [...]
}
```

APR cap uses the live insurance vault amount

After the cap is applied, the controller computes the amount allocated to the insurance fund, requires a nonzero settlement in the public checked path, updates the settlement timestamp, and reduces the market revenue-pool balance by the settled amount. In the staker-owned phase, the surrounding controller logic does not mint new staker shares for the settlement, so the transferred revenue increases the vault value backing existing shares rather than diluting them.

The normal insurance-fund removal path then values requested shares against the live vault balance through `if_shares_to_vault_amount` and pays the resulting withdrawal amount from the insurance fund vault after the unstaking period. This supports the reported recovery path: if the staker owns the affected shares, the donated balance and the additional revenue credited during settlement can be reflected in the share redemption value, subject to the withdrawal request value computed by the unstake flow.

The result is a bypass of the intended per-period insurance-fund revenue APR cap. Temporary capital that is not consumed by the protocol can increase the cap used for a due settlement, causing more market revenue to be moved into the insurance fund than the APR limit would allow if it were based on donation-resistant accounting.

Remediation

Do not use the mutable live IF-vault token balance as the APR-cap denominator; base the cap on donation-resistant accounting (user-share value, a time-weighted IF balance, or settled cost-basis backstop capital) or explicitly ignore direct vault-balance increases when sizing the same-period settlement.

Patch

Resolved in PR#407.

Revenue-Share Sweep Drains PnL Pool Without Reserving Live `net_user_pnl`

ID	Severity	Status
OS-VEP-ADV-15	High	Resolved

Description

`sweep_completed_revenue_share_for_market` iterates the revenue-share escrow orders for a perp market and gates each completed builder or referral payment on the raw token amount held by the market's `pnl_pool`. The function computes that pool balance and compares it only to `fees_accrued` it does not receive `net_user_pnl` or compute a protected amount for live positive user claims, as shown in the revenue-share sweep balance check.

programs/velocity/src/controller/revenue_share.rs

```
pub fn sweep_completed_revenue_share_for_market<'a>([...]) ->
crate::error::VelocityResult<()> {
    [...]
    for i in 0..orders_len {
        [...]
        let pnl_pool_token_amount = get_token_amount(
            perp_market.pnl_pool.scaled_balance,
            quote_spot_market,
            perp_market.pnl_pool.balance_type(),
        )?;
        // TODO: should we add buffer on pnl pool?
        if pnl_pool_token_amount < fees_accrued as u128 {
            msg!(
                "market {} PNL pool has insufficient balance to sweep fees for builder.
                pnl_pool_token_amount: {}, fees_accrued: {}",
                market_index,
                pnl_pool_token_amount,
                fees_accrued
            );
            break;
        }
        [...]
    }
    [...]
}
```

Revenue-share sweep checks only raw PnL-pool balance against accrued fees.

After that balance check, the builder-code path transfers `fees_accrued` directly from `perp_market.pnl_pool` into the builder user's quote spot position and then marks the builder rewards as settled. The ordinary market-fee sweep over the same pool uses a different reserve model. `sweep_market_fees` derives `reserved_claims` from `max(net_user_pnl, 0)` and only treats

the remaining pool balance as available for fee drains, which demonstrates the missing reservation in the revenue-share sweep.

programs/velocity/src/controller/perp_pools.rs

```rust
pub fn sweep_market_fees([...]) -> VelocityResult<(u128, u128, u128)> {
    [...]
    // live user claims stay fully backed by every drain; the buffer is a
    // retention margin on top that only the IF and AMM-provision drains
    // respect — the protocol drain is exempt and goes first (its per-settle
    // cadence keeps each drain small, and unlike the other two its value is
    // not recoverable later anyway)
    let reserved_claims: u128 = net_user_pnl.max(0).cast::<u128>()?;
    let mut available_unbuffered: u128 =
    pnl_pool_tokens.saturating_sub(reserved_claims);
    [...]
}
```

Ordinary fee sweeping reserves positive net user PnL before draining the pool.

The keeper settle path loads the revenue-share escrow when builder codes are enabled, executes `settle_pnl` or `settle_expired_position`, then revokes completed orders and calls `sweep_completed_revenue_share_for_market`. The call passes the market and account maps but does not forward the `net_user_pnl` computed during normal PnL settlement, so the later revenue-share sweep has no claim reserve to apply.

programs/velocity/src/instructions/keeper.rs

```rust
pub fn handle_settle_pnl<'c: 'info, 'info>(
    ctx: Context<'info, SettlePNL>,
    market_index: u16,
) -> Result<()> {
    [...]
    if state.builder_codes_enabled() {
        if let Some(ref mut escrow) = builder_escrow {
            escrow.revoke_completed_orders(user)?;
            if let Some(ref builder_map) = maybe_rev_share_map {
                controller::revenue_share::sweep_completed_revenue_share_for_market(
                    market_index,
                    escrow,
                    &perp_market_map,
                    &spot_market_map,
                    builder_map,
                    clock.unix_timestamp,
                    state.builder_codes_enabled(),
                )?;
            } else {
```

```
                msg!("Builder Users not provided, but RevenueEscrow was provided");
            }
        }
    }
```

Keeper settlement invokes the revenue-share sweep after PnL settlement without passing net user PnL.

The `SettlePNL` account context defines `authority` as a plain `Signer` while the `user` account is mutable, with no account constraint tying the signer to the user. Combined with the keeper flow above, a caller that supplies the required user, escrow, and revenue-share accounts can trigger settlement and then the revenue-share sweep for that user path when builder codes are enabled.

When the PnL pool is tight, this can move builder or referrer fees before all positive aggregate user PnL remains backed. A profitable third-party user who would otherwise settle against the full pool balance may receive only the reduced remaining pool balance after the sweep, creating a direct transfer of available PnL-pool liquidity to the builder or referrer.

Remediation

Pass the settle-time `net_user_pnl`, or an equivalent precomputed protected-claim amount, into `sweep_completed_revenue_share_for_market`. Before paying any completed builder or referral row, compute available sweep capacity as the PnL pool token balance minus `max(net_user_pnl, 0)` and any intended PnL-pool buffer, mirroring the reservation used by `sweep_market_fees`.

Patch

Resolved in PR#392.

Stale Builder Revenue-Share Row Overcharges a Later Non-Builder Order With the Same Order ID

ID	Severity	Status
OS-VEP-ADV-16	High	Resolved

Description

`handle_place_perp_order` prepares builder revenue-share state before the order insertion path has confirmed that an order will be opened. When builder codes are enabled and the builder parameters validate, the handler calls `add_builder_order` with `user.next_order_id` and the market index, then passes the returned sidecar order into `place_perp_order`, as shown in `handle_place_perp_order`.

programs/velocity/src/instructions/user.rs

```
pub fn handle_place_perp_order<'c: 'info, 'info>(
    ctx: Context<'info, PlaceOrder>,
    params: OrderParams,
) -> Result<()> {
    [...]
    let mut builder_order = add_builder_order(
        &mut escrow,
        &user,
        params.builder_idx,
        builder_fee_bps,
        user.next_order_id,
        params.market_index,
    )?;

    controller::orders::place_perp_order(
        &*ctx.accounts.state.load()?,
        &mut user,
        user_key,
        &perp_market_map,
        &spot_market_map,
        &mut oracle_map,
        clock,
        params,
        PlaceOrderOptions::default(),
        &mut builder_order,
    )?;

    Ok(())
}
```

Builder revenue-share row is created before order placement

`place_perp_order` can soft-skip a placement when the resolved `max_ts` is nonzero and already expired. That return happens before the order bit flags are finalized, before the `HasBuilder` bit would be set from the revenue-share order, and before `get_then_update_id!` consumes `user.next_order_id` for the new `Order`.

As a result, the builder revenue-share row can remain keyed to an order id that was never inserted while the user's next order id remains unchanged. The `add_builder_order` implementation also chooses the first available user order slot when creating the revenue-share row, and `revoke_completed_orders` treats a revenue-share row as still open when the user order at that stored slot has the same `order_id`. A later ordinary perp order can therefore reuse the same id and slot without carrying builder attribution, causing the stale sidecar row to continue matching the live order.

The fill-time lookup does not verify the live `Order`'s builder bit before using escrow data. `get_builder_escrow_info` derives the builder fee terms from the supplied escrow by searching only for the taker's `sub_account_id` and `order_id`, so a stale row for the reused id can provide `fee_tenth_bps` and `builder_idx` for an order that was not placed with a builder code.

programs/velocity/src/controller/orders.rs

```rust
fn get_builder_escrow_info(
    escrow_opt: &mut Option<&mut RevenueShareEscrowZeroCopyMut>,
    sub_account_id: u16,
    order_id: u32,
    market_index: u16,
) -> (Option<u32>, Option<u32>, Option<u16>, Option<u8>) {
    if let Some(escrow) = escrow_opt {
        let builder_order_idx = escrow.find_order_index(sub_account_id, order_id);
        let referrer_builder_order_idx =
        escrow.find_or_create_referral_index(market_index);

        let builder_order = builder_order_idx.and_then(|idx|
        escrow.get_order(idx).ok());
        let builder_order_fee_bps = builder_order.map(|order| order.fee_tenth_bps);
        let builder_idx = builder_order.map(|order| order.builder_idx);
        (
            builder_order_idx,
            referrer_builder_order_idx,
            builder_order_fee_bps,
            builder_idx,
        )
    }[...]
}
```

Builder lookup matches escrow rows by sub-account and order id

In the AMM fill path, the fee calculation receives the builder fee terms returned from that lookup, and any nonzero builder fee is accrued into the escrow row. The same path then subtracts the sum of the normal user fee and builder fee from the taker's perp quote balance, so the stale row can cause an unauthorized builder fee to be charged on the later non-builder order.

The trigger is constrained by the following conditions: builder codes must be enabled, the taker must have approved and submitted a builder-coded placement that expires before insertion, a later non-builder order must reuse the same id and slot, and the filler must supply the escrow account. Under those conditions, the taker can be overcharged up to the approved builder fee cap, and the stale row accrues the fee for later revenue-share sweeping.

Remediation

Create or finalize `RevenueShareOrder` rows only after `place_perp_order` has actually inserted the order and the final order id and slot are known, or store the builder terms directly on the inserted `Order` so the sidecar cannot outlive the order it describes. In addition, make fill-time builder-fee lookup require the live taker order to carry `HasBuilder` and ignore or reject escrow rows whose sidecar metadata does not match the live order state.

Patch

Resolved in PR#256.

Pyth Lazer Replay Refreshes a Stale Price as Slot-Fresh (No Max-Age Check)

ID	Severity	Status
OS-VEP-ADV-17	High	✓ Resolved

Description

The `post_pyth_lazer_oracle_update` path is exposed through `UpdatePythLazerOracle`, where the `keeper` account is only required to be a mutable signer and the remaining fixed accounts are the Pyth Lazer storage account and the instructions sysvar. As shown in the account context, there is no admin or privileged authority constraint on the caller, so any signer that supplies a valid Pyth Lazer message and matching remaining oracle accounts can reach the update handler.

programs/velocity/src/instructions/pyth_lazer_oracle.rs

```
146 #[derive(Accounts)]
147 pub struct UpdatePythLazerOracle<'info> {
148     #[account(mut)]
149     pub keeper: Signer<'info>,
150     /// CHECK: Pyth lazer storage account not available to us
151     #[account(
152       address = PYTH_LAZER_STORAGE_ID @ ErrorCode::InvalidPythLazerStorageOwner,
153     )]
154     pub pyth_lazer_storage: UncheckedAccount<'info>,
155     /// CHECK: checked by ed25519 verify
156     #[account(address = solana_program::sysvar::instructions::ID)]
157     pub ix_sysvar: UncheckedAccount<'info>,
158 }
```

Permissionless Pyth Lazer oracle update account context

`handle_update_pyth_lazer_oracle` verifies the signed Pyth Lazer message and then iterates through the feed updates, but its freshness gate only compares the incoming feed timestamp to the cached `publish_time` on the `PythLazerOracle` account. In the update logic, a message with an equal timestamp is not skipped, and any accepted update rewrites the oracle price while setting `posted_slot` to the current Solana slot. The handler does not compare the signed feed timestamp against `Clock::unix_timestamp` before refreshing `posted_slot`, which allows an authentic old message that is not older than the cached timestamp to be restamped as current by slot.

Downstream readers then convert a Pyth Lazer oracle account into `OraclePriceData` in `get_pyth_price`. For Lazer sources, the function uses the stored `posted_slot` as the published slot and computes `delay` as the current slot minus that value, while the signed `publish_time` is carried only as `sequence_id`. This means a replay that refreshes `posted_slot` causes downstream code to observe a near-zero slot delay even when the payload timestamp is stale in wall-clock terms.

`oracle_validity` classifies Lazer oracle staleness from the `delay` value supplied in `OraclePriceData`. As shown in the validity checks, `StaleForMargin` and `StaleForAMM` are driven by slot-delay thresholds rather than the Lazer feed timestamp, so a replayed message that updates

`posted_slot` can bypass those stale-oracle states unless another guard, such as confidence or volatility, rejects the price.

programs/velocity/src/math/oracle.rs

```rust
pub fn oracle_validity([...]) -> VelocityResult<OracleValidity> {
    [...]
    let is_stale_for_amm_immediate = if slots_before_stale_for_amm_immdiate_override != 0 {
        oracle_delay.gt(&slots_before_stale_for_amm_immdiate_override.max(0).cast()?)
    } else {
        true
    };
    let is_stale_for_amm_low_risk = if oracle_low_risk_slot_delay_override != 0 {
        oracle_delay.gt(&oracle_low_risk_slot_delay_override.max(0).cast()?)
    } else {
        oracle_delay.gt(&valid_oracle_guard_rails.slots_before_stale_for_amm)
    };
    let is_stale_for_margin = if matches!(oracle_source, OracleSource::PythLazerStableCoin) {
        oracle_delay.gt(&(valid_oracle_guard_rails
            .slots_before_stale_for_margin
            .saturating_mul(3)))
    } else {
        oracle_delay.gt(&valid_oracle_guard_rails.slots_before_stale_for_margin)
    };
    let oracle_validity = if is_oracle_price_nonpositive {
        OracleValidity::NonPositive
    } else if is_oracle_price_too_volatile {
        OracleValidity::TooVolatile
    } else if is_conf_too_large {
        OracleValidity::TooUncertain
    } else if is_stale_for_margin {
        OracleValidity::StaleForMargin
    } else if !has_sufficient_number_of_data_points {
        OracleValidity::InsufficientDataPoints
    } else if is_stale_for_amm_immediate || is_stale_for_amm_low_risk {
        OracleValidity::StaleForAMM {
            immediate: is_stale_for_amm_immediate,
            low_risk: is_stale_for_amm_low_risk,
        }
    } else {
        OracleValidity::Valid
    };
    [...]
}
```

Oracle staleness classified from slot delay

For markets or spot assets configured with `OracleSource::PythLazer`, `PythLazer1K`, `PythLazer1M`, or `PythLazerStableCoin`, this decouples signed publish freshness from the stale-oracle protection used by AMM and margin paths. During a Lazer publishing gap or outage, a caller may be able to replay the last accepted signed message, keep `oracle_delay` close to zero, and make an old price appear fresh to trading, settlement, borrowing, or liquidation flows that rely on the common oracle-validity result. The impact remains dependent on a live Lazer-configured market, a real price divergence window, and the other downstream oracle guards.

Remediation

In `handle_update_pyth_lazer_oracle`, reject each feed update when its signed feed-update timestamp is older than a configured maximum age relative to `Clock::unix_timestamp`, and perform that wall-clock recency check before updating price fields or refreshing `posted_slot`. In addition, make Lazer oracle validity account for signed `publish_time` freshness so a current `posted_slot` alone cannot make a stale payload valid.

Patch

Resolved in PR#389.

Cross-Market Builder Sidecar Can Sweep The Wrong PnL Pool

ID	Severity	Status
OS-VEP-ADV-18	High	Resolved

Description

Builder-fee sidecar rows are created in the taker's `RevenueShareEscrow` before the perp order is installed, but the row lookup used during fills does not prove that the sidecar belongs to the live order being filled. In `RevenueShareEscrow::find_order_index`, the search key is limited to the taker's `sub_account_id` and `order_id`, even though each `RevenueShareOrder` also stores the market type, market index, builder index, fee terms, and state flags. As a result, a stale row from one perp market can be selected for a later order in another perp market when the same taker subaccount reuses the same order id.

programs/velocity/src/state/revenue_share.rs

```
pub fn find_order_index(&self, sub_account_id: u16, order_id: u32) -> Option<u32> {
    for i in 0..self.orders_len() {
        if let Ok(existing_order) = self.get_order(i) {
            if existing_order.order_id == order_id
                && existing_order.sub_account_id == sub_account_id
            {
                return Some(i);
            }
        }
    }
    None
}
```

Builder sidecar lookup matches only subaccount and order id.

The stale-row condition is reachable because builder rows are added by the instruction layer before `controller::orders::place_perp_order` performs all no-op placement checks. The placement path can return successfully for an already-expired `max_ts` before it constructs the live `Order`, sets the order's `HasBuilder` bit, or advances `next_order_id`. The code does not remove the just-created revenue-share row in that soft-skip case. The row therefore remains open with its original market metadata while the user can later place a different order that reuses the same order id.

When the later order fills, the fill path obtains builder escrow information using the weak sidecar lookup and passes the resulting builder fee basis points into fee calculation. In `settle_amm_house_fill`, any computed builder fee is accrued onto the selected escrow row, while the same builder fee is included in the taker's quote-position debit for the current fill's market. This charges the market-B taker based on a sidecar that may still describe market A.

After the row is marked completed, sweeping is keyed by the market fields stored in the `RevenueShareOrder`, not by the market where the mismatched fill occurred.

`sweep_completed_revenue_share_for_market` filters completed builder rows by the stored perp `market_index`, checks that market's PnL pool balance, and transfers `fees_accrued` from that pool to the approved builder's quote spot position. If the stale row originated in market A but accrued fees during a market-B fill, the builder can be paid from market A's PnL pool without a corresponding market-A fill fee.

This results in a cross-market value movement. The later market's taker is debited for an unauthorized builder fee, while the builder is paid liquid quote from the stale row's original market PnL pool. When the builder beneficiary is a Strategy Vault PDA, the payment can raise vault NAV without minting shares.

Remediation

Bind builder-fee sidecar rows to the live order before using them. At placement time, remove or reset any builder sidecar row created for an order whenever `place_perp_order` returns a no-op or otherwise does not install the order and set `HasBuilder`. At fill time, only charge and accrue a builder fee when the live order carries builder attribution and the matching revenue-share row agrees with the live order's `market_type`, `market_index`, `sub_account_id`, `order_id`, builder index, and fee terms. If any field does not match, treat the row as invalid for that fill and do not debit the taker or accrue builder fees to it.

Patch

Resolved in PR#305.

Perp Bankruptcy Socialization Double-Credits The AMM Via A Stale last_cumulative_funding_rate

ID	Severity	Status
OS-VEP-ADV-19	High	Resolved

Description

When `resolve_perp_bankruptcy` reaches a residual perp loss after the pending insurance-fee tranche, insurance-fund payment, and AMM backstop have been applied, it socializes the remaining loss by moving the market cumulative funding rates. The socialization path updates `total_social_loss`, increases the long cumulative funding rate, decreases the short cumulative funding rate, and records the unsettled funding liability, but the same block does not update the AMM's `last_cumulative_funding_rate_long` or `last_cumulative_funding_rate_short`.

programs/velocity/src/controller/liquidation.rs

```rust
pub fn resolve_perp_bankruptcy([...]) -> VelocityResult<u64> {
    [...]
    // socialize loss
    if loss_to_socialize < 0 {
        let mut market = perp_market_map.get_ref_mut(&market_index)?;

        market.total_social_loss = market
            .total_social_loss
            .safe_add(loss_to_socialize.unsigned_abs())?;

        market.cumulative_funding_rate_long = market
            .cumulative_funding_rate_long
            .safe_add(cumulative_funding_rate_delta)?;

        market.cumulative_funding_rate_short = market
            .cumulative_funding_rate_short
            .safe_sub(cumulative_funding_rate_delta)?;

        // The cum-rate bump makes surviving positions owe `loss_to_socialize`
        // in aggregate funding. Record it the same way funding accrual does
        // (net_unsettled_funding_pnl -= protocol funding revenue): without this
        // the obligation is absent from net_unsettled until users settle, and
        // calculate_net_user_pnl overstates aggregate user PnL by the socialized
        // loss in the meantime.
        market.net_unsettled_funding_pnl = market
            .net_unsettled_funding_pnl
            .safe_add(loss_to_socialize.cast()?)?;
    }
    [...]
}
```

That leaves the AMM funding stamp behind the market-level socialization bump. The next AMM funding calculation derives the AMM payment from the difference between the current cumulative rates and the AMM's last recorded rates for both long-side and short-side aggregate exposure. Because the bankruptcy adjustment moved both market cumulative rates relative to the AMM stamp, `calculate_amm_funding_payment` can treat the socialized-loss bump as ordinary funding owed to the AMM rather than as a bankruptcy accounting adjustment.

programs/velocity/src/math/funding.rs

```rust
pub fn calculate_amm_funding_payment([...]) -> VelocityResult<i128> {
    let long_delta =
        cumulative_funding_rate_long.safe_sub(last_cumulative_funding_rate_long.cast()?)?
    let short_delta =
        cumulative_funding_rate_short.safe_sub(last_cumulative_funding_rate_short.cast()?

    let mut amm_payment = 0i128;

    if long_delta != 0 && base_asset_amount_long != 0 {
        // AMM is short to the long-side aggregate; position = -base_long.
        amm_payment = amm_payment.safe_add(_calculate_funding_payment(
            long_delta,
            base_asset_amount_long.safe_mul(-1)?,
        )?)?;
    }

    if short_delta != 0 && base_asset_amount_short != 0 {
        // AMM is long to the short-side aggregate; position = -base_short.
        amm_payment = amm_payment.safe_add(_calculate_funding_payment(
            short_delta,
            base_asset_amount_short.safe_mul(-1)?,
        )?)?;
    }

    amm_payment.safe_div(AMM_TO_QUOTE_PRECISION_RATIO.cast::<i128>()?)
}
```

The stale delta is consumed when the perp-vAMM handles a `FundingUpdated` market event. In that handler, the AMM payment is calculated from the current cumulative rates and the stale AMM stamps, then immediately recorded as AMM PnL before the stamps are advanced to the current cumulative rates. This means the first funding update after residual socialization is the point where the stale bankruptcy bump can be converted into AMM accounting revenue.

The result is that the same residual loss is accounted for once through the socialized funding obligation imposed on surviving positions and again as AMM PnL recorded from the stale cumulative-rate delta. This phantom AMM credit can be paid from the PnL pool through subsequent

funding periods or used as curve capacity, and that the effect can grow if gross open interest increases between the bankruptcy socialization and the next funding update because new positions are stamped at the already-shifted market rate while the AMM remains unsynchronized.

Remediation

In `resolve_perp_bankruptcy`, synchronize `perp_market.amm.last_cumulative_funding_rate_long` and `perp_market.amm.last_cumulative_funding_rate_short` to the updated market cumulative funding rates when residual loss is socialized, after first settling any legitimate AMM funding through the pre-bankruptcy point if needed. Alternatively, account for bankruptcy socialization through a mechanism that does not move market cumulative funding rates relative to the AMM stamp.

Patch

Resolved in PR#306.

AMM Summary Counts Deferred Revenue Share As Spendable Fee Equity

ID	Severity	Status
OS-VEP-ADV-20	High	Resolved

Description

`calculate_perp_market_amm_summary_stats` recomputes `total_fee_minus_distributions` from the perp market's PnL pool, AMM fee pool, net user PnL, and the protocol and insurance-fund pending fee counters. The formula treats remaining pool backing as AMM equity, but it does not subtract builder or referrer revenue-share amounts that have accrued and are still unpaid.

programs/velocity/src/math/perp_market.rs

```rust
pub fn calculate_perp_market_amm_summary_stats([...]) -> VelocityResult<i128> {
    let pnl_pool_token_amount = get_token_amount(
        perp_market.pnl_pool.scaled_balance,
        spot_market,
        perp_market.pnl_pool.balance_type(),
    )?;

    let fee_pool_token_amount = perp_market.amm.fee_pool_token_amount(spot_market)?;

    let pnl_tokens_available: i128 = pnl_pool_token_amount
        .safe_add(fee_pool_token_amount)?
        .cast()?;

    let net_user_pnl = calculate_net_user_pnl(
        &perp_market.amm,
        perp_market_oracle_price,
        perp_market.quote_asset_amount,
        perp_market.net_unsettled_funding_pnl,
    )?;

    pnl_tokens_available
        .safe_sub(net_user_pnl)?
        .safe_sub(perp_market.fee_ledger.pending_protocol_fee.cast()?)?
        .safe_sub(perp_market.fee_ledger.pending_if_fee.cast()?)
}
```

AMM summary calculation omits revenue-share payables

This creates a balance-sheet mismatch while a revenue-share row is outstanding. The fill path records builder and referrer rewards into `RevenueShareOrder.fees_accrued` on the user's `RevenueShareEscrow`, and the taker's quote position is debited for the builder fee, but the summary calculation has no market-level revenue-share liability input. As a result, the lower user claim can appear as additional AMM equity even though the same backing is still owed to the builder or referrer.

The `AmmCrank`-gated summary update handler can commit this recomputed value into the AMM's accounting. When `update_amm_summary_stats` is enabled, `handle_update_perp_market_amm_summary_stats` calls the summary function, computes the difference from the stored value, and applies that correction to AMM fee counters through `apply_summary_stats_correction`.

The omitted liability is not merely an accounting annotation: `sweep_completed_revenue_share_for_market` later pays accrued `fees_accrued` from the perp market PnL pool into the referrer or builder quote spot position, and then clears or removes the escrow order. This reinforces that the accrued revenue share remains a payable against the same pool backing while it is outstanding.

If the inflated summary is committed during that window, downstream logic that relies on `total_fee_minus_distributions` may treat the deferred payable as retained AMM profit. This can let funding, spread, repeg, and curve-affordability logic draw against backing that is also needed to settle the builder or referrer reward, potentially leaving the later-settled claim under-backed. The update requires a genuine `AmmCrank` signer, so the issue depends on an authorized correction being made while the escrow liability remains unpaid.

Remediation

Track aggregate unpaid builder and referrer revenue-share rewards per perp market. Increment that market-level liability whenever `RevenueShareOrder.fees_accrued` is increased, decrement it only when the reward is paid or validly cancelled, and subtract it in `calculate_perp_market_amm_summary_stats` and any PnL-pool reservation logic that decides whether AMM equity is spendable. For an upgrade, reconcile or settle existing escrow rows before enabling summary corrections that depend on the new aggregate counter.

Patch

Resolved in PR#305.

Unswept Referral Rewards Let New Depositors Dilute Existing Vault Shareholders

ID	Severity	Status
OS-VEP-ADV-21	High	Resolved

Description

Strategy Vault share pricing is based on the equity of the Velocity `User` account owned by the vault. As shown in `calculate_equity`, the vault converts `calculate_user_equity` for that `User` into the vault denomination and adds `manager_borrowed_value`, but this path does not include revenue-share rewards that have accrued in a separate `RevenueShareEscrow` and have not yet been swept into the vault-owned `User`.

programs/vaults/src/state/vault.rs

```rust
pub fn calculate_equity([...]) -> VaultResult<u64> {
    [...]
    let spot_market = spot_market_map.get_ref(&self.spot_market_index)?;
    let spot_market_precision = spot_market.get_precision().cast::<i128>()?;
    let oracle_price = oracle_map
        .get_price_data(&spot_market.oracle_id())?
        .price
        .cast::<i128>()?;
    Ok(vault_equity
        .safe_mul(spot_market_precision)?
        .safe_div(oracle_price)?
        .safe_add(self.manager_borrowed_value as i128)?
        .cast::<u64>()?)
}
```

`calculate_equity` prices the vault from the vault-owned `User`.

A deposit is therefore priced against the vault equity value available before any unswept referral reward is credited. In `VaultDepositor::deposit`, the depositor's new shares are calculated from the deposit amount, current total shares, and the `vault_equity` passed by the deposit instruction, so an understated equity value lets the entrant mint shares as though the accrued reward did not exist.

programs/vaults/src/state/vault_depositor.rs

```rust
pub fn deposit([...]) -> Result<()> {
    [...]
    let n_shares = vault_amount_to_depositor_shares(amount, vault.total_shares,
    vault_equity)?;
    [...]
}
```

`deposit` mints shares from the passed vault equity.

The later revenue-share sweep credits accrued referral fees into the referrer's quote spot position. When the vault PDA is the referrer authority, this sweep moves the previously accrued reward into the vault-owned `User` after shares have already been minted, without minting compensating shares for the existing shareholders who economically earned the reward. This ordering lets a new depositor acquire a pro rata claim on reward value that accrued before its deposit.

Remediation

Prevent externally accrued revenue-share receivables from crossing a share-pricing boundary unaccounted. Before pricing deposits, either sweep and crystallize all reward escrows owed to the vault-owned `User`, include those accrued receivables in a vault-aware NAV accumulator used by `calculate_equity`, or route vault-PDA builder and referral rewards to a segregated beneficiary that is outside depositor NAV.

Patch

Resolved in PR#307.

Late Vault Entrant Can Capture A Pending Withdrawer's Accrued Reward

ID	Severity	Status
OS-VEP-ADV-22	High	Resolved

Description

Vault withdrawals are handled as a request followed by a later execution. The request path receives `vault_equity` that the instruction handler calculates from the vault's Velocity `User` account, and this equity value does not include rewards already fixed to the Vault PDA but still held in an external escrow. If such a reward is swept after the request, the recorded request value can be lower than the value represented by the requested shares at execution time.

In `request_withdraw`, the depositor's withdrawal value and share count are calculated from the request-time `vault_equity`, the depositor's current vault shares, and the vault's total shares. The function then stores those values in `last_withdraw_request` and adds the frozen value to `total_withdraw_requested`, so the request carries a fixed token amount forward into the redeem period.

programs/vaults/src/state/vault_depositor.rs

```rust
pub fn request_withdraw([...]) -> Result<()> {
    [...]
    let (withdraw_value, n_shares) = withdraw_unit.get_withdraw_value_and_shares(
        withdraw_amount,
        vault_equity,
        self.get_vault_shares(),
        vault.total_shares,
        rebase_divisor,
    )?;
    validate!(n_shares > 0, [...])?;
    let vault_shares_before: u128 = self.checked_vault_shares(vault)?;
    let total_vault_shares_before = vault.total_shares;
    let user_vault_shares_before = vault.user_shares;
    let protocol_shares_before = vault.get_protocol_shares(vault_protocol);
    self.last_withdraw_request.set(
        vault_shares_before,
        n_shares,
        withdraw_value,
        vault_equity,
        now,
    )?;
    [...]
}
```

Withdrawal request stores request-time value and shares

When `withdraw` later executes, it first derives the current token value of the requested shares from the current `vault_equity`, but the payable amount is capped at the stored request value. As shown in `withdraw`, any increase in share value between request and execution is therefore excluded from the withdrawing depositor's payout.

programs/vaults/src/state/vault_depositor.rs

```
pub fn withdraw([...]) -> Result<(u64, bool)> {
    [...]
    let withdraw_amount = amount.min(self.last_withdraw_request.value);
    [...]
}
```

Withdrawal payout is capped at the frozen request value

The same `withdraw` execution then removes the full requested share amount from the depositor and from the vault share totals, while accounting only for the capped token amount as withdrawn. This means the withdrawing depositor can lose all requested shares even when the current value of those shares is higher than the frozen request value.

programs/vaults/src/state/vault_depositor.rs

```
pub fn withdraw([...]) -> Result<(u64, bool)> {
    [...]
    self.decrease_vault_shares(n_shares, vault)?;
    self.total_withdraws = self.total_withdraws.saturating_add(withdraw_amount);
    self.net_deposits = self.net_deposits.safe_sub(withdraw_amount.cast()?)?;
    vault.total_withdraws = vault.total_withdraws.saturating_add(withdraw_amount);
    vault.net_deposits = vault.net_deposits.safe_sub(withdraw_amount.cast()?)?;
    vault.total_shares = vault.total_shares.safe_sub(n_shares)?;
    vault.user_shares = vault.user_shares.safe_sub(n_shares)?;
    [...]
}
```

Execution burns all requested shares despite the capped payout

THus, a reward swept into vault equity after the victim's request raises the NAV before execution. A later depositor can enter at that higher NAV, and the victim's capped withdrawal leaves the reward-backed excess value allocated across the remaining shares, allowing the later depositor's shares to benefit from value that was already attributable to the earlier share epoch. The vault-level manager and protocol withdrawal variants follow the same pattern of storing request-time value and later paying the minimum of current value and stored value while subtracting requested shares, so the same accounting issue to apply to those withdrawal paths as well.

Remediation

Ensure withdrawal requests are priced against all value attributable to the vault at request time. Before accepting a request, either sweep or crystallize all registered Vault-PDA reward escrows into the NAV used by `calculate_equity`, or extend vault equity accounting to include accrued

receivables that are fixed for the vault but not yet swept. Preserve the stale-value cap only for gains that genuinely arise after the request.

Patch

Resolved in PR#307.

Builder-Reward Donation Can Burn A Cancelling Victim's Vault Claim

ID	Severity	Status
OS-VEP-ADV-23	High	✓ Resolved

Description

A pending vault withdrawal request can lose ownership during cancellation if vault equity is increased after the request is created. The request stores both the requested shares and the request value, and `calculate_shares_lost` compares the current value of those shares against the snapshotted value. When current equity has increased, it recomputes the shares that should remain for the snapshotted value using the reduced share supply and reduced equity base, as shown in `calculate_shares_lost`.

programs/vaults/src/state/withdraw_request.rs

```rust
pub fn calculate_shares_lost(&self, vault: &Vault, vault_equity: u64) ->
VaultResult<u128> {
    let n_shares = self.shares;
    let amount = depositor_shares_to_vault_amount(n_shares, vault.total_shares,
    vault_equity)?;
    let vault_shares_lost = if amount > self.value {
        let new_n_shares = vault_amount_to_depositor_shares(
            self.value,
            vault.total_shares.safe_sub(n_shares)?,
            vault_equity.safe_sub(self.value)?,
        )?;
        validate!(
            new_n_shares <= n_shares,
            ErrorCode::InvalidVaultSharesDetected,
            "Issue calculating delta if_shares after canceling request {} < {}",
            new_n_shares,
            n_shares
        )?;
        n_shares.safe_sub(new_n_shares)?
    } else {
        0
    };
    Ok(vault_shares_lost)
}
```

Because the value-to-share conversion floors the result, a low-share vault can round the retained shares down to zero even though the pending request still represents a positive claim. In that case, the delta between the originally requested shares and the recomputed retained shares is treated as shares lost from profit accrued while the request was pending, rather than as ownership that must be preserved for the cancelling depositor.

`cancel_withdraw_request` applies that calculated share loss to the depositor and to vault accounting unless the depositor owned the entire vault before cancellation. The check shown in `cancel_withdraw_request` only protects a sole owner, so it does not apply when another admitted depositor exists.

programs/vaults/src/state/vault_depositor.rs

```
pub fn cancel_withdraw_request([...]) -> Result<()> {
    [...]
    let vault_shares_lost = self
        .last_withdraw_request
        .calculate_shares_lost(vault, vault_equity)?;
    // only deduct lost shares if user doesn't own 100% of the vault
    let user_owns_entire_vault = total_vault_shares_before == vd_vault_shares_before;
    if vault_shares_lost > 0 && !user_owns_entire_vault {
        self.decrease_vault_shares(vault_shares_lost, vault)?;
        vault.total_shares = vault.total_shares.safe_sub(vault_shares_lost)?;
        vault.user_shares = vault.user_shares.safe_sub(vault_shares_lost)?;
    }
    [...]
}
```

This means the sole-owner guard can be bypassed by maintaining a second depositor in the vault. In a two-depositor case, the victim's one-share withdrawal request can be cancelled after equity inflation, the retained-share calculation floors to zero, and the cancellation burns the victim's only share instead of preserving its pending claim.

The equity inflation path is the revenue-share sweep. `sweep_completed_revenue_share_for_market` can transfer accrued builder or referral fees from the market PNL pool into the selected builder or referrer `User` quote spot position, and the surrounding `settle_pnl` flow loads the revenue-share escrow and revenue-share map from remaining accounts before invoking this sweep.

If the selected reward recipient is the Vault-owned Velocity `User`, that sweep can increase the vault user's equity without minting vault depositor shares. For example, with two depositors holding one share each, equity of 200,000,000, and a victim request worth 100,000,000, a 99,999,999 reward raises equity to 299,999,999 so the victim's retained shares floor to `floor(100,000,000 / 199,999,999) = 0` and its share is burned. The attacker becomes sole owner and withdraws 299,999,999, recovering its own claim, the reward it fronted, and the victim's full 100,000,000 claim,

Remediation

Preserve the pending request's ownership on cancellation. When equity has increased while a withdrawal request is pending, compute the returned shares with conservation-aware rounding and reject any cancellation path that would burn shares while the request still has a positive

snapshotted claim. Also reject positive deposits or reward-crediting paths that would increase vault equity without minting any depositor shares.

Patch

Resolved in PR#307.

Unvalidated Vault Denomination Oracle Can Overmint Depositor Shares

ID	Severity	Status
OS-VEP-ADV-24	High	✓ Resolved

Description

Vault share pricing depends on the vault denomination oracle, but the denomination oracle can be excluded from the validity result that `Vault::calculate_equity` relies on. In `calculate_user_equity`, spot positions that are available are skipped before the function loads their spot market, fetches oracle data with validity, and folds that validity into `all_oracles_valid`. Therefore, when the vault's denomination spot position is available, that market's oracle is not checked by this equity walk.

programs/velocity/src/math/margin.rs

```
pub fn calculate_user_equity([...]) -> VelocityResult<(i128, bool)> {
    [...]
    for spot_position in user.spot_positions.iter() {
        if spot_position.is_available() {
            continue;
        }
        [...]
    }
    [...]
}
```

Available spot positions are skipped before oracle validity is checked.

`Vault::calculate_equity` then treats the returned `all_oracles_valid` flag as sufficient for share pricing and verifies only that the quote-denominated equity is non-negative. After that, it loads the vault's spot market and divides the quote equity by the denomination oracle price to return token-denominated vault equity, so a stale-high denomination price can understate the reported vault equity used by vault accounting.

The price used for this denomination conversion is obtained through `OracleMap::get_price_data`, which loads or returns cached price data but does not calculate or return an `OracleValidity` value. `get_price_data_and_validity` provides the validity-aware path in the equity walk, so this raw fetch leaves the denomination conversion without the staleness, confidence, data-point, or volatility guard.

programs/velocity/src/state/oracle_map.rs

```
pub fn get_price_data(&mut self, id: &OracleIdentifier) ->
VelocityResult<&OraclePriceData> {
    if self.should_get_quote_asset_price_data(&id.0) {
        return Ok(&self.quote_asset_price_data);
    }
```

```
57
58      if self.price_data.contains_key(id) {
59          return self.price_data.get(id).safe_unwrap();
60      }
61
62      let account_info = match self.oracles.get(&id.0) {
63          Some(account_info) => account_info,
64          None => {
65              msg!("oracle pubkey not found in oracle_map: {}", id.0);
66              return Err(ErrorCode::OracleNotFound);
67          }
68      };
69
70      let price_data = get_oracle_price(&id.1, account_info, self.slot)?;
71
72      self.price_data.insert(*id, price_data);
73      self.price_data.get(id).safe_unwrap()
74  }
```

Raw oracle price loading returns price data without validity.

Deposits use `vault.calculate_equity` before calling the depositor share-minting path, and the vault state mints shares from the deposit amount against the supplied vault equity. If the stale-high denomination price makes vault equity appear lower than it should, a depositor can receive too many shares for the amount deposited. When the oracle later recovers, the corrected vault value is split across an inflated share count. This enables direct value extraction from existing vault participants through ordinary deposit and withdrawal mechanics.

Remediation

Fetch the vault denomination oracle through `get_price_data_and_validity` inside the vault share-pricing path, enforce a vault-specific validity policy for that oracle even when the denomination spot position is otherwise available, and reuse the validated price consistently for equity, capacity, fee, deposit, request, and withdrawal calculations.

Patch

Resolved in PR#307 and PR#339.

Zero Utilization Backdates Interest Onto The First Later Borrower

ID	Severity	Status
OS-VEP-ADV-25	High	✓ Resolved

Description

The spot market interest checkpoint can remain stale across an idle epoch with zero borrows. After that idle period, a lender who has deposited into the unpaused market can wait for a victim to create the first later borrow, then trigger cumulative interest accrual so the new nonzero utilization is applied over time that elapsed before the borrow existed.

The root cause is in `calculate_accumulated_interest`: it first compares `now` against `last_interest_ts`, then computes utilization, but when utilization is zero it returns zero interest without advancing any checkpoint. In `calculate_accumulated_interest`, the later elapsed-time calculation still subtracts the old `last_interest_ts` from `now` once utilization becomes nonzero.

The accrual controller applies the returned interest to the market-wide cumulative indices and only updates `last_interest_ts` inside the branch where positive lender and borrower interest is credited. In `update_spot_market_cumulative_interest`, the deposit interest is split between insurance fund, protocol fees, and lenders, while `cumulative_borrow_interest`, `cumulative_deposit_interest`, and `last_interest_ts` are updated together after a positive accrual.

This means the first borrower after a zero-borrow idle span can be charged for the full idle interval, even though their debt did not exist during that period. The impact depends on an empty utilization epoch, an unpaused market, rate and liquidity configuration, and a first later borrower, so the extractable amount is bounded by elapsed interest and the victim borrow.

Remediation

Advance `spot_market.last_interest_ts` whenever `now` is greater than the current checkpoint, including the zero-utilization path that returns zero interest.

Patch

Resolved in PR#395.

Unchecked Swap Outputs Turn The Deposit Cap Into A Global Exit And Repayment Lock

ID	Severity	Status
OS-VEP-ADV-26	High	Resolved

Description

The daily deposit cap is implemented as a check over the spot market's resulting aggregate deposit level, not as an admission check over a specific user deposit delta. In `check_deposit_limits`, the market's current deposit balance is converted to a token amount and compared against a maximum derived from the deposit TWAP, guard threshold, and configured daily cap, so the predicate fails whenever the shared post-state remains above the cap.

The limited balance-update helper shown in `update_spot_balances_and_cumulative_deposits_with_limits` first mutates the user's spot position and the market balances, then validates both withdraw limits and the deposit cap against the resulting market state. Because `check_deposit_limits` receives only the `SpotMarket`, the cap check is not direction-aware at the call site and can reject later operations based solely on the already-over-cap aggregate deposit state.

programs/velocity/src/controller/spot_position.rs

```
pub fn update_spot_balances_and_cumulative_deposits_with_limits(
    token_amount: u128,
    update_direction: &SpotBalanceType,
    spot_market: &mut SpotMarket,
    user: &mut User,
) -> VelocityResult {
    [...]
    // Enforce the daily deposit cap on the shared credit path so every
    // deposit-crediting caller (transfer_pools, end_swap, transfers) is bound,
    // not just the direct `deposit` instruction. No-op when the market has no
    // cap configured (`max_deposit_bps_per_day == 0`) and on withdraw-direction
    // updates (deposits don't grow), so it only bites a real over-cap deposit.
    validate!(
        check_deposit_limits(spot_market)?,
        ErrorCode::DailyDepositLimit,
        "Spot Market {} has hit daily deposit limit (deposits exceed {} bps above 24h
        twap)",
        spot_market.market_index,
        spot_market.max_deposit_bps_per_day
    )?;

    validate!(
        matches!(
            spot_market.status,
```

```
            MarketStatus::Active | MarketStatus::ReduceOnly | MarketStatus::Settlement
        ),
        ErrorCode::MarketWithdrawPaused,
        "Spot Market {} withdraws are currently paused, market not active or in
        settlement",
        spot_market.market_index
    )?;
    [...]
}
```

Post-update cap validation in the limited balance helper

The swap completion path treats the input and output sides differently. The `end_swap` flow debits the input market through the limited helper, but the output market credit shown here calls the lower-level `update_spot_balances_and_cumulative_deposits` helper directly with a deposit direction. That output credit can therefore increase the destination market's recorded deposits without running the daily deposit-cap admission check.

programs/velocity/src/instructions/user.rs

```
pub fn handle_end_swap<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    validate!(
        amount_out != 0,
        ErrorCode::InvalidSwap,
        "amount_out must be greater than 0"
    )?;
    let out_token_amount_before = user
        .force_get_spot_position_mut(out_market_index)?
        .get_signed_token_amount(&out_spot_market)?;
    update_spot_balances_and_cumulative_deposits(
        amount_out_after_fee.cast()?,
        &SpotBalanceType::Deposit,
        &mut out_spot_market,
        user.force_get_spot_position_mut(out_market_index)?,
        false,
        Some(amount_out.cast()?),
    )?;
    let out_token_amount_after = user
        .force_get_spot_position_mut(out_market_index)?
        .get_signed_token_amount(&out_spot_market)?;
    [...]
}
```

Swap output credited through the lower-level balance helper

Together, these paths allow a swap output to push a capped market above its allowed deposit level, after which unrelated balance changes routed through the limited helper may fail because

the market is still globally over the cap. This results in a market-wide denial of normal exit and debt curing while the over-cap condition persists.

Remediation

Make the cap a delta- and direction-aware admission check applied only to balance-increasing user deposits, so risk-reducing withdrawals and repayments are never gated on it, and apply that admission to swap output credits so the cap cannot be bypassed on the way in.

Patch

Resolved in PR#341.

Post-Bankruptcy Revenue-Share Credit Escapes Setoff And Overdraws Insurance

ID	Severity	Status
OS-VEP-ADV-27	High	Resolved

Description

Cross-margin bankruptcy is represented by a stored user-status bit that can outlive the economic condition that originally set it. During perpetual liquidation, `liquidate_perp` enters bankruptcy when the liquidation does not cover the margin shortage and the live bankruptcy predicate remains true, as shown below. The status then remains set until a later path explicitly clears it.

programs/velocity/src/controller/liquidation.rs

```rust
pub fn liquidate_perp([...]) -> VelocityResult {
    [...]
    let (margin_freed_for_perp_position, _) = calculate_margin_freed(
        user,
        perp_market_map,
        spot_market_map,
        oracle_map,
        liquidation_margin_buffer_ratio,
        margin_shortage,
        Some(liquidation_mode.as_ref()),
    )?;
    margin_freed = margin_freed.safe_add(margin_freed_for_perp_position)?;
    liquidation_mode.increment_free_margin(user, margin_freed_for_perp_position)?;

    if base_asset_amount >= base_asset_amount_to_cover_margin_shortage {
        liquidation_mode.exit_liquidation(user)?;
    } else if liquidation_mode.should_user_enter_bankruptcy(user)? {
        liquidation_mode.enter_bankruptcy(user)?;
    }
    [...]
}
```

Perpetual liquidation persists the bankruptcy status when a liability remains

The live predicate does not treat that status as authoritative. `is_cross_margin_bankrupt` immediately returns false when it encounters any positive spot deposit, even if the user still has a negative cross-margin perpetual quote balance. Consequently, crediting quote tokens after the status was set makes the stored status stale.

programs/velocity/src/math/bankruptcy.rs

```rust
pub fn is_cross_margin_bankrupt(user: &User) -> bool {
    // user is bankrupt iff they have spot liabilities, no spot assets, and no perp exposure

    let mut has_liability = false;

    for spot_position in user.spot_positions.iter() {
        if spot_position.scaled_balance > 0 {
            match spot_position.balance_type {
                SpotBalanceType::Deposit => return false,
                SpotBalanceType::Borrow => has_liability = true,
            }
        }
    }
    [...]
}
```

A positive spot deposit makes the live cross-margin predicate false

The revenue-share sweep can create exactly this state. In `sweep_completed_revenue_share_for_market`, both referral and builder rewards are transferred from the market PnL pool directly into the beneficiary's quote spot position. Although the function excludes vault-owned beneficiaries, neither beneficiary branch checks whether the destination `User` is bankrupt before creating the deposit. The sweep is reachable through `handle_settle_pnl`. Its `SettlePNL` account context requires `authority` to be a signer but does not constrain that signer to the settled user, and the handler invokes the revenue-share sweep after a successful settlement when the relevant accounts are supplied. An ordinary third party can therefore choose when a deferred reward is credited to a beneficiary whose bankruptcy status has already been latched.

The resulting deposit cannot be applied through the ordinary asset-for-PnL liquidation path. `liquidate_perp_pnl_for_deposit` obtains the applicable liquidation mode and rejects the user with `UserBankrupt` based on the stored status before it can transfer the deposit against negative perpetual PnL.

programs/velocity/src/controller/liquidation.rs

```rust
pub fn liquidate_perp_pnl_for_deposit([...]) -> VelocityResult {
    // liquidator takes over remaining negative perpetual pnl in exchange for a user deposit
    // can only be done once the perpetual position's size is 0
    // blocked when 1) user deposit oracle is deemed invalid
    // or 2) user has outstanding liability with higher tier

    let liquidation_mode = get_perp_liquidation_mode(user, perp_market_index)?;
```

```
8
9       validate!(
10          !liquidation_mode.is_user_bankrupt(user)?,
11          ErrorCode::UserBankrupt,
12          "user bankrupt",
13      )?;
14
15      [...]
16  }
```

Asset-for-PnL liquidation rejects a user with the stored bankruptcy status

In contrast, `resolve_perp_bankruptcy` accepts the same stored status as sufficient authorization to proceed. It derives the loss from the user's negative quote balance and calculates the shared-insurance payment from that amount after the pending-insurance-fee tranche, without offsetting the newly credited spot deposit. The resolver then clears the perpetual bad debt before reevaluating the live bankruptcy predicate.

After the debt is cleared, the positive spot deposit causes the predicate to be false and the stale bankruptcy status is removed, leaving the reward with the beneficiary. For example, subject to the applicable insurance limits and available balance, a user with 100 quote of negative perpetual PnL and a subsequently swept 40-quote reward can receive coverage calculated against the 100-quote loss while retaining the 40-quote deposit. The estate is therefore understated by the reward amount: shared insurance may pay more than the beneficiary's net deficit, and any uncovered remainder can proceed to the resolver's socialized-loss path.

Remediation

Before consuming any pending-insurance tranche, withdrawing shared insurance, using another backstop, socializing loss, or clearing debt, make `resolve_perp_bankruptcy` reevaluate `should_user_enter_bankruptcy`. If the stored status is set but the live predicate is false, clear the stale status and return without forgiving debt so ordinary liquidation can apply the user's deposit. Separately, account for rewards accrued before resolution even when they have not yet been swept: maintain a beneficiary-attributed receivable total when revenue share accrues, decrement it when rows are paid or cancelled, and have bankruptcy resolution offset the beneficiary's debt by the lesser of that receivable and the debt while atomically marking the corresponding claim unavailable for later payment. Revenue-share sweeps for an already-bankrupt beneficiary should likewise route the payable amount toward debt repayment instead of creating a withdrawable deposit.

Patch

Resolved in PR#422.

Strictly Reducing Swap Trusts An Invalid Oracle In Its Loss Bound

ID	Severity	Status
OS-VEP-ADV-28	High	Resolved

Description

A malicious maker delegate can exploit a stale-high or uncertain output oracle during a strictly reducing swap. In `handle_begin_swap`, the program retrieves the input and output oracle data without checking their validity, records the flash-loan state, and releases the requested input tokens from the program vault. Consequently, an invalid oracle does not prevent the swap from proceeding to external execution.

Instruction introspection in `handle_begin_swap` requires a matching `end_swap` and limits intervening instructions to whitelisted swap programs. However, a delegate recognized as an authorized signer remains permitted to route the released assets through those programs. The whitelist therefore constrains which DEX programs may execute the swap but does not ensure that the route returns fair value.

In `handle_end_swap`, the program again obtains the input and output oracle prices through raw price-data lookups. It does not retrieve or require the corresponding validity classifications before using these prices later in the protected-swap loss calculation. A stale-high output price can therefore overstate the value returned by the route.

programs/velocity/src/instructions/user.rs

```
pub fn handle_end_swap<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    let mut in_spot_market = spot_market_map.get_ref_mut(&in_market_index)?;
    validate!(
        !in_spot_market.is_operation_paused(SpotOperation::Withdraw),
        ErrorCode::MarketFillOrderPaused,
        "withdraw from market {} paused",
        in_market_index
    )?;
    validate!(
        in_spot_market.flash_loan_amount != 0,
        ErrorCode::InvalidSwap,
        "the in_spot_market must have a flash loan amount set"
    )?;
    let in_oracle_data = oracle_map.get_price_data(&in_spot_market.oracle_id())?;
    let in_oracle_price = in_oracle_data.price;
    let mut out_spot_market = spot_market_map.get_ref_mut(&out_market_index)?;
    validate!(
        !out_spot_market.is_operation_paused(SpotOperation::Deposit),
        ErrorCode::MarketFillOrderPaused,
        "deposit to market {} paused",
```

```
        out_market_index
    )?;
    let out_oracle_data = oracle_map.get_price_data(&out_spot_market.oracle_id())?;
    let out_oracle_price = out_oracle_data.price;
    [...]
}
```

Swap completion reloads raw prices without requiring oracle validity

The output tokens are credited to the user's spot position before the strictly reducing exemption and the final margin checks are evaluated. As shown by the output-position update in `handle_end_swap`, an exact output amount can fully repay the existing borrow and cause the position to be classified as reduced. This mutation removes the invalid liability that a subsequent liability-oracle check would otherwise inspect.

programs/velocity/src/instructions/user.rs

```
pub fn handle_end_swap<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    let out_token_amount_before = user
        .force_get_spot_position_mut(out_market_index)?
        .get_signed_token_amount(&out_spot_market)?;
    update_spot_balances_and_cumulative_deposits(
        amount_out_after_fee.cast()?,
        &SpotBalanceType::Deposit,
        &mut out_spot_market,
        user.force_get_spot_position_mut(out_market_index)?,
        false,
        Some(amount_out.cast()?),
    )?;
    let out_token_amount_after = user
        .force_get_spot_position_mut(out_market_index)?
        .get_signed_token_amount(&out_spot_market)?;
    // update fees
    update_revenue_pool_balances(
        fee.cast()?,
        &SpotBalanceType::Deposit,
        &mut out_spot_market,
        false,
    )?;
    let out_position_is_reduced = out_token_amount_before < 0
        && out_token_amount_before.unsigned_abs() >= amount_out_after_fee.cast()?;
    [...]
}
```

Output repayment mutates the borrow position before final validation

After both positions have been mutated, `handle_end_swap` classifies the swap as strictly reducing when it consumes an existing input deposit and repays an existing output borrow. For a breaker-tripped account or an account with an equity floor, the only route-value protection is a one-percent loss bound computed from the same unvalidated raw prices. An overstated output oracle can make a materially unfavorable execution satisfy this bound.

Finally, `meets_withdraw_margin_requirement_swap` checks liability-oracle validity only if the post-swap margin calculation still contains a margin requirement or liability, and it skips the equity-floor calculation for a strictly reducing swap. Exact repayment can therefore leave no invalid liability for this check to observe while the exemption suppresses the floor that would reveal the economic loss. This permits the delegate to exchange approximately ten dollars of SOL for 100 USDT when the protected account relinquishes one SOL worth approximately 100 USDT at realizable value, capturing roughly 90 USDT before route fees and leaving the account below its 150-USDT floor. The loss can scale with the matched deposit and borrow and with the difference between the raw oracle price and executable value.

Remediation

Use validity-aware oracle reads for both swap legs and require each oracle to be valid for `VelocityAction::MarginCalc` before releasing input-vault tokens and again in `handle_end_swap` before updating either spot position. Do not allow exact repayment to remove the liability before these validity checks complete. For the strictly reducing loss bound, value the input conservatively at the strict maximum and the output at the strict minimum of the current valid price and five-minute TWAP.

Patch

Resolved in PR#344.

Negative Expiry Price Is Clipped Out Of Margin And Equity

ID	Severity	Status
OS-VEP-ADV-29	High	Resolved

Description

When a perpetual market enters `Settlement`, `calculate_perp_position_value_and_pnl` selects the committed `expiry_price` but passes it to the generic oracle-price valuation helper. For a net-long position and a negative expiry price, this path omits the signed base-position loss from unrealized PnL, causing the account's margin collateral to be overstated by the omitted loss.

programs/velocity/src/math/margin.rs

```
pub fn calculate_perp_position_value_and_pnl([...]) -> VelocityResult<(u128, i128, u128, u128)> {
    let valuation_price = if market.status == MarketStatus::Settlement {
        market.expiry_price
    } else {
        oracle_price_data.price
    };
    // the funding must be calculated before calculated the unrealized pnl w simulated lp position
    let unrealized_funding = calculate_funding_payment(
        if market_position.base_asset_amount > 0 {
            market.cumulative_funding_rate_long
        } else {
            market.cumulative_funding_rate_short
        },
        market_position,
    )?;
    let (base_asset_value, unrealized_pnl) =
        calculate_base_asset_value_and_pnl_with_oracle_price(market_position, valuation_price)?;
    let total_unrealized_pnl = unrealized_pnl.safe_add(unrealized_funding.cast()?)?;
    let (worst_case_base_asset_amount, worse_case_liability_value) =
        market_position.worst_case_liability_value(oracle_price_data.price)?;
    // for calculating the perps value, since it's a liability, use the large of twap and quote oracle price
    let worse_case_liability_value = worse_case_liability_value
        .safe_mul(strict_quote_price.max().cast()?)?
        .safe_div(PRICE_PRECISION)?;
    let mut margin_requirement = if market.status == MarketStatus::Settlement {
        0
    }[...]
}
```

Settlement margin uses the clipping oracle-price helper

The same valuation mismatch exists in `calculate_user_equity`. Settlement markets are valued using `expiry_price`, but unrealized PnL is still calculated through the generic helper. Because `calculate_net_equity_for_floor` delegates to this function when an equity floor is configured, that additional withdrawal safeguard does not account for the negative base value either.

programs/velocity/src/math/margin.rs

```
pub fn calculate_user_equity([...]) -> VelocityResult<(i128, bool)> {
    [...]
    for market_position in user.perp_positions.iter() {
        [...]
        all_oracles_valid &=
            is_oracle_valid_for_action(oracle_validity,
            Some(VelocityAction::MarginCalc))?;

        let valuation_price = if market.status == MarketStatus::Settlement {
            market.expiry_price
        } else {
            oracle_price_data.price
        };

        let unrealized_funding = calculate_funding_payment(
            if market_position.base_asset_amount > 0 {
                market.cumulative_funding_rate_long
            } else {
                market.cumulative_funding_rate_short
            },
            market_position,
        )?;

        let (_, unrealized_pnl) =
            calculate_base_asset_value_and_pnl_with_oracle_price(market_position,
            valuation_price)?;

        let pnl = unrealized_pnl.safe_add(unrealized_funding.cast()?)?;

        let pnl_value = pnl
            .safe_mul(quote_oracle_price.cast()?)?
            .safe_div(PRICE_PRECISION_I128)?;

        net_usd_value = net_usd_value.safe_add(pnl_value)?;
    }
    [...]
}
```

Net equity repeats the settlement valuation mismatch

The discrepancy arises because the two available valuation helpers treat negative prices differently. `calculate_base_asset_value_and_pnl_with_oracle_price` replaces a nonpositive price with zero before valuing the base position, whereas `calculate_base_asset_value_with_expiry_price` multiplies the signed position by the signed expiry price. While the former behavior is appropriate to preserve for its live-oracle callers, it is inconsistent with settlement accounting.

A negative committed expiry price is reachable through `calculate_expiry_price`. For net-long market exposure, the calculated best expiry price is bounded only above by the target price and then decremented, so the function does not impose a zero lower bound. The delisting test `delist_market_with_1000_balance_long_negative_expiry_price` demonstrates that the settlement process can commit a negative expiry price and place the market into `Settlement`. It also exercises margin calculation and expired-position settlement under that state, establishing that a negative expiry price is an expected code path rather than an invalid input rejected by the protocol.

Unlike margin and equity calculation, `settle_expired_position` uses the expiry-specific signed helper to construct the closing position delta. It then applies the resulting PnL and closeout fee to the user's quote spot balance. Consequently, settlement recognizes a loss that the preceding withdrawal admission checks omitted. If the user has already removed sufficient collateral, the negative settlement can create an unsecured quote borrow.

In `handle_withdraw`, the requested amount is first applied to the user's spot balance and the resulting account is then checked against the initial margin requirement. Once the incorrect margin and, where configured, equity calculations pass, the handler transfers the requested SPL tokens from the program vault to the user. An account can therefore remove collateral while its unsettled long position still carries an unrecognized negative-expiry loss.

Permissionless settlement can subsequently materialize the omitted loss as quote debt in an account that no longer holds the withdrawn collateral. If the remaining account cannot repay that debt, bankruptcy resolution may pass the shortfall to available insurance resources or quote depositors. The discrepancy scales with the withdrawn collateral and the absolute signed value omitted from the long position.

Remediation

For markets in `Settlement`, calculate unrealized PnL in both `calculate_perp_position_value_and_pnl` and `calculate_user_equity` from `calculate_base_asset_value_with_expiry_price`, the position's quote amount, settled or applicable funding, and the same deterministic closeout fee charged by `settle_expired_position`. Keep the generic live-oracle helper's nonpositive-price behavior unchanged for its other callers.

Patch

Resolved in PR#365.

Withdraw Margin Uses Unaccrued Debt From Read-Only Spot Markets

ID	Severity	Status
OS-VEP-ADV-30	High	✓ Resolved

Description

A borrower can withdraw collateral from one spot market while an outstanding borrow in another spot market is valued using stale cumulative interest. In `handle_withdraw`, `load_maps` declares only the selected withdrawal market writable, even though the subsequent margin calculation depends on every non-empty spot position held by the user.

programs/velocity/src/instructions/user.rs

```
pub fn handle_withdraw<'c: 'info, 'info>([...]) -> anchor_lang::Result<()> {
    [...]
    let remaining_accounts_iter = &mut ctx.remaining_accounts.iter().peekable();
    let AccountMaps {
        perp_market_map,
        spot_market_map,
        mut oracle_map,
    } = load_maps(
        remaining_accounts_iter,
        &MarketSet::new(),
        &get_writable_spot_market_set(market_index),
        clock.slot,
        Some(state.oracle_guard_rails),
    )?
    let mint = get_token_mint(remaining_accounts_iter)?;
    [...]
}
```

Withdrawal loads only the selected spot market as writable

After loading the account maps, `handle_withdraw` calls `update_spot_market_cumulative_interest` only for the selected withdrawal market. Consequently, a distinct market containing the user's debt is not advanced to the current cumulative-borrow index before the withdrawal is assessed.

programs/velocity/src/instructions/user.rs

```
pub fn handle_withdraw<'c: 'info, 'info>([...]) -> anchor_lang::Result<()> {
    [...]
    validate!(!user.is_bankrupt(), ErrorCode::UserBankrupt)?;

    let (spot_market_is_reduce_only, refreshed_liability_twap) = {
        let spot_market = &mut spot_market_map.get_ref_mut(&market_index)?;
        let oracle_price_data = oracle_map.get_price_data(&spot_market.oracle_id())?;
        [...]
```

```
        let pre_refresh_liability_twap =
        spot_market.historical_oracle_data.last_oracle_price_twap;

        controller::spot_balance::update_spot_market_cumulative_interest(
            spot_market,
            Some(oracle_price_data),
            now,
            state.funding_paused()?,
        )?;
        let refreshed_liability_twap =
        spot_market.historical_oracle_data.last_oracle_price_twap;
        spot_market.historical_oracle_data.last_oracle_price_twap =
        pre_refresh_liability_twap;
        [...]
    };
    [...]
}
```

Cumulative interest is refreshed only for the withdrawal market

This omission is enforced by the market map's mutability boundary rather than being merely an absent call. In `get_ref_mut`, any attempt to mutate a market outside the declared writable set fails with `SpotMarketWrongMutability`, so the handler cannot refresh the other liability markets using its current account configuration.

programs/velocity/src/state/spot_market_map.rs

```
pub fn get_ref_mut(&self, market_index: &u16) -> VelocityResult<RefMut<'_,
SpotMarket>> {
    if !self.1.contains(market_index) {
        let caller = Location::caller();
        msg!(
            "Spot market {} not expected to be mutable at {}:{}",
            market_index, caller.file(), caller.line()
        );
        return Err(ErrorCode::SpotMarketWrongMutability);
    }
    let loader = match self.0.get(market_index) {
        Some(loader) => loader,
        None => { let caller = Location::caller(); [...] };
        return Err(ErrorCode::SpotMarketNotFound);
        }
    };[...]
}
```

Read-only markets cannot be mutably accessed

The stale index directly affects liability valuation. `get_token_amount` derives a borrow's token amount by multiplying its scaled balance by the market's stored `cumulative_borrow_interest`. Therefore, an index that has not been advanced excludes interest accrued since the market's previous refresh.

programs/velocity/src/math/spot_balance.rs

```rust
pub fn get_token_amount(
    balance: u128,
    spot_market: &SpotMarket,
    balance_type: &SpotBalanceType,
) -> VelocityResult<u128> {
    let precision_decrease = 10_u128.pow(19_u32.safe_sub(spot_market.decimals)?);

    let cumulative_interest = match balance_type {
        SpotBalanceType::Deposit => spot_market.cumulative_deposit_interest,
        SpotBalanceType::Borrow => spot_market.cumulative_borrow_interest,
    };

    let token_amount = match balance_type {
        SpotBalanceType::Deposit => balance
            .safe_mul(cumulative_interest)?
            .safe_div(precision_decrease)?,
        SpotBalanceType::Borrow => balance
            .safe_mul(cumulative_interest)?
            .safe_div_ceil(precision_decrease)?,
    };

    Ok(token_amount)
}
```

Token amounts depend on the stored cumulative-interest index

The margin calculation iterates over every non-empty spot position and retrieves its corresponding market, including positions outside the selected withdrawal market. As the calculation uses each loaded market's current stored state, a stale liability index can cause it to understate the user's debt and initial-margin requirement.

The withdrawal flow debits the user's selected spot position, evaluates `meets_withdraw_margin_requirement`, and then transfers tokens from the program vault. A borrower may therefore withdraw real SPL tokens from market A while interest due in market B is absent from the admission check. The discrepancy grows with the size of the borrow and the time since market B was refreshed. If the omitted interest exceeds the collateral and recoverable liquidation value, the resulting shortfall may be borne through the insurance fund or cumulative deposit-interest socialization.

Remediation

Before permitting a risk-increasing withdrawal, derive the writable spot-market set from every non-empty spot position that participates in the user's margin calculation, require each corresponding market account to be supplied as writable, and refresh cumulative interest for all of those markets before debiting the user or evaluating initial margin.

Patch

Resolved in PR#361.

Vault Deposit Mints Shares Before Target-Market Interest Refresh

ID	Severity	Status
OS-VEP-ADV-31	High	Resolved

Description

The `deposit` handler determines the Vault's equity and passes that value to `VaultDepositor::deposit` before transferring the entrant's tokens or invoking the Velocity deposit CPI. Consequently, the Vault prices and mutates share ownership using the target spot market's existing state rather than the state that will result from the CPI's interest refresh.

`Vault::calculate_equity` delegates valuation of the Vault-owned Velocity user to `calculate_user_equity` and denominates the resulting value in the Vault's configured spot asset. The surrounding valuation logic converts held spot positions using each market's stored cumulative-interest index, but `calculate_equity` does not refresh those indices. Any accrued but uncranked deposit interest is therefore omitted from the NAV used for share pricing.

programs/vaults/src/state/vault.rs

```
pub fn calculate_equity(
    &self,
    user: &User,
    perp_market_map: &PerpMarketMap,
    spot_market_map: &SpotMarketMap,
    oracle_map: &mut OracleMap,
) -> VaultResult<u64> {
    let (vault_equity, all_oracles_valid) =
        calculate_user_equity(user, perp_market_map, spot_market_map, oracle_map)?;
    validate!(all_oracles_valid, ErrorCode::InvalidEquityValue,"oracle invalid")?;
    validate!(vault_equity >= 0, ErrorCode::InvalidEquityValue, "vault equity
    negative")?;
    let spot_market = spot_market_map.get_ref(&self.spot_market_index)?;
    let spot_market_precision = spot_market.get_precision().cast::<i128>()?;
    [...]
    let (oracle_price_data, oracle_validity) = oracle_map.get_price_data_and_validity(
        MarketType::Spot,
        spot_market.market_index,
        &spot_market.oracle_id(),
        spot_market.historical_oracle_data.last_oracle_price_twap,
        spot_market.get_max_confidence_interval_multiplier()?,
        -1,
        0,
        Some(LogMode::Margin),
    )?;
    validate!(
        is_oracle_valid_for_action(oracle_validity, Some(VelocityAction::MarginCalc))?,
        [...]
```

```
        )?;
        let oracle_price = oracle_price_data.price.cast::<i128>()?;
        [...]
    }
```

Vault equity is calculated from the Velocity user's positions and stored market state.

Within `VaultDepositor::deposit`, the stale `vault_equity` is also used when applying fees and profit share. The function then derives the entrant's shares from the deposit amount, current total shares, and that equity before increasing both the depositor's and Vault's share balances. The entrant therefore owns a proportional Vault claim before the underlying Velocity position is refreshed.

Velocity's `handle_deposit` refreshes the target market's cumulative interest before adding the deposited amount to the Vault-owned user's spot position. This refresh increases the token value represented by the incumbent scaled deposit position, while the entrant's newly supplied tokens are added using the refreshed index. The entrant's tokens do not receive retroactive interest, but the shares already minted to the entrant participate in the resulting increase in Vault equity.

The ordinary `withdraw` path has the mirrored ordering where it calculates equity and calls `VaultDepositor::withdraw`, which determines the payout and burns the requested shares, before invoking the Velocity withdrawal CPI. The withdrawal is consequently priced against the pre-refresh NAV even though the subsequent CPI refreshes the target market before changing the Velocity position.

The `force_withdraw` handler repeats the same sequence by calculating equity and completing the Vault-level withdrawal before calling `velocity_withdraw`. Forced exits can therefore also determine the payout and mutate share ownership from an unrefreshed cumulative-interest state.

An ordinary entrant can wait until deposit interest has accrued without a corresponding market refresh and then deposit immediately before the refresh performed by Velocity. For incumbent equity `E`, omitted interest `I`, and entrant deposit `D`, stale pricing awards the entrant approximately `D / (E + D)` of `I`, subject to integer rounding and Vault fees. This value is transferred from incumbent shareholders because the entrant receives shares at an understated NAV and those shares participate in interest earned before the entrant supplied capital.

Remediation

Refresh cumulative interest for every spot market that can affect the Vault-owned Velocity user's equity before calling `calculate_equity`, applying Vault fees or profit share, or minting, burning, or valuing shares. For the configured deposit and withdrawal market, invoke a synchronization-only Velocity CPI before taking the pre-operation equity snapshot.

Patch

Resolved in PR#403.

Vault Withdrawal Requests And Cancellations Use Stale Interest Epochs

ID	Severity	Status
OS-VEP-ADV-32	High	Resolved

Description

The withdrawal-request boundary can be recorded against stale Vault equity. As shown in `request_withdraw`, the handler loads the Velocity market maps with `None` as the writable spot-market index and immediately passes those maps to `calculate_equity`. It does not refresh cumulative interest before using the resulting equity to create the request.

programs/vaults/src/instructions/request_withdraw.rs

```
pub fn request_withdraw<'info>(
    ctx: Context<'info, RequestWithdraw<'info>>,
    withdraw_amount: u64,
    withdraw_unit: WithdrawUnit,
) -> Result<()> {
    [...]
    let AccountMaps {
        perp_market_map,
        spot_market_map,
        mut oracle_map,
    } = ctx.load_maps(clock.slot, None, vp.is_some(), has_fee_update)?;

    let vault_equity =
        vault.calculate_equity(&user, &perp_market_map, &spot_market_map, &mut
        oracle_map)?;

    let spot_market = spot_market_map.get_ref(&vault.spot_market_index)?;
    let oracle = oracle_map.get_price_data(&spot_market.oracle_id())?;

    vault_depositor.request_withdraw(
        withdraw_amount.cast()?,
        withdraw_unit,
        vault_equity,
        vault,
        &mut vp,
        &mut fee_update,
        clock.unix_timestamp,
        oracle.price,
    )?;

    Ok(())
}
```

Withdrawal request calculates equity without a writable spot market or interest refresh

`cancel_withdraw_request` follows the same ordering. It loads maps without a writable spot market, calculates Vault equity from the loaded state, and supplies that value to the cancellation accounting without first synchronizing accrued interest.

programs/vaults/src/instructions/cancel_withdraw_request.rs

```rust
pub fn cancel_withdraw_request<'info>(
    ctx: Context<'info, CancelWithdrawRequest<'info>>,
) -> Result<()> {
    [...]
    let AccountMaps {
        perp_market_map,
        spot_market_map,
        mut oracle_map,
    } = ctx.load_maps(clock.slot, None, vp.is_some(), has_fee_update)?;

    let vault_equity =
        vault.calculate_equity(&user, &perp_market_map, &spot_market_map, &mut
        oracle_map)?;

    let spot_market = spot_market_map.get_ref(&vault.spot_market_index)?;
    let oracle = oracle_map.get_price_data(&spot_market.oracle_id())?;

    vault_depositor.cancel_withdraw_request(
        vault_equity.cast()?,
        &mut vault,
        &mut vp,
        &mut fee_update,
        clock.unix_timestamp,
        oracle.price,
    )?;

    Ok(())
}
```

Cancellation uses the same stale-equity ordering

This behavior is structural rather than an incidental omitted call. In `AccountMapProvider::load_maps`, a `None` writable spot-market index becomes an empty writable-spot set, so the request and cancellation paths do not load any spot market as writable for an interest update.

programs/vaults/src/state/account_maps.rs

```rust
fn load_maps(
    &self,
    slot: u64,
```

```
    writable_spot_market_index: Option<u16>,
    has_vault_protocol: bool,
    has_fee_update: bool,
) -> VelocityResult<AccountMaps<'info>> {
    [...]
    let remaining_accounts_iter = &mut
    self.remaining_accounts[..end_index].iter().peekable();
    load_maps(
        remaining_accounts_iter,
        &BTreeSet::new(),
        &writable_spot_market_index
            .map(get_writable_spot_market_set)
            .unwrap_or_default(),
        slot,
        None,
    )
}
```

A missing writable index produces an empty writable-spot set

The stale equity becomes part of the ownership boundary. `VaultDepositor::request_withdraw` derives the request's value and share count from the supplied `vault_equity` and stores them in `last_withdraw_request`. On cancellation, `VaultDepositor::cancel_withdraw_request` passes the then-supplied equity to `calculate_shares_lost`, which compares the current value of the requested shares with the frozen request value and removes shares representing post-request gains. If either equity value omits accrued interest, the request epoch or the forfeiture calculation is misdated.

The execution path does not correct the stale request cap before consuming the request. In `withdraw`, the Vault calculates equity and invokes the depositor accounting before calling `velocity_withdraw`. The underlying depositor logic caps the payout at the stored request value and burns the requested shares before that CPI. Consequently, interest accrued before the request but omitted from its snapshot can remain in the Vault after the requesting depositor's shares have been removed.

Velocity refreshes the target spot market's cumulative interest only within the later withdrawal CPI, as shown in `handle_withdraw`. This refresh therefore occurs after the Vault has calculated the payout, applied the stored cap, and updated its share accounting.

programs/velocity/src/instructions/user.rs

```
pub fn handle_withdraw<'c: 'info, 'info>([...]) -> anchor_lang::Result<()> {
    [...]

    let (spot_market_is_reduce_only, refreshed_liability_twap) = {
        let spot_market = &mut spot_market_map.get_ref_mut(&market_index)?;
        let oracle_price_data = oracle_map.get_price_data(&spot_market.oracle_id())?;
```

```
        [...]
        let pre_refresh_liability_twap =
        spot_market.historical_oracle_data.last_oracle_price_twap;

        controller::spot_balance::update_spot_market_cumulative_interest(
            spot_market,
            Some(oracle_price_data),
            now,
            state.funding_paused()?,
        )?;
        [...]
    };[...]
}
```

Velocity refreshes cumulative interest inside the later CPI

The ordering can shift lender return between depositors in both directions. Unrefreshed interest accrued before a request is excluded from the requester's frozen value and can be left for the remaining shareholders when the request is executed. Conversely, if interest accrues during the request window but remains unrefreshed when the request is cancelled, `calculate_shares_lost` can omit that gain and allow the requester to retain shares that the cancellation rule would otherwise forfeit. These paths are available to the authority of an ordinary Vault depositor, and the allocation error grows with the amount of interest accrued since the affected markets were last refreshed.

Remediation

Synchronize cumulative interest for every spot market contributing to Vault equity before the request and cancellation snapshot math.

Patch

Resolved in PR#403.

IF Cancel Before Revenue Settlement Evades Pending-Window Share Forfeiture

ID	Severity	Status
OS-VEP-ADV-33	High	Resolved

Description

When an insurance-fund staker requests removal, `handle_request_remove_insurance_fund_stake` first settles any already-due revenue and reloads the vault balances before calculating the requested shares and freezing their value. This establishes the request-time ownership boundary where revenue already due is included in the snapshot, while subsequent appreciation belongs to the remaining stakers during the pending window.

In contrast, `handle_cancel_request_remove_insurance_fund_stake` does not synchronize due revenue. Its account set omits the state, spot-market vault, program signer, and token program needed to perform settlement, and the handler passes the current insurance-fund vault balance directly into the cancellation controller. Consequently, a signed cancellation executed before an already-due settlement prices the cancellation from a stale vault balance.

The controller function, `cancel_request_remove_insurance_fund_stake`, implements the anti-free-option rule by treating cancellation as withdrawal at the frozen request value followed by re-staking at the current share price. It burns `if_shares_lost` so that appreciation during the pending window remains with the other stakers, but the result depends entirely on the vault balance supplied by the handler.

As shown in `calculate_if_shares_lost`, the requested shares are valued against the supplied vault balance and no shares are forfeited when that value does not exceed `last_withdraw_request_value`. Due-but-unsettled revenue is absent from the vault balance, so a cancellation ordered before settlement appears to have experienced no appreciation and clears the pending request without burning shares.

programs/velocity/src/math/insurance.rs

```rust
pub fn calculate_if_shares_lost(
    insurance_fund_stake: &InsuranceFundStake,
    spot_market: &SpotMarket,
    insurance_fund_vault_balance: u64,
) -> VelocityResult<u128> {
    let n_shares = insurance_fund_stake.last_withdraw_request_shares;
    [...]
    let amount = if_shares_to_vault_amount(
        n_shares,
        spot_market.insurance_fund.total_shares,
        insurance_fund_vault_balance,
    )?;
    let if_shares_lost = if amount > insurance_fund_stake.last_withdraw_request_value {
```

```
            let new_n_shares = vault_amount_to_if_shares(
                insurance_fund_stake.last_withdraw_request_value,
                spot_market.insurance_fund.total_shares.safe_sub(n_shares)?,
                insurance_fund_vault_balance
                    .safe_sub(insurance_fund_stake.last_withdraw_request_value)?,
            )?;
            validate!(
                new_n_shares <= n_shares,
                ErrorCode::InvalidIFSharesDetected,
                "Issue calculating delta if_shares after canceling request {} < {}",
                new_n_shares,
                n_shares
            )?;

            n_shares.safe_sub(new_n_shares)?
        } else {
            0
        };
        Ok(if_shares_lost)
    }
```

The forfeiture calculation returns zero when no appreciation is visible.

The public `settle_revenue_to_insurance_fund` path can then transfer the due revenue into the insurance-fund vault. With existing stakers, the settlement increases the assets backing the existing shares without minting additional shares, thereby increasing their value. Because the settlement instruction has no authority signer account, the staker can place it after the signed cancellation in the same transaction.

This ordering defeats the pending-window forfeiture rule: the requester retains the shares that should have been burned and participates in the subsequent revenue appreciation. Since the staker controls instruction order within the transaction, the behavior is deterministic and can recur whenever revenue is due for settlement during a pending request.

Remediation

Synchronize the revenue ownership epoch before calculating `if_shares_lost`. Expand `CancelRequestRemoveInsuranceFundStake` with the state, spot-market vault, program signer, token program, mint, and any transfer-hook accounts required to settle already-due revenue. Invoke the same due-revenue settlement used by `handle_request_remove_insurance_fund_stake`, reload both vault balances, and only then price the cancellation. If cancellation must remain available without withdrawal-pause-dependent settlement accounts, instead record a settlement watermark when the request is created and calculate forfeiture from an effective balance that includes revenue due since that watermark, preventing deferred settlement from lowering the burn.

Patch

Resolved in PR#366.

Reducing DLOB Match Crystallizes A Too-Uncertain Oracle Loss Below The Equity Floor

ID	Severity	Status
OS-VEP-ADV-34	High	Resolved

Description

The oracle policy explicitly treats `TooUncertain` as invalid for `VelocityAction::FillOrderMatch`. This establishes that DLOB order matching is not intended to proceed when the oracle confidence is insufficient for that action.

programs/velocity/src/math/oracle.rs

```
pub fn is_oracle_valid_for_action([...]) -> VelocityResult<bool> {
    let is_ok = match action {
        Some(action) => match action {
            [...]
            }
            VelocityAction::MarginCalc => !matches!(
                oracle_validity,
                OracleValidity::NonPositive
                    | OracleValidity::TooVolatile
                    | OracleValidity::TooUncertain
                    | OracleValidity::StaleForMargin
            ),
            VelocityAction::TriggerOrder => !matches!(
                oracle_validity,
                OracleValidity::NonPositive | OracleValidity::TooVolatile
            ),
            VelocityAction::SettlePnl => matches!(
                oracle_validity,
                OracleValidity::Valid
                    | OracleValidity::StaleForAMM { .. }
                    | OracleValidity::InsufficientDataPoints
                    | OracleValidity::StaleForMargin
            ),
            VelocityAction::FillOrderMatch
            | VelocityAction::UpdateAmmCache
            | VelocityAction::UpdateLpPoolAum
            | VelocityAction::LpPoolSwap => !matches!(
                oracle_validity,
                OracleValidity::NonPositive
                    | OracleValidity::TooVolatile
                    | OracleValidity::TooUncertain
            ),
            VelocityAction::Liquidate => !matches!(
```

```
                [...]
        },
        [...]
    };
    [...]
}
```

`FillOrderMatch` rejects a `TooUncertain` oracle

However, `fill_perp_order` computes `safe_oracle_validity` and uses it to determine AMM and AMM-JIT availability. It separately checks whether the oracle may supply an oracle-derived order price, but it does not validate `VelocityAction::FillOrderMatch`. Consequently, an explicit DLOB price can remain usable even when the oracle-derived price is unavailable.

The routing logic in `determine_perp_fulfillment_methods` independently appends a `Match` method for each crossing maker order. The `amm_is_available` flag controls only whether AMM methods are considered, so disabling the AMM during an uncertain oracle does not prevent the router from emitting DLOB matches. For every emitted `PerpFulfillmentMethod::Match`, `fulfill_perp_order` calls `fulfill_perp_order_step` with the selected maker price. Although the call receives the optional valid oracle price and AMM-related controls, it does not receive or enforce the action-specific `FillOrderMatch` oracle-validity decision.

The post-fill checks in `fulfill_perp_order` then exempt position-reducing fills from the stronger protections. An exactly closing taker fill uses maintenance margin and skips the taker equity-floor check. Similarly, `select_margin_type_for_perp_maker` classifies a maker whose position becomes zero as non-risk-increasing, and the maker equity-floor check shown here executes only when `maker_risk_increasing` is true. Exact closes on both sides can therefore avoid the equity-floor checks.

The remaining `meets_margin_requirement` gate does not compensate for the exemption. It verifies that cross-margin and isolated-margin collateral meet their calculated requirements, while the final cross-margin comparison only compares `total_collateral` with `margin_requirement`. It does not reject the calculation based on `all_liability_oracles_valid`. A maintenance-margin calculation using a `TooUncertain` liability oracle can therefore satisfy this Boolean check.

programs/velocity/src/state/margin_calculation.rs

```rust
pub fn meets_margin_requirement(&self) -> bool {
    let cross_margin_meets_margin_requirement = self.meets_cross_margin_requirement();
    if !cross_margin_meets_margin_requirement {
        return false;
    }
    for isolated_margin_calculation in self.isolated_margin_calculations.values() {
        if !isolated_margin_calculation.meets_margin_requirement() {
            return false;
        }
```

```
        }
        true
    }

    pub fn meets_margin_requirement_with_buffer(&self) -> bool {
        let cross_margin_meets_margin_requirement =
            self.meets_cross_margin_requirement_with_buffer();
        if !cross_margin_meets_margin_requirement {
            return false;
        }
        for isolated_margin_calculation in self.isolated_margin_calculations.values() {
            if !isolated_margin_calculation.meets_margin_requirement_with_buffer() {
                return false;
            }
        }
        true
    }

    #[inline(always)]
    pub fn meets_cross_margin_requirement(&self) -> bool {
        self.total_collateral >= self.margin_requirement as i128
    }
```

Margin sufficiency does not incorporate oracle-validity flags

A malicious delegate can use these conditions to close a protected account's position against an attacker-controlled opposite position at the uncertain mark. This permanently realizes the protected account's mark loss and records the corresponding positive PnL for the attacker. After the oracle recovers and the settlement checks permit settlement, `settle_pnl` determines the claimable amount from the PnL pool, credits positive settlement as a quote deposit, and updates the attacker's settled PnL.

In the reported regression scenario, the protected account begins with 200 quote collateral, a one-base short entered at $10, and a $150 equity floor. A temporary $100 invalid mark would leave the account recoverable if the oracle returned to $10, but an exact close at $100 realizes the loss below the protected floor. The attacker can subsequently settle the corresponding gain from the shared PnL pool. The loss scales with the closed position size and the uncertain mark's divergence from the true price.

Remediation

Require `is_oracle_valid_for_action(safe_oracle_validity, Some(VelocityAction::FillOrderMatch))` to succeed, before routing or executing any DLOB match. If degraded-oracle DLOB matching must remain available generally, enforce the strict gate at minimum whenever either participant is equity-floor-protected, has a tripped equity breaker, or is admitted through the position-reducing exemption.

Patch

Resolved in PR#415.

Standard Fill Margin Credits A Stale Deposit Into DLOB Bad Debt

ID	Severity	Status
OS-VEP-ADV-35	High	Resolved

Description

Standard margin contexts retain positive deposits even when their price oracle is invalid for margin calculation. Both `MarginContext::standard` and `MarginContext::standard_with_config` initialize `ignore_invalid_deposit_oracles` to `false`, so callers must explicitly opt into excluding such collateral, as shown in the context constructors.

programs/velocity/src/state/margin_calculation.rs

```
impl MarginContext {
    pub fn standard(margin_type: MarginRequirementType) -> Self {
        Self {
            margin_type_config: MarginTypeConfig::Default(margin_type),
            mode: MarginCalculationMode::Standard,
            strict: false,
            ignore_invalid_deposit_oracles: false,
            margin_buffer: 0,
            margin_ratio_override: None,
        }
    }

    pub fn standard_with_config(margin_type_config: MarginTypeConfig) -> Self {
        Self {
            margin_type_config,
            mode: MarginCalculationMode::Standard,
            strict: false,
            ignore_invalid_deposit_oracles: false,
            margin_buffer: 0,
            margin_ratio_override: None,
        }
    }
    [...]
}
```

Standard margin contexts retain invalid deposit values by default

During the spot-position walk, `calculate_margin_requirement_and_total_collateral_and_liability_info` obtains the oracle validity and evaluates it for `VelocityAction::MarginCalc`. An invalid result is recorded in `oracle_valid`, but the calculation does not reject the position at that point and continues constructing a price for valuation. For a positive non-quote spot position, the calculation zeroes `worst_case_weighted_token_value` only when `ignore_invalid_deposit_oracles` is enabled. With a

standard context, the stale weighted value is instead added to total collateral, while the invalid result only updates `all_deposit_oracles_valid`.

programs/velocity/src/math/margin.rs

```
pub fn calculate_margin_requirement_and_total_collateral_and_liability_info([...]) ->
VelocityResult<MarginCalculation> {
    [...]
    for spot_position in user.spot_positions.iter() {
        [...]
        } else {
            [...]

            match worst_case_token_value.cmp(&0) {
                Ordering::Greater => {
                    if calculation.context.ignore_invalid_deposit_oracles && !
                    oracle_valid {
                        msg!(
                            "worst_case_weighted_token_value set to 0 for
                            market_index={}",
                            spot_market.market_index
                        );
                        worst_case_weighted_token_value = 0;
                    }

                    calculation.add_cross_margin_total_collateral(
                        worst_case_weighted_token_value.cast::<i128>()?,
                    )?;

                    calculation.update_all_deposit_oracles_valid(oracle_valid);

                    [...]
        }
    }
    [...]
}
```

Invalid positive deposits still increase total collateral

The resulting validity flag does not participate in `MarginCalculation::meets_margin_requirement`. That predicate checks whether the numeric cross-margin and isolated-margin collateral totals cover their corresponding requirements, allowing collateral derived from an oracle already classified as invalid for margin calculation to satisfy the check.

The post-fill taker path constructs a standard context and accepts the numeric result of `meets_margin_requirement`. The maker loop repeats the same standard-context calculation and predicate. Consequently, either side of an ordinary DLOB match may receive new perp exposure backed by the stale positive deposit; the separate stale-oracle override is not activated in

the described scenario because the perp oracle remains valid. The `FillOrder` account context requires the transaction authority to be able to sign for the filler account, but the filled user account is merely mutable and need not authorize the fill. A public filler therefore supplies its own authorized filler account and does not provide a safeguard against the affected traders' invalid collateral valuation.

programs/velocity/src/instructions/keeper.rs

```
[derive(Accounts)]
pub struct FillOrder<'info> {
    pub state: AccountLoader<'info, State>,
    pub authority: Signer<'info>,
    #[account(
        mut,
        constraint = can_sign_for_user(&filler, &authority)?
    )]
    pub filler: AccountLoader<'info, User>,
    [...]
}
```

Fill authorization is tied to the filler account

Two ordinary accounts can exploit this admission path by taking opposing sides of an in-band DLOB trade: the account backed by stale collateral takes the losing exposure, while the counterparty receives the corresponding profit. During positive-PnL settlement, `settle_pnl` determines the tokens available in the market PnL pool, updates pool balances, and credits the settled amount as a quote deposit, allowing the gain to be paid from shared liquidity.

The configured SOL spot market demonstrates an affected collateral configuration: it uses nine decimals, the `Collateral` asset tier, and initial and maintenance asset weights of 8000 and 9000. The admitted exposure therefore scales with the stale deposit's reported value and its configured asset weight. Once the collateral oracle refreshes, the losing perp exposure remains while the apparent collateral supporting it may disappear. The losing account can then be left with bad debt after the counterparty's gain has been credited from the PnL pool, shifting the resulting shortfall to the market rather than either attacker-controlled account.

Remediation

For every risk-increasing admission path, construct the margin context with `ignore_invalid_deposit_oracles(true)` so that a positive deposit whose oracle is invalid for `VelocityAction::MarginCalc` contributes zero collateral before the numeric margin predicate runs.

Patch

Resolved in PR#361.

Standard Fill Underprices A Stale Liability Into DLOB Bad Debt

ID	Severity	Status
OS-VEP-ADV-36	High	Resolved

Description

Standard perp fills evaluate post-fill solvency with a standard `MarginContext`. As shown by `MarginContext::standard_with_config`, this context disables strict oracle pricing, so it does not require the conservative comparison between the current oracle price and the five-minute TWAP.

programs/velocity/src/state/margin_calculation.rs

```
impl MarginContext {
    [...]
    pub fn standard_with_config(margin_type_config: MarginTypeConfig) -> Self {
        Self {
            margin_type_config,
            mode: MarginCalculationMode::Standard,
            strict: false,
            ignore_invalid_deposit_oracles: false,
            margin_buffer: 0,
            margin_ratio_override: None,
        }
    }
    [...]
}
```

Standard margin contexts disable strict oracle pricing

During `calculate_margin_requirement_and_total_collateral_and_liability_info`, each spot position's oracle is evaluated for `VelocityAction::MarginCalc`, but the resulting validity Boolean is separate from construction of `StrictOraclePrice`. Because the standard fill context has `strict` set to false, a non-quote borrow is valued using the raw current price even when that price is stale for margin calculations, rather than the maximum of the current price and five-minute TWAP that would conservatively value a liability.

The negative-value branch of the spot-position walk adds the weighted borrow value to the account's margin requirement and records the position as a spot liability. In that branch, an invalid oracle only clears `all_liability_oracles_valid`, it does not reject the calculation or replace the stale-low value. The resulting numeric requirement therefore understates a borrow whose value has risen since the stale observation. Although `MarginCalculation` retains `all_liability_oracles_valid`, `meets_margin_requirement` checks only whether cross and isolated collateral totals cover their corresponding numeric requirements. The predicate never reads the liability-oracle flag, allowing a calculation based on an invalid, stale-low borrow price to report that the account satisfies margin.

The post-fill checks in the perp order controller construct standard margin contexts for the taker and each maker, calculate their resulting margin, and call `meets_margin_requirement`. The special stale-oracle handling in the maker path is driven by the perp market's `oracle_stale_for_margin` condition. A valid perp oracle therefore does not trigger this path merely because a separate spot-liability oracle is stale. Consequently, a risk-increasing DLOB match can be accepted using the understated spot-borrow requirement on either participating account.

The `FillOrder` account context requires the transaction authority to be authorized for the filler account, while the order-owning `user` account is mutable without an equivalent signer constraint. Public fillers may therefore execute eligible orders without controlling the affected trader, and the filler authorization does not provide a safeguard against the margin calculation's treatment of the trader's stale liability.

programs/velocity/src/instructions/keeper.rs

```rust
[derive(Accounts)]
pub struct FillOrder<'info> {
    pub state: AccountLoader<'info, State>,
    pub authority: Signer<'info>,
    #[account(
        mut,
        constraint = can_sign_for_user(&filler, &authority)?
    )]
    pub filler: AccountLoader<'info, User>,
    [...]
}
```

Fill authorization is tied to the filler account

After the admitted perp exposure produces a winner, `settle_pnl` determines the amount available from the market's PnL pool and updates the user's quote spot balance. Positive settled PnL is credited as a quote deposit while the corresponding value is supplied through the shared pool, allowing the winning side to realize its gain before the losing account's understated liability is recognized as a shortfall.

The relaunch spot-market configuration documents the units used for initial and maintenance liability weights and related borrow controls. In the surrounding SOL market entry, those weights are configured above 100%, but they are still applied to the stale raw valuation; weighting an underpriced base amount does not correct the missing price increase. Thus, an ordinary pair of accounts may match an in-band perp trade while the perp oracle remains valid, using a stale-low non-quote borrow to support exposure that the refreshed liability value would reject. The losing account can be converted from solvent before the trade into protocol bad debt, while the counterparty's gain is paid from the PnL pool. The potential shortfall grows with the borrow size and the gap between the stale liability price and its refreshed value.

Remediation

Make invalid liability-oracle state enforceable in every risk-increasing margin context. For standard perp fills, reject a risk-increasing taker or maker when its post-fill calculation has `all_liability_oracles_valid == false`. alternatively, conservatively value every liability whose oracle is invalid for `VelocityAction::MarginCalc` using at least the maximum of the current price and five-minute TWAP before evaluating margin. Apply the same policy consistently to both sides of a match.

Patch

Resolved in PR#361.

Bid/Ask TWAP Crank Sets A Third Party's Forced-Close Auction Band

ID	Severity	Status
OS-VEP-ADV-37	High	Resolved

Description

`update_perp_bid_ask_twap` is available to any signer whose own `UserStats` permits bid/ask TWAP updates and records at least 1,000 USDC of insurance-fund stake. These conditions restrict access but do not prevent an ordinary qualifying user from invoking the crank immediately before triggering another user's order.

programs/velocity/src/instructions/keeper.rs

```
pub fn handle_update_perp_bid_ask_twap<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    let keeper_stats = load!(ctx.accounts.keeper_stats)?;
    validate!(
        keeper_stats.can_update_bid_ask_twap(),
        ErrorCode::CantUpdatePerpBidAskTwap,
        "Keeper stats can_update_bid_ask_twap is false"
    )?;

    let min_if_stake = 1000 * QUOTE_PRECISION_U64;
    validate!(
        keeper_stats.if_staked_quote_asset_amount >= min_if_stake,
        ErrorCode::CantUpdatePerpBidAskTwap,
        "Keeper doesnt have min if stake. stake = {} min if stake = {}",
        keeper_stats.if_staked_quote_asset_amount,
        min_if_stake
    )?;
    [...]
}
```

Keeper flag and insurance-fund stake gate

The crank constructs its maker map from caller-supplied remaining accounts. It finds resting bids and asks in those accounts, filters them to the permitted oracle-divergence range, and estimates each side at the required depth. Consequently, a caller can curate the DLOB liquidity sampled by `handle_update_perp_bid_ask_twap`, including by supplying an immediately resting self-signed limit order.

programs/velocity/src/instructions/keeper.rs

```
pub fn handle_update_perp_bid_ask_twap<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    let remaining_accounts_iter = &mut ctx.remaining_accounts.iter().peekable();
    let makers = load_user_map(remaining_accounts_iter, false)?;
```

```
5
6       let depth = perp_market.get_market_depth_for_funding_rate()?;
7
8       let (bids, asks) =
9           find_bids_and_asks_from_users(perp_market, oracle_price_data, &makers, slot,
            now)?;
10      let (bids, asks) = filter_bids_asks_by_oracle_divergence(
11          bids,
12          asks,
13          oracle_price_data.price,
14          BID_ASK_TWAP_MAX_ORACLE_DIVERGENCE_PERCENT,
15      )?;
16      let estimated_bid = estimate_price_from_side(&bids, depth)?;
17      let estimated_ask = estimate_price_from_side(&asks, depth)?;
18      [...]
19  }
```

Caller-supplied maker map and DLOB price estimation

`get_market_depth_for_funding_rate` derives the required sampled depth as one-thousandth of open interest, clamped between 100 and 5,000 times the market's minimum order size. Once this comparatively small threshold is met, the amount that can subsequently execute through a victim's auction is not bounded by the attacker's sampled quote size.

programs/velocity/src/state/perp_market.rs

```
896     pub fn get_market_depth_for_funding_rate(&self) -> VelocityResult<u64> {
897         // base amount used on user orders for funding calculation
898
899         let open_interest = self.get_open_interest();
900
901         let depth = (open_interest.safe_div(1000)?.cast::<u64>()?).clamp(
902             self.market_stats.min_order_size.safe_mul(100)?,
903             self.market_stats.min_order_size.safe_mul(5000)?,
904         );
905
906         Ok(depth)
907     }
```

Open-interest-based depth threshold

The selected bid and ask are passed into the mark-TWAP update path. Although `update_mark_twap` sanitizes the new prices, it writes `last_bid_price_twap`, `last_ask_price_twap`, `last_mark_price_twap`, `last_mark_price_twap_5min`, `mark_std`, and the update timestamp. The code also documents that funding was intentionally separated from this caller-curated crank, but trigger-auction construction still consumes the resulting statistics.

Trigger-auction baselines use the stored bid, ask, and five-minute mark TWAPs relative to the oracle TWAP, while the end-price buffer incorporates `mark_std`. The caller-curated crank may therefore influence both the starting offset and the width or endpoint of a subsequently activated perpetual auction, subject to the configured sanitization and tier-specific clamps.

`handle_trigger_order` loads no writable perpetual-market set before dispatching to the order controller. It therefore cannot refresh the affected market statistics before the trigger calculation consumes them, allowing values written by an earlier instruction in the same transaction to remain authoritative for the auction calculation.

programs/velocity/src/instructions/keeper.rs

```
pub fn handle_trigger_order<'c: 'info, 'info>(
    ctx: Context<'info, TriggerOrder<'info>>,
    order_id: u32,
) -> Result<()> {
    let market_type = match load!(ctx.accounts.user)?.get_order(order_id) {
        Some(order) => order.market_type,
        None => {
            msg!("order_id not found {}", order_id);
            return Ok(());
        }
    };
    validate_spot_dlob_trading_enabled_for_market_type(market_type)?;
    let (writeable_perp_markets, writeable_spot_markets) = (MarketSet::new(),
    MarketSet::new());
    let AccountMaps {
        perp_market_map,
        spot_market_map,
        mut oracle_map,
    } = load_maps([...])?;
    [...]
}
```

Trigger handler loads no writable market set

The `TriggerOrder` account constraints bind the signer only to the mutable `filler` account. The mutable victim `user` account is not bound to that authority, while `user_stats` is only checked for correspondence with the victim. A caller may therefore trigger an eligible order belonging to a third party.

programs/velocity/src/instructions/keeper.rs

```
#[derive(Accounts)]
pub struct TriggerOrder<'info> {
    pub state: AccountLoader<'info, State>,
    pub authority: Signer<'info>,
```

```
3448        #[account(
3449            mut,
3450            constraint = can_sign_for_user(&filler, &authority)?
3451        )]
3452        pub filler: AccountLoader<'info, User>,
3453        #[account(mut)]
3454        pub user: AccountLoader<'info, User>,
3455        #[account(
3456            constraint = is_stats_for_user(&user, &user_stats)?
3457        )]
3458        pub user_stats: AccountLoader<'info, UserStats>,
```

Caller-bound filler and unbound victim account

`update_trigger_order_params` calculates and then unconditionally replaces the victim order's auction duration, start price, and end price. For a `TriggerMarket` order, the derived auction band is the victim's only price protection because the order has no user-supplied limit.

A qualifying caller can compose order placement, the bid/ask TWAP crank, cancellation of the temporary quote, and activation of a victim's eligible `TriggerMarket` stop atomically. This can worsen the victim's forced-close execution and direct the corresponding value to an attacker-selected maker. Because the poisoning threshold does not scale with the victim order after the required DLOB depth is supplied, the resulting loss can scale with the victim's position rather than the attacker's temporary quote.

Remediation

Prevent caller-curated DLOB state from determining a third party's trigger auction. Derive `TriggerMarket` auction start and end prices, duration, and volatility buffer from an oracle-only source that `update_perp_bid_ask_twap` cannot modify, or reject a trigger until a non-zero, protocol-defined interval has elapsed since the bid/ask/mark statistics were last updated. Applying the same DLOB-based update inside `trigger_order` is insufficient because the caller would still select the maker accounts used to compute the statistics.

Patch

Resolved in PR#370.

Standard Fill Uses Unaccrued Spot Debt To Create Bad Debt

ID	Severity	Status
OS-VEP-ADV-38	High	Resolved

Description

The public `fill_order` path loads the target perpetual market as writable but supplies an empty writable spot-market set before invoking `fill_perp_order`. Consequently, spot markets needed to value the taker or a matched maker are available only for immutable access during the fill.

programs/velocity/src/instructions/keeper.rs

```rust
fn fill_order<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    let AccountMaps {
        perp_market_map,
        spot_market_map,
        mut oracle_map,
    } = load_maps(
        remaining_accounts_iter,
        &get_writable_perp_market_set(market_index),
        &MarketSet::new(),
        clock.slot,
        Some(state.oracle_guard_rails),
    )?;

    let (makers_and_referrer, makers_and_referrer_stats) =
        load_user_maps(remaining_accounts_iter, true)?;

    let builder_codes_enabled = state.builder_codes_enabled();
    let mut escrow = if builder_codes_enabled {
        get_revenue_share_escrow_account(
            remaining_accounts_iter,
            &load!(ctx.accounts.user)?.authority,
        )?
    } else {
        None
    };
    controller::orders::fill_perp_order([...])?;
    Ok(())
}
```

The fill handler declares no writable spot markets

This account configuration prevents the fill path from accruing spot interest. As enforced by `SpotMarketMap::get_ref_mut`, a spot market can be borrowed mutably only when its index

belongs to the writable set. Attempting to mutate any market loaded by this handler therefore returns `SpotMarketWrongMutability`.

programs/velocity/src/state/spot_market_map.rs

```rust
#[track_caller]
#[inline(always)]
pub fn get_ref_mut(&self, market_index: &u16) -> VelocityResult<RefMut<'_, SpotMarket>> {
    if !self.1.contains(market_index) {
        let caller = Location::caller();
        msg!(
            "Spot market {} not expected to be mutable at {}:{}",
            market_index,
            caller.file(),
            caller.line()
        );
        return Err(ErrorCode::SpotMarketWrongMutability);
    }[...]
}
```

Mutable access is restricted to the declared writable set

Spot liabilities are represented as scaled balances rather than current token amounts. `get_token_amount` converts a borrow by multiplying its scaled balance by the market's stored `cumulative_borrow_interest`, so interest accrued economically since the index was last updated remains absent from the calculated liability.

programs/velocity/src/math/spot_balance.rs

```rust
pub fn get_token_amount([...]) -> VelocityResult<u128> {
    let precision_decrease = 10_u128.pow(19_u32.safe_sub(spot_market.decimals)?);
    let cumulative_interest = match balance_type {
        SpotBalanceType::Deposit => spot_market.cumulative_deposit_interest,
        SpotBalanceType::Borrow => spot_market.cumulative_borrow_interest,
    };
    let token_amount = match balance_type {
        SpotBalanceType::Deposit => balance
            .safe_mul(cumulative_interest)?
            .safe_div(precision_decrease)?,
        SpotBalanceType::Borrow => balance
            .safe_mul(cumulative_interest)?
            .safe_div_ceil(precision_decrease)?,
    };
    Ok(token_amount)
}
```

Borrow amounts use the stored cumulative-interest index

The cross-margin calculation iterates over every non-empty spot position and obtains each corresponding market immutably. It derives the position's signed token amount from that market state and incorporates negative spot value into the account's margin requirement. Thus, a stale cumulative-borrow index understates the liability used by the admission decision even though the calculation examines the position. The fill executes before the taker's post-fill margin calculation. The resulting account is accepted when `meets_margin_requirement` succeeds, but this check receives the same immutable `spot_market_map` and therefore values any scaled borrow using the unrefreshed index.

Matched DLOB makers are evaluated through an equivalent post-fill calculation after their fills have been accumulated. A risk-increasing maker can therefore receive the same understated-liability treatment when its active spot borrow belongs to an unaccrued market. A borrower can wait while interest remains unaccrued and then participate in an adverse, otherwise valid perpetual fill whose post-fill checks still use the earlier debt index. If the omitted interest is sufficient to cross the account's solvency boundary, the fill can pass while the losing account becomes insolvent once the spot market is subsequently refreshed. Because the handler declares no spot market writable, supplying or ordering additional remaining accounts does not enable interest accrual within this path.

The economic loss is not confined to stale internal accounting. When the winning account's positive perpetual PnL is settled, `settle_pnl` determines the amount available from the market's PnL pool, updates the pool balances, and credits the user's quote spot balance. The protocol can therefore realize the counterparty's gain from the PnL pool while later recognizing additional debt on the losing account. The resulting shortfall is borne by the market, and the potential discrepancy grows with both the scaled borrow and the interest omitted while its market remains unrefreshed.

Remediation

Before executing any risk-increasing perpetual fill, identify every non-empty spot market referenced by the taker's and all matched makers' margin calculations, require each of those market accounts to be present and writable, and update its cumulative interest to the current timestamp before performing the fill or either post-fill margin check. Reject the fill if the supplied writable set is incomplete. If transaction account limits prevent synchronizing every required market, fail closed by rejecting a risk-increasing fill whenever any participant has a liability in a spot market whose `last_interest_ts` exceeds an explicit maximum staleness bound.

Patch

Resolved in PR#361.

Oracle Spread Check Uses Non-Strict Prices

ID	Severity	Status
OS-VEP-ADV-39	High	Resolved

Description

A warm or cold administrator schedules a perpetual market's expiry through `handle_update_perp_market_expiry`. This operation moves the market to `ReduceOnly` and records `expiry_ts`, but it does not commit `expiry_price` or move the market to `Settlement`, leaving those changes to a later administrative finalization transaction. The market's `is_in_settlement` predicate nevertheless treats the market as being in settlement as soon as `expiry_ts` has elapsed, even while its explicit status remains `ReduceOnly`. It therefore represents both finalized settlement status and time-based expiry.

programs/velocity/src/state/perp_market.rs

```
562  pub fn is_in_settlement(&self, now: i64) -> bool {
563      let in_settlement = matches!(
564          self.status,
565          MarketStatus::Settlement | MarketStatus::Delisted
566      );
567      let expired = self.expiry_ts != 0 && now >= self.expiry_ts;
568      in_settlement || expired
569  }
```

Settlement detection includes both explicit status and elapsed expiry

`place_perp_order` applies this dual predicate and rejects new orders once the timestamp has elapsed. Consequently, a position holder cannot submit a new order to reduce or close the position during the interval between expiry and finalization.

programs/velocity/src/controller/orders.rs

```
1  pub fn place_perp_order([...]) -> VelocityResult<PlaceOrderResult> {
2      [...]
3      validate!(
4          !matches!(market.status, MarketStatus::Initialized),
5          ErrorCode::MarketBeingInitialized,
6          "Market is being initialized"
7      )?;
8      validate!(
9          user.pool_id == 0,
10         ErrorCode::InvalidPoolId,
11         "user pool id ({}) != 0",
12         user.pool_id
13     )?;
14     validate!(
```

```
        !market.is_in_settlement(now),
        ErrorCode::MarketPlaceOrderPaused,
        "Market is in settlement mode",
    )?;
    let position_index = get_position_index(&user.perp_positions, market_index)
        .or_else(|_| add_new_position(&mut user.perp_positions, market_index))?;
    [...]
}
```

Order placement is disabled immediately upon time-based expiry

`fill_perp_order` applies the same time-aware settlement check, preventing existing orders from being filled after expiry. The surrounding `trigger_order` path also uses `is_in_settlement`, so dormant trigger orders cannot be activated as an alternative means of managing the position.

programs/velocity/src/controller/orders.rs

```
pub fn fill_perp_order([...]) -> VelocityResult<(u64, u64)> {
    [...]
    let amm_not_globally_paused: bool = !state.amm_paused()?;
    let mut amm_is_available: bool = amm_not_globally_paused;
    // AMM JIT in a DLOB match: honors the hard gates but, unlike
    // `amm_is_available`, not the auction-timing gates (JIT feeds the auction).
    let amm_jit_allowed: bool;
    {
        let market = &mut perp_market_map.get_ref_mut(&market_index)?;
        validation::perp_market::validate_perp_market(market)?;
        validate!(
            !market.is_in_settlement(now),
            ErrorCode::MarketFillOrderPaused,
            "Market is in settlement mode",
        )?;
        let oracle_price_data = oracle_map.get_price_data(&market.oracle_id())?;
        [...]
    }[...]
}
```

Order filling uses the same time-aware settlement gate

`handle_transfer_perp_position` likewise rejects transfers based on `is_in_settlement` before calculating the transfer at the live oracle. Thus, transferring the position does not remain available as a post-expiry escape path.

programs/velocity/src/instructions/user.rs

```
pub fn handle_transfer_perp_position<'c: 'info, 'info>([...]) ->
anchor_lang::Result<()> {
    [...]
```

```
    {
        [...]
            ErrorCode::InvalidTransferPerpPosition,
            "perp market fills paused"
        )?;
        [...]
        validate!(
            !perp_market.is_in_settlement(now),
            ErrorCode::InvalidTransferPerpPosition,
            "market is in settlement mode"
        )?;
        oracle_price = oracle_price_data.price;
    }
    [...]
}
```

Position transfers are unavailable after expiry

`settle_expired_position` requires the market's explicit status to equal `Settlement` and also enforces the configured post-expiry settlement buffer. Before the administrator finalizes the market and commits `expiry_price`, the holder therefore cannot settle the expired position. `liquidate_perp` does not apply the equivalent time-aware settlement gate. It uses `expiry_price` only when the explicit status is already `Settlement` otherwise, including after `expiry_ts` while the status remains `ReduceOnly`, margin calculations and position transfers are valued using current oracle data. A permissionless liquidator may therefore act in a price domain that every user-directed position-management path already treats as expired.

The separate `settle_expired_market` finalization calculates an expiry price from the market's stored oracle statistics and market balances, then writes `expiry_price` and changes the status to `Settlement`. Until this warm-admin operation executes, the eventual fixed settlement value can differ from the live-oracle value accepted by direct liquidation.

If the live oracle makes the account liquidatable while the subsequently committed expiry price would leave it healthy, liquidation transfers pending settlement value from the frozen holder to the liquidator. FOr example, if a position healthy at an expiry price of 104.999999 is liquidated at a live oracle price of 100, with the configured 1% liquidation discount, the holder loses 5.999999 quote of gross settlement value and the liquidator receives the corresponding claim, approximately 5.9% of position notional after accounting for the ordinary expiry closeout fee. The vulnerable interval lasts from `expiry_ts` until the administrator's finalization transaction executes.

Remediation

In `liquidate_perp`, reject liquidation when `market.is_in_settlement(now)` is true but the market has not yet entered explicit `Settlement` status and committed `expiry_price`. Perform this check before calculating margin or selecting a liquidation price. Once the status is `Settlement`, retain the existing settlement-aware behavior that values the position using `expiry_price`. Apply the same distinction consistently to all perp-aware liquidation entrypoints,

including `liquidate_perp_with_fill`, even though that path currently fails through its internal `place_perp_order` call.

Patch

Resolved in PR#365.

Pyth Ownership Check Accepts Invalid Oracles

ID	Severity	Status
OS-VEP-ADV-40	High	✓ Resolved

Description

`admin::recenter_perp_market_amm_crank` reads the supplied oracle without verifying that its key matches the oracle configured for the target perpetual market. A hot admin may therefore provide any structurally valid Pyth feed using the expected `OracleSource`, including one for an unrelated and differently priced asset. The resulting price becomes the AMM's new `peg_multiplier`, allowing the caller to manipulating the price offered by the vAMM.

programs/velocity/src/vlp/amm/admin.rs

```
pub fn handle_recenter_perp_market_amm_crank(
    ctx: Context<AdminUpdatePerpMarketAmmSummaryStats>,
    depth: Option<u128>,
) -> Result<()> {
    let perp_market = &mut load_mut!(ctx.accounts.perp_market)?;

    let clock = Clock::get()?;
    let price_oracle = &ctx.accounts.oracle;

    let OraclePriceData {
        price: oracle_price,
        ..
    } = get_oracle_price(&perp_market.oracle_source, price_oracle, clock.slot)?;

    msg!(
        "recentering amm crank for perp market {}",
        perp_market.market_index
    );
    [...]
}
```

Remediation

Aplly the `valid_oracle_for_perp_market` check that other methods use.

Patch

Resolved in PR#440.

Direct Deposit Ignores the Market-Scoped Deposit Pause Bit

ID	Severity	Status
OS-VEP-ADV-41	Medium	Resolved

Description

`handle_deposit` is protected by the exchange-wide deposit pause guard, so the direct user deposit path rejects deposits only when `ExchangeStatus::DepositPaused` is set at the global state level. The surrounding implementation then loads the requested spot market and continues with market-specific accounting, but this entry-point guard does not evaluate the spot market's own paused operation bitmap, as shown in `handle_deposit`.

programs/velocity/src/instructions/user.rs

```
#[access_control(
    deposit_not_paused(&ctx.accounts.state)
)]
pub fn handle_deposit<'c: 'info, 'info>(
    ctx: Context<'info, Deposit<'info>>,
    market_index: u16,
    amount: u64,
    [...]
}
```

After loading the spot market and updating cumulative interest, `handle_deposit` credits the user's spot position through the spot-balance controller and only checks that a positive deposit position is in an `Active` market. This status check can reject initialized or otherwise inactive markets, but it is separate from the per-market `SpotOperation::Deposit` bit exposed by `SpotMarket::is_operation_paused`, so a market can be active while its deposit operation is paused.

programs/velocity/src/instructions/user.rs

```
pub fn handle_deposit<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    let total_withdraws_after = user.total_withdraws;

    let spot_position = &mut user.spot_positions[position_index];
    controller::spot_position::update_spot_balances_and_cumulative_deposits([...])?;

    let token_amount = spot_position.get_token_amount(&spot_market)?;
    if token_amount == 0 {
        validate!(
            spot_position.scaled_balance == 0,
            ErrorCode::InvalidSpotPosition,
```

```
            "deposit left user with invalid position. scaled balance = {} token amount =
            {}",
            spot_position.scaled_balance,
            token_amount
        )?;
    }

    if spot_position.balance_type == SpotBalanceType::Deposit &&
    spot_position.scaled_balance > 0 {
        validate!(
            matches!(spot_market.status, MarketStatus::Active),
            ErrorCode::MarketActionPaused,
            "spot_market not active",
        )?;
    }

    [...]
}
```

The function later receives tokens into the spot market vault, emits a deposit record, and validates the aggregate token deposit and borrow limits. These checks happen without an intervening `SpotOperation::Deposit` pause check, so the direct deposit flow can complete when global deposits are unpaused, the market status is `Active`, and the market cap is not exceeded.

programs/velocity/src/instructions/user.rs

```
pub fn handle_deposit<'c: 'info, 'info>([...]) -> Result<()> {
    [...]

    let spot_market = &mut spot_market_map.get_ref_mut(&market_index)?;

    controller::token::receive([...])?;
    ctx.accounts.spot_market_vault.reload()?;

    let deposit_record_id = get_then_update_id!(spot_market, next_deposit_record_id);
    let oracle_price = oracle_price_data.price;
    let explanation = if is_borrow_before {
        DepositExplanation::RepayBorrow
    } else {
        DepositExplanation::None
    };
    let signer = if ctx.accounts.authority.key() != user.authority
        && ctx.accounts.authority.key() != user.delegate
    {
        #[cfg(feature = "mainnet-beta")]
        validate!(
```

```
            WHITELISTED_EXTERNAL_DEPOSITORS.contains(&ctx.accounts.authority.key()),
            ErrorCode::DefaultError,
            "Not whitelisted external depositor"
        )?;

        Some(ctx.accounts.authority.key())
    } else {
        None
    };
    let user_token_amount_after = user.get_total_token_amount(spot_market)?;
    let deposit_record = DepositRecord {[...]};
    emit!(deposit_record);

    spot_market.validate_max_token_deposits_and_borrows(false)?;

    Ok(())
}
```

As a result, a market-scoped deposit circuit breaker does not stop deposits through the main public deposit handler. Operators can set the market's deposit pause bit during risk containment, but users may still add deposits to that market through `handle_deposit` under the conditions above, weakening the intended market-specific exposure control.

Remediation

Before updating the user's spot balance or receiving tokens, make `handle_deposit` reject when `spot_market.is_operation_paused(SpotOperation::Deposit)` is true, using the existing market action error pattern.

Patch

Resolved in PR#137.

Same-Market transfer-deposit Bypasses Recipient Deposit Admission

ID	Severity	Status
OS-VEP-ADV-42	Medium	Resolved

Description

Direct deposits into a spot market perform admission checks after updating the user's deposit balance. `handle_deposit` checks that a positive resulting deposit position is only accepted when the spot market is active, and the rendered excerpt shows the same direct-deposit path finishing by validating market-level deposit and borrow limits before returning.

The market-level validator used by direct deposit calculates the current aggregate deposits and compares them against `max_token_deposits`, unless that cap is disabled. This makes the cap an explicit admission condition for paths that call `validate_max_token_deposits_and_borrows(false)` after increasing deposit balances.

programs/velocity/src/state/spot_market.rs

```
pub fn validate_max_token_deposits_and_borrows(
    &self,
    do_max_borrow_check: bool,
) -> VelocityResult {
    let deposits = self.get_deposits()?;
    let max_token_deposits = self.max_token_deposits.cast::<u128>()?;
    validate!(
        max_token_deposits == 0 || deposits <= max_token_deposits,
        ErrorCode::MaxDeposit,
        "max token amount ({}) < deposits ({})",
        max_token_deposits,
        deposits,
    )?;
    [...]
}
```

Spot market max-deposit validator

`handle_transfer_deposit_by_delegate` debits the source user by updating the source balance with the limit-aware helper and then checks withdrawal margin and spot margin trading. However, unlike the direct withdraw path read in `handle_withdraw`, this transfer debit does not first apply the reduce-only capping that restricts withdrawals in a reduce-only market to the user's existing deposit and maximum withdrawable amount.

The recipient side of `handle_transfer_deposit_by_delegate` then increments the recipient's total deposits and credits the recipient spot position with the raw balance update helper. The surrounding transfer logic checks the recipient pool id and validates the zero-token edge case, but it does not apply the direct-deposit active-market admission check before accepting a positive deposit balance.

After crediting the recipient and emitting the transfer deposit record, the same-market transfer path updates the recipient activity slot and only validates the vault amount before returning. Because the path does not call the same max-deposit validator used by direct deposit after increasing the recipient deposit balance, users may move existing exposure between same-authority subaccounts into a market where a direct deposit would be rejected by active/status gates or the aggregate deposit cap, and the source debit may create a source-side borrow in a reduce-only market that direct withdraw would cap.

programs/velocity/src/instructions/user.rs

```rust
pub fn handle_transfer_deposit_by_delegate<'c: 'info, 'info>([...]) ->
anchor_lang::Result<()> {
    [...]
    to_user.update_last_active_slot(slot);
    let spot_market = spot_market_map.get_ref(&market_index)?;
    math::spot_withdraw::validate_spot_market_vault_amount(
        &spot_market,
        ctx.accounts.spot_market_vault.amount,
    )?;
    Ok(())
}
```

Transfer path ends with vault amount validation only

Remediation

Before debiting the source, apply direct-withdraw's reduce-only capping so an internal transfer can't create a borrow in a reduce-only market. After crediting the recipient, apply direct-deposit admission: active spot-market status for positive deposit balances and `validate_max_token_deposits_and_borrows(false)`.

Patch

Resolved in PR#139.

`liquidate_spot_with_swap` Bricked by Stale Account-Index Guard

ID	Severity	Status
OS-VEP-ADV-43	Medium	Resolved

Description

`handle_liquidate_spot_with_swap_begin` performs instruction introspection after initiating the swap-backed liquidation flow: it scans subsequent transaction instructions, requires the next Velocity instruction to be `LiquidateSpotWithSwapEnd`, and then validates that selected begin accounts match the end instruction. Those validations still use hard-coded end-instruction positions and treat the fixed account prefix as 13 accounts before comparing `ctx.remaining_accounts` against the end instruction accounts.

The current `LiquidateSpotWithSwap` account context used by both begin and end contains 11 fixed accounts, from `state` through the instructions sysvar, as shown in the account structure. With remaining accounts beginning after that 11-account prefix, the begin handler's `ix.accounts.len() - 13` comparison is offset from the current layout and cannot be satisfied for a correctly generated begin/end pair with the same remaining accounts.

programs/velocity/src/instructions/keeper.rs

```
pub struct LiquidateSpotWithSwap<'info> {
    pub state: AccountLoader<'info, State>,
    pub authority: Signer<'info>,
    #[account(
        mut,
        constraint = can_sign_for_user(&liquidator, &authority)?
    )]
    pub liquidator: AccountLoader<'info, User>,
    #[account(mut)]
    pub user: AccountLoader<'info, User>,
    #[account(
        mut,
        seeds = [b"spot_market_vault".as_ref(),
        liability_market_index.to_le_bytes().as_ref()],
        bump,
    )]
    pub liability_spot_market_vault: Box<InterfaceAccount<'info, TokenAccount>>,
    #[account(
        mut,
        seeds = [b"spot_market_vault".as_ref(),
        asset_market_index.to_le_bytes().as_ref()],
        bump,
    )]
    pub asset_spot_market_vault: Box<InterfaceAccount<'info, TokenAccount>>,
    #[account(
        mut,
```

```
        constraint =
        &liability_spot_market_vault.mint.eq(&liability_token_account.mint),
        token::authority = authority
    )]
    pub liability_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
    #[account(
        mut,
        constraint = &asset_spot_market_vault.mint.eq(&asset_token_account.mint),
        token::authority = authority
    )]
    pub asset_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
    pub token_program: Interface<'info, TokenInterface>,
    #[account(
        constraint = state.load()?.signer.eq(&velocity_signer.key())
    )]
    /// CHECK: forced velocity_signer
    pub velocity_signer: UncheckedAccount<'info>,
    /// Instructions Sysvar for instruction introspection
    /// CHECK: fixed instructions sysvar account
    #[account(address = instructions::ID)]
    pub instructions: UncheckedAccount<'info>,
}
```

Current `LiquidateSpotWithSwap` fixed account layout.

Because the begin handler rejects the matching end instruction before the transaction can complete, the atomic `liquidate_spot_with_swap` route becomes unusable under the current account layout. The direct `liquidate_spot` entrypoint remains separate, but it forces the liquidator to take over the borrow and satisfy margin; under stressed or long-tail collateral/liability combinations, losing the swap route reduces liquidation coverage and raises the chance that unhealthy accounts slide into bankruptcy/bad debt.

Remediation

Replace the hard-coded indexes and fixed-account count with the current LiquidateSpotWithSwap layout: fixed accounts start at 0 and remaining accounts start at 11.

Patch

Resolved in PR#134.

Reduce-Only Maker Order Can Overfill When AMM Fulfillment Is Paused

ID	Severity	Status
OS-VEP-ADV-44	Medium	Resolved

Description

Reduce-only orders are intended to reduce an existing position rather than create new opposite-side exposure. The position-aware sizing logic in `get_base_asset_amount_unfilled` supports that invariant by returning the raw remaining order size only when no existing position is supplied, and otherwise capping reduce-only capacity to the current opposing position or zero when the order would increase exposure.

The reduce-only cancellation check uses that same position-aware helper, but only to decide whether an open reduce-only order should be canceled because its capped remaining size is below the market step size. This means maker discovery can avoid keeping fully unfillable reduce-only orders alive, but the check does not itself carry the capped size forward as the amount available for matching.

programs/velocity/src/math/orders.rs

```rust
pub fn should_cancel_reduce_only_order(
    order: &Order,
    existing_base_asset_amount: i64,
    step_size: u64,
) -> VelocityResult<bool> {
    let should_cancel = order.status == OrderStatus::Open
        && order.reduce_only
        && order.get_base_asset_amount_unfilled(Some(existing_base_asset_amount))? < step_size;
    Ok(should_cancel)
}
```

`should_cancel_reduce_only_order` uses the capped size only as a cancellation predicate

The DLOB maker quoter then exposes the order to the matcher using raw remaining size. In `DlobOrderQuoter`, `level_capacity` delegates to the quoter's raw remaining amount, so a stale or oversized reduce-only maker order can advertise more maker liquidity than the maker's current position can safely reduce.

programs/velocity/src/state/quoter.rs

```rust
impl<'a> DlobOrderQuoter<'a> {
    [...]
    fn remaining(&self) -> u64 {
        self.order
            .base_asset_amount
            .saturating_sub(self.order.base_asset_amount_filled)
```

```
    }
}

impl<'a> Quoter for DlobOrderQuoter<'a> {
    fn best_price(&self, ctx: &QuoteContext, side: PositionDirection) ->
    VelocityResult<u64> {
        if !self.quotes_on(side) {
            return Ok(Self::no_quote(side));
        }
        Ok(self.effective_price(ctx)?.unwrap_or(Self::no_quote(side)))
    }

    fn level_capacity(&self, ctx: &QuoteContext, side: PositionDirection) ->
    VelocityResult<u64> {
        if !self.quotes_on(side) || self.effective_price(ctx)?.is_none() {
            return Ok(0);
        }
        Ok(self.remaining())
    }
```

`DlobOrderQuoter::level_capacity` exposes raw remaining DLOB order size

In the perp match path, the controller computes `maker_unfilled` with the maker's current perp position and passes that capped value into `AmmJitQuoter::from_match_context`. However, it separately constructs `DlobOrderQuoter` directly from the maker order and passes both quoters to `match_take` the DLOB quoter is not given the maker-position context that produced `maker_unfilled`. Since `match_take` materializes each maker's fillable level from `level_capacity`, the raw DLOB capacity can be allocated to the reduce-only maker order when AMM fulfillment is paused or otherwise unavailable to absorb the capped amount.

As a result, a public taker can fill an oversized reduce-only maker order past the maker's current position, flipping a reduce-only order into a position-increasing one.

Remediation

Carry the maker's reduce-only-aware max fill size into matching: DLOB maker capacity should use `get_base_asset_amount_unfilled(Some(maker_position.base_asset_amount))`, or store a capped maker size that `DlobOrderQuoter` respects.

Patch

Resolved in PR#241.

ReduceOnly Markets Can Still Fill Legacy Non-Reduce-Only Perp Orders

ID	Severity	Status
OS-VEP-ADV-45	Medium	Resolved

Description

`MarketStatus::ReduceOnly` is documented as a market state where fills are only able to reduce liability. This makes the market status itself part of the risk-control semantics for wind-down, delist, or incident-response conditions, as shown by the `MarketStatus` enum.

programs/velocity/src/state/market_status.rs

```
4  pub enum MarketStatus {
5      /// warm up period for initialization, fills are paused
6      #[default]
7      Initialized,
8      /// all operations allowed
9      Active,
10     /// fills only able to reduce liability
11     ReduceOnly,
12     /// market has determined settlement price and positions are expired must be settled
13     Settlement,
14     /// market has no remaining participants
15     Delisted,
16 }
```

`ReduceOnly` documented as allowing only liability-reducing fills

The `fill_perp_order` path does not enforce that semantic for every order at fill time. After loading the taker order and settling funding, it accepts both `Active` and `ReduceOnly` perp markets before continuing through order validation, maker discovery, reduce-only cancellation checks, and fulfillment.

programs/velocity/src/controller/orders.rs

```
1  pub fn fill_perp_order([...]) -> VelocityResult<(u64, u64)> {
2      [...]
3      validate!(
4          matches!(
5              market.status,
6              MarketStatus::Active | MarketStatus::ReduceOnly
7          ),
8          ErrorCode::MarketFillOrderPaused,
9          "Market not active",
10     )?;
11     [...]
12 }
```

`fill_perp_order` accepts reduce-only markets

The reduce-only cancellation path used around fulfillment is keyed to the stored order flag. `fill_perp_order` calls this logic before fulfillment for the taker and again after the fill, and maker-side handling uses the same helper, but the helper only considers an order reduce-only when the order itself has `reduce_only` set.

programs/velocity/src/math/orders.rs

```
pub fn should_cancel_reduce_only_order(
    order: &Order,
    existing_base_asset_amount: i64,
    step_size: u64,
) -> VelocityResult<bool> {
    let should_cancel = order.status == OrderStatus::Open
        && order.reduce_only
        && order.get_base_asset_amount_unfilled(Some(existing_base_asset_amount))? < step_size;

    Ok(should_cancel)
}
```

Cancellation depends on the stored order flag

The position-aware unfilled-size calculation reinforces that behavior. When an order is not marked reduce-only, `get_base_asset_amount_unfilled` returns the full unfilled amount even when an existing position is supplied, so the clipping logic that would prevent risk-increasing fills is skipped for legacy non-reduce-only orders.

As a result, a perp order placed while the market was `Active` can retain `reduce_only = false` after the market is later moved to `ReduceOnly`. Because the fill path permits `ReduceOnly` markets and the liability-reducing checks depend on the stored order flag rather than the current market status, a public filler can still fill such legacy orders in directions that increase user exposure and aggregate open interest, undermining the intended circuit breaker.

Remediation

At fill time, treat `market.status == ReduceOnly` as a forced reduce-only condition for all existing orders: clip/cancel legacy orders that would increase exposure.

Patch

Resolved in PR#274.

Unbounded Order Expiry Overflows AMM Fallback Pricing And Aborts Fills

ID	Severity	Status
OS-VEP-ADV-46	Medium	Resolved

Description

Perp order placement accepts a caller-provided `max_ts` and, when it is present, uses it as the order expiry timestamp. In `place_perp_order`, the surrounding placement flow computes auction parameters, derives `max_ts` either from `params.max_ts` or from a short default for market and oracle orders, and then only rejects timestamps that are already in the past. This leaves future values unbounded, including extremely large `i64` values.

programs/velocity/src/controller/orders.rs

```rust
pub fn place_perp_order([...]) -> VelocityResult {
    [...]
    let max_ts = match params.max_ts {
        Some(max_ts) => max_ts,
        None => match params.order_type {
            OrderType::Market | OrderType::Oracle => now.safe_add(
                30_i64.max(
                    (auction_duration.safe_div(2)?)
                        .cast::<i64>()?
                        .safe_add(10_i64)?,
                ),
            )?,
            _ => 0_i64,
        },
    };
    if max_ts != 0 && max_ts < now {
        msg!("max_ts ({}) < now ({}), skipping order", max_ts, now);
        return Ok(());
    }
    [...]
}
```

Perp order placement only rejects past expiries.

During maker-match fulfillment, the fallback taker price is computed when no explicit taker limit price is available. In that branch, the fill logic reads the taker order's remaining time through `seconds_til_expiry(now)` and passes it directly into AMM fallback pricing. The `Order` helper returns the nonnegative difference between the order's stored `max_ts` and `now`, so an order placed with a very large future expiry can carry a very large seconds-to-expiry value into this path.

The AMM `get_fallback_price` routine uses `seconds_til_order_expiry` to derive price buffers for both long and short directions. The relevant arithmetic multiplies the seconds-to-expiry value before applying the divisor clamp, so a sufficiently large expiry-derived value can overflow before

the clamp reduces it. An order with `max_ts` set to `i64::MAX` can therefore abort fallback price calculation for fills routed through this maker-match path.

Thus, a fill that includes the poisoned order may abort before the matching logic can complete or skip the order. If such an order has favorable matching priority, it could interfere with DLOB matching and degrade execution for affected fills.

Remediation

Cap user-supplied `max_ts` for market/oracle orders to a protocol-defined max TTL, and compute the fallback divisors with safe/saturating arithmetic (or clamp seconds before multiplying).

Patch

Resolved in PR#274.

Trigger-Order Path Uses The Exchange-Pause Guard Instead Of The Fill-Pause Guard

ID	Severity	Status
OS-VEP-ADV-47	Medium	Resolved

Description

`handle_trigger_order` is guarded by `exchange_not_paused` before it loads market and oracle account maps and calls `controller::orders::trigger_order`, as shown in the trigger-order instruction handler. This means the trigger-order path is not using the same pause condition as the public fill path even though it mutates order execution state.

programs/velocity/src/instructions/keeper.rs

```
#[access_control(
    exchange_not_paused(&ctx.accounts.state)
)]
pub fn handle_trigger_order<'c: 'info, 'info>(
    ctx: Context<'info, TriggerOrder<'info>>,
    order_id: u32,
) -> Result<()> {[...]}
```

`handle_trigger_order` uses the exchange pause guard.

The adjacent fill instruction uses `fill_not_paused` for `handle_fill_perp_order`, which makes public fill execution subject to the `FillPaused` bit. The mismatch allows a standalone `FillPaused` state to block normal fills while leaving trigger-order execution reachable through the keeper trigger path.

programs/velocity/src/instructions/keeper.rs

```
#[access_control(
    fill_not_paused(&ctx.accounts.state)
)]
pub fn handle_fill_perp_order<'c: 'info, 'info>(
    ctx: Context<'info, FillOrder<'info>>,
    order_id: Option<u32>,
) -> Result<()> {[...]}
```

`handle_fill_perp_order` uses the fill pause guard.

`exchange_not_paused` only returns an exchange pause error when the exchange status reports `is_all()`. Because this guard does not check whether `FillPaused` is contained in the status flags, setting only the fill pause bit does not stop `handle_trigger_order`.

programs/velocity/src/instructions/constraints.rs

```rust
pub fn exchange_not_paused(state: &AccountLoader<State>) -> anchor_lang::Result<()>
{
    let state = state.load()?;
    if state.get_exchange_status()?.is_all() {
        return Err(ErrorCode::ExchangePaused.into());
    }
    Ok(())
}
```

exchange_not_paused only rejects the all-paused state.

Once reached, `trigger_order` validates that the order is open and triggerable, checks the trigger condition, then updates trigger-order auction parameters, increments open-auction accounting when an auction exists, and increases open bids or asks for the user's perp position. Those mutations can occur during a `FillPaused` window even though fills are intended to be frozen.

programs/velocity/src/controller/orders.rs

```rust
pub fn trigger_order([...]) -> VelocityResult {
    [...]
    {
        [...]
        // state (refreshed by the keeper crank / fill setup for this slot).
        update_trigger_order_params([...])?;

        if user.orders[order_index].has_auction() {
            user.increment_open_auctions();
        }

        let direction = user.orders[order_index].direction;
        let base_asset_amount = user.orders[order_index].base_asset_amount;
        let update_open_bids_and_asks =
        user.orders[order_index].update_open_bids_and_asks();

        let user_position = user.get_perp_position_mut(market_index)?;
        increase_open_bids_and_asks([...])?;
        if user_position.is_isolated() {[...]}
    }
    [...]
}
```

trigger_order initializes auction and open-order accounting.

As a result, `FillPaused` does not fully preserve the intended execution window for trigger orders. A caller can crystallize the trigger and start the auction during the pause; when fills are later

enabled, execution may occur against auction parameters that were initialized earlier during the incident window rather than from a fresh post-pause trigger state.

Remediation

Gate `handle_trigger_order` with `fill_not_paused` instead of `exchange_not_paused`, or otherwise defer the auction-parameter initialization and open-auction/open-bids-or-asks mutations until fill execution is enabled.

Patch

Resolved in PR#270.

Swap-Backed Spot Liquidation Bypasses The Max-Pct Liquidation Throttle

ID	Severity	Status
OS-VEP-ADV-48	Medium	Resolved

Description

Direct spot liquidation applies the time-ramped liquidation percentage as an actual transfer cap. In `liquidate_spot`, the controller computes `max_pct_allowed` from the user's liquidation state and margin shortage, derives `max_liability_allowed_to_be_transferred`, and includes that capped amount in the chain that selects the final `liability_transfer`.

The swap-backed begin path computes the same capped liability amount from `calculate_max_pct_to_liquidate`, but the surrounding flow only treats the result as a liveness condition. In `liquidate_spot_with_swap_begin`, a zero capped amount rejects the liquidation, while a nonzero capped amount does not itself limit the later swap sizing.

programs/velocity/src/controller/liquidation.rs

```
pub fn liquidate_spot_with_swap_begin([...]) -> VelocityResult {
    [...]
    let max_pct_allowed = calculate_max_pct_to_liquidate(
        user,
        margin_shortage,
        slot,
        initial_pct_to_liquidate,
        liquidation_duration,
    )?;
    let max_liability_allowed_to_be_transferred =
    liability_transfer_to_cover_margin_shortage
        .saturating_mul(max_pct_allowed)
        .safe_div(LIQUIDATION_PCT_PRECISION)?;

    if max_liability_allowed_to_be_transferred == 0 {
        msg!("max_liability_allowed_to_be_transferred == 0");
        return Err(ErrorCode::InvalidLiquidation);
    }
    [...]
}
```

Swap begin computes the pct-capped liability amount but only checks that it is nonzero.

After that check, `liquidate_spot_with_swap_begin` derives the maximum asset input from `liability_transfer_to_cover_margin_shortage`, which is the full margin-shortage coverage amount rather than the pct-capped liability amount. The resulting `max_asset_transfer` is then used to validate `swap_amount_in`, so the begin phase can permit a swap input sized against the uncapped shortage.

The end phase does not close this gap. `liquidate_spot_with_swap_end` receives the realized `asset_transfer` and `liability_transfer` from the instruction handler and validates the swap against liquidation price boundaries, but the cited validation does not recompute `calculate_max_pct_to_liquidate` or compare the transfer amounts against the time-ramped pct cap.

programs/velocity/src/controller/liquidation.rs

```
pub fn liquidate_spot_with_swap_end([...]) -> VelocityResult {
    [...]
    validate_swap_within_liquidation_boundaries(
        asset_transfer,
        liability_transfer,
        asset_decimals,
        liability_decimals,
        asset_price,
        liability_price,
        asset_liquidation_multiplier,
        liability_liquidation_multiplier,
    )?;
    [...]
}
```

Swap end validates the swap price boundary without enforcing the pct cap.

As a result, once the swap-backed lane is otherwise reachable, a liquidator may be able to route through `liquidate_spot_with_swap` and seize collateral based on the full cover-shortage amount even when the liquidation-duration policy would currently allow only a smaller fraction, causing premature full/near-full spot liquidation and defeating the liquidation-duration policy.

Remediation

In `liquidate_spot_with_swap_begin`, size the permitted asset input from the pct-capped liability amount, including the intended buffer and dust handling, rather than from `liability_transfer_to_cover_margin_shortage`. In `liquidate_spot_with_swap_end`, recompute the current liquidation state and reject any realized liability or asset transfer that exceeds the pct-capped allowance after applying the same dust rules used by direct spot liquidation.

Patch

Resolved in PR#408.

PnL-vs-Spot Liquidations Skip the 5-Minute Spot Oracle TWAP Band

ID	Severity	Status
OS-VEP-ADV-49	Medium	Resolved

Description

The shared spot-market refresh path updates cumulative interest, skips further validation for the quote market, computes spot oracle validity against the one-hour risk EMA, and checks whether that validity is acceptable for the requested `VelocityAction`. The `update_spot_market_and_check_validity` helper does not perform a 5-minute oracle-vs-TWAP divergence check or return `PriceBandsBreached` when the current spot oracle is outside that shorter band.

Direct spot liquidation applies an additional price-band guard after calculating the spot transfer amounts and before validating the limit price or moving balances. In `liquidate_spot`, both the liability and asset spot oracle prices are compared against each market's `last_oracle_price_twap_5min`, and liquidation reverts with `PriceBandsBreached` when either oracle is too divergent.

The PnL-for-borrow route relies on the shared spot validity helper for the liability spot market, then uses the resulting `liability_price` to derive `liability_transfer_implied_by_pnl`. `liquidate_borrow_for_perp_pnl` sizes the path using the spot oracle price for the borrow market without the 5-minute TWAP band check that `liquidate_spot` applies before comparable spot-priced transfers.

The deposit-for-negative-PnL route follows the same pattern for the asset spot market. `liquidate_perp_pnl_for_deposit` sizes the spot deposit transfer with no explicit `is_oracle_too_divergent_with_twap_5min` check on the asset oracle.

As a result, PnL-vs-spot liquidation conversions may be priced using a spot oracle state that the protocol rejects for direct spot liquidation. When the oracle is valid for the general liquidation action but outside the 5-minute TWAP band, the affected routes can size transfers at the divergent oracle price, which may produce unfair liquidation sizing such as excess collateral seized for negative PnL or too little liability assumed for positive PnL.

Remediation

After `update_spot_market_and_check_validity`, run `is_oracle_too_divergent_with_twap_5min` for every non-quote spot market whose oracle sizes a PnL liquidation transfer

Patch

Resolved in PR#391.

Spot Bankruptcy Clears the Borrow Using Stale Cumulative Borrow Interest

ID	Severity	Status
OS-VEP-ADV-50	Medium	Resolved

Description

`resolve_spot_bankruptcy` computes the bankrupt user's spot debt from the user's scaled borrow balance before refreshing the spot market's cumulative borrow interest. The resulting `borrow_amount` is then used to determine the insurance fund payment, the loss to socialize, the cumulative deposit interest reduction, and the amount used to clear the user's spot position. When the market's cumulative borrow interest is stale, this sequence resolves the bankruptcy using a lower token amount than the current borrow value.

Spot borrow balances are interest-scaled, so the token amount is only current if the market's cumulative interest fields are current. The spot balance conversion shown in `get_token_amount` multiplies a borrow balance by `cumulative_borrow_interest`, which means any unrefreshed accrued borrow interest is excluded from the token amount returned to the bankruptcy controller.

programs/velocity/src/math/spot_balance.rs

```rust
pub fn get_token_amount(
    balance: u128,
    spot_market: &SpotMarket,
    balance_type: &SpotBalanceType,
) -> VelocityResult<u128> {
    let precision_decrease = 10_u128.pow(19_u32.safe_sub(spot_market.decimals)?);

    let cumulative_interest = match balance_type {
        SpotBalanceType::Deposit => spot_market.cumulative_deposit_interest,
        SpotBalanceType::Borrow => spot_market.cumulative_borrow_interest,
    };

    let token_amount = match balance_type {
        SpotBalanceType::Deposit => balance
            .safe_mul(cumulative_interest)?
            .safe_div(precision_decrease)?,
        SpotBalanceType::Borrow => balance
            .safe_mul(cumulative_interest)?
            .safe_div_ceil(precision_decrease)?,
    };

    Ok(token_amount)
}
```

Borrow token amounts depend on cumulative borrow interest.

The public keeper path does not unconditionally refresh cumulative spot interest before calling the liquidation controller. In `handle_resolve_spot_bankruptcy`, the handler first attempts revenue settlement and reloads vault balances, then passes the insurance fund vault amount into `resolve_spot_bankruptcy` there is no separate mandatory interest refresh in the shown pre-call path.

The attempted revenue settlement path only reaches the interest-refreshing settlement logic when the revenue settlement timer is due. In `attempt_settle_revenue_to_insurance_fund`, the call to `settle_revenue_to_insurance_fund` is gated by `valid_revenue_settle_time` otherwise the function returns without settlement.

As a result, if borrow interest has accrued since the last market interest update and the revenue-settle timer is not due, spot bankruptcy can clear the user's full scaled borrow at the stale token amount. The delta is the accrued borrow interest between the stale market state and a current update; based on the accounting shown in the controller, this may reduce the insurance fund draw, reduce depositor socialization, reduce protocol or insurance-fund interest carveouts associated with the missing interest update, and understate the bankrupt user's recorded socialized loss.

Remediation

Refresh the affected spot market's cumulative interest before `resolve_spot_bankruptcy` computes `borrow_amount`, preferably inside the controller immediately before reading the user's spot position token amount so every caller observes the same accounting. Alternatively, make `handle_resolve_spot_bankruptcy` unconditionally call `update_spot_market_cumulative_interest` for the target spot market before invoking the controller.

Patch

Resolved in PR#273.

Perp PnL Deficit Recap Draws From the Insurance Fund on a Stale PnL Pool

ID	Severity	Status
OS-VEP-ADV-51	Medium	Resolved

Description

`resolve_perp_pnl_deficit` decides whether the perp market's PnL pool is insufficient before it refreshes quote spot market interest. The PnL pool is represented as a spot deposit balance, and the repository's spot balance math derives its token amount from the pool's scaled balance and the spot market's cumulative deposit interest. As shown in `resolve_perp_pnl_deficit`, the function computes the pool token amount, compares it with net user PnL, and only then updates spot market cumulative interest.

programs/velocity/src/controller/insurance.rs

```rust
pub fn resolve_perp_pnl_deficit([...]) -> VelocityResult<u64> {
    [...]
    let pnl_pool_token_amount = get_token_amount(
        market.pnl_pool.scaled_balance,
        spot_market,
        &SpotBalanceType::Deposit,
    )?;

    let net_user_pnl = calculate_net_user_pnl(
        &market.amm,
        market
            .market_stats
            .historical_oracle_data
            .last_oracle_price_twap_5min,
        market.quote_asset_amount,
        market.net_unsettled_funding_pnl,
    )?;
    validate!(
        pnl_pool_token_amount.cast::<i128>()? < net_user_pnl,
        ErrorCode::SufficientPerpPnlPool,
        "pnl_pool_token_amount >= net_user_pnl ({} >= {})",
        pnl_pool_token_amount,
        net_user_pnl
    )?;
    update_spot_market_cumulative_interest(spot_market, None, now)?;
    [...]
}
```

Stale PnL pool sufficiency check before interest refresh

After that refresh, `resolve_perp_pnl_deficit` continues into the insurance-fund draw path without recomputing the PnL pool token amount or repeating the sufficiency check. The amount

drawn from insurance is capped by the excess user PnL imbalance, the per-period revenue-withdraw limit, the market's remaining quote insurance allowance, and the insurance vault amount minus one, then credited to the AMM and deposited into the market PnL pool.

The instruction handler uses the controller result as the amount to pay from the insurance fund. In `handle_resolve_perp_pnl_deficit`, a positive `pay_from_insurance` causes the program to transfer tokens from the insurance fund vault to the spot market vault, so a stale-low sufficiency decision in the controller becomes an actual insurance-fund vault transfer.

`handle_resolve_perp_pnl_deficit` attempts to settle revenue to the insurance fund before calling the deficit resolver, but `attempt_settle_revenue_to_insurance_fund` only performs the revenue settlement path when the revenue-settle timer is due. When that timer is not due, this pre-step does not necessarily refresh the quote market interest before the PnL pool sufficiency gate is evaluated.

programs/velocity/src/controller/insurance.rs

```
pub fn attempt_settle_revenue_to_insurance_fund<'info>([...]) -> Result<()> {
    let valid_revenue_settle_time = if spot_market.insurance_fund.revenue_settle_period
    > 0 {
        let time_until_next_update = on_the_hour_update(
            now,
            spot_market.insurance_fund.last_revenue_settle_ts,
            spot_market.insurance_fund.revenue_settle_period,
        )?;

        time_until_next_update == 0
    } else {
        false
    };

    let _token_amount = if valid_revenue_settle_time {
        // uses proportion of revenue pool allocated to insurance fund
        let spot_market_vault_amount = spot_market_vault.amount;
        let insurance_fund_vault_amount = insurance_fund_vault.amount;

        let token_amount = settle_revenue_to_insurance_fund(
            spot_market_vault_amount,
            insurance_fund_vault_amount,
            spot_market,
            now,
            false,
        )?;[...]
    } else {0};
    Ok(())
}
```

Revenue settlement only runs when the timer is due

As a result, a caller of the public `resolve_perp_pnl_deficit` instruction may cause quote insurance-fund capital to be moved into a perp market even when the PnL pool would satisfy the controller's own sufficiency condition after current interest is applied. The impact is bounded by the same limits enforced in the draw path: `excess_user_pnl_imbalance`, the remaining per-period revenue-withdraw capacity, remaining `quote_max_insurance`, and `insurance_vault_amount - 1`.

Remediation

Refresh the quote spot market cumulative interest before deriving `pnl_pool_token_amount` and applying the PnL pool sufficiency gate, or recompute and revalidate that gate immediately after the refresh and before calculating `insurance_withdraw`.

Patch

Resolved in PR#273.

Transfer Pools Bypasses Destination Market Deposit Caps And Status Gate

ID	Severity	Status
OS-VEP-ADV-52	Medium	Resolved

Description

The direct `handle_deposit` path establishes the intended admission controls for increasing a user's deposit balance in a spot market. After updating the user's spot position, it requires the resulting deposit position to be in an `Active` market, and the same flow later calls the market-level deposit cap validator before returning.

The market-level validator compares current market deposits against `max_token_deposits`, and optionally checks borrows against the configured borrow fraction. This is the cap enforcement reached by the normal deposit path, but it is not reached by the destination-credit path in `handle_transfer_pools`.

programs/velocity/src/state/spot_market.rs

```
pub fn validate_max_token_deposits_and_borrows(
    &self,
    do_max_borrow_check: bool,
) -> VelocityResult {
    let deposits = self.get_deposits()?;
    let max_token_deposits = self.max_token_deposits.cast::<u128>()?;
    validate!(
        max_token_deposits == 0 || deposits <= max_token_deposits,
        ErrorCode::MaxDeposit,
        "max token amount ({}) < deposits ({})",
        max_token_deposits,
        deposits,
    )?;
    [...]
}
```

Market validator enforces `max_token_deposits` and optional borrow limits.

In `handle_transfer_pools`, the deposit transfer branch first checks that the source and destination markets use matching mints and compatible pool relationships, then debits the source market and credits the destination market through `update_spot_balances_and_cumulative_deposits_with_limits`. The destination credit is performed with `SpotBalanceType::Deposit`, but the surrounding transfer logic does not apply the direct-deposit `Active` status gate or call `validate_max_token_deposits_and_borrows` for the destination market.

The shared helper used by `handle_transfer_pools` updates the spot balances and then applies withdrawal-oriented checks, and status checks. It does not enforce the stricter deposit-only

`Active` gate and does not validate the destination market's deposit cap, so a transfer can increase destination `deposit_balance` under conditions where a direct deposit would be rejected.

As a result, an ordinary user with two same-authority subaccounts can move a same-mint deposit into a destination market that is `ReduceOnly` or `Settlement`, or already at `max_token_deposits`, bypassing the per-market deposit/status exposure controls described in the finding. The transfer conserves vault tokens, so the issue is not an unbacked-token mint, but it can still mutate destination market aggregate deposits outside the intended admission checks.

Remediation

After every destination-side balance increase in `handle_transfer_pools`, enforce the same checks used by direct deposits and borrows. For destination deposit increases, require the destination market to be `MarketStatus::Active` and call `validate_max_token_deposits_and_borrows(false)` after the balance update. For destination borrow increases, apply the corresponding borrow-fraction validation by calling the validator with borrow checks enabled where appropriate.

Patch

Resolved in PR#272.

Funding Pause Does Not Stop Spot Interest Accrual Through Public Paths

ID	Severity	Status
OS-VEP-ADV-53	Medium	Resolved

Description

The dedicated spot-interest keeper establishes that the global `FundingPaused` status is intended to stop spot cumulative-interest accrual for this path. In `handle_update_spot_market_cumulative_interest`, the handler loads `State`, checks `state.funding_paused()`, and calls the cumulative-interest controller only when funding is not paused; when funding is paused, it updates only spot market TWAP stats.

programs/velocity/src/instructions/keeper.rs

```rust
pub fn handle_update_spot_market_cumulative_interest(
    ctx: Context<UpdateSpotMarketCumulativeInterest>,
) -> Result<()> {
    [...]
    if !state.funding_paused()? {
        controller::spot_balance::update_spot_market_cumulative_interest(
            spot_market,
            Some(oracle_price_data),
            now,
        )?;
    } else {
        // even if funding is paused still update twap stats
        controller::spot_balance::update_spot_market_twap_stats(
            spot_market,
            Some(oracle_price_data),
            now,
        )?;
    }
    [...]
}
```

Keeper path branches on the global funding pause

The shared `update_spot_market_cumulative_interest` controller does not enforce the same global pause condition. It receives a mutable `SpotMarket`, optional oracle data, and a timestamp, but no `State`, so its only pause check is the market-scoped `SpotOperation::UpdateCumulativeInterest` bit. As a result, any direct caller of this helper can continue past the pause check when the market-scoped operation pause is unset, even if the exchange-wide funding pause is active.

programs/velocity/src/controller/spot_balance.rs

```rust
pub fn update_spot_market_cumulative_interest(
    spot_market: &mut SpotMarket,
    oracle_price_data: Option<&OraclePriceData>,
    now: i64,
) -> VelocityResult {
    if spot_market.is_operation_paused(SpotOperation::UpdateCumulativeInterest) {
        update_spot_market_twap_stats(spot_market, oracle_price_data, now)?;
        return Ok(());
    }

    let InterestAccumulated {
        deposit_interest,
        borrow_interest,
    } = calculate_accumulated_interest(spot_market, now)?;
    [...]
}
```

Shared helper checks only the market-scoped operation pause

When the shared helper proceeds, it changes persistent spot-market accounting rather than merely refreshing oracle-derived statistics. The helper adds lender deposit interest, adds borrow interest, and advances `last_interest_ts`, so a public call into this path during a global funding pause can update the market's cumulative interest state.

programs/velocity/src/controller/spot_balance.rs

```rust
pub fn update_spot_market_cumulative_interest([...]) -> VelocityResult {
    [...]
    if deposit_interest > 0 && borrow_interest > 1 {
        [...]
        if deposit_interest_for_lenders > 0 {
            spot_market.cumulative_deposit_interest = spot_market
                .cumulative_deposit_interest
                .safe_add(deposit_interest_for_lenders)?;

            spot_market.cumulative_borrow_interest = spot_market
                .cumulative_borrow_interest
                .safe_add(borrow_interest)?;
            spot_market.last_interest_ts = now.cast()?;
            [...]
        }
    }
    [...]
}
```

Shared helper updates cumulative interest state

The same successful accrual path also converts portions of the deposit-interest gain into insurance-fund and protocol carveouts, then credits the market revenue and protocol fee pools. The revenue-settlement instruction is exposed through `settle_revenue_to_insurance_fund` and is guarded by `withdraw_not_paused`, while its controller first invokes the shared interest helper before settling revenue from the spot vault to the insurance fund vault.

During an emergency funding pause, this mismatch can allow ordinary public transactions that reach the shared helper to keep changing spot borrow and deposit accounting, `last_interest_ts`, insurance-fund revenue, and protocol fee balances. Downstream flows that depend on the updated spot-market balances, including PnL sufficiency, liquidation health, bankruptcy accounting, and revenue settlement, may therefore observe accounting changes that the dedicated keeper path would have skipped under the same global pause.

Remediation

Either make `FundingPaused` explicitly perp-only and document the spot behavior, or enforce the same global pause policy before every `update_spot_market_cumulative_interest` callsite (thread the global pause into the shared helper).

Patch

Resolved in PR#404.

PnL-vs-Spot Liquidation Can Worsen Account Health When Liquidator Fee Exceeds Buffer

ID	Severity	Status
OS-VEP-ADV-54	Medium	Resolved

Description

`liquidate_perp_pnl_for_deposit` can accept a quote PnL-for-deposit liquidation that does not improve the liquidated account's health. The function first derives a PnL transfer amount from the current margin shortage, the user's quote-denominated negative PnL, and the available deposit amount, then applies the spot deposit transfer and the perp quote asset update before calculating how much margin was freed.There is no explicit postcondition in the transfer path that the post-transfer margin shortage must be less than or equal to the pre-transfer shortage.

After those state updates, `calculate_margin_freed` recomputes the user's liquidation margin calculation and derives `margin_freed` from the difference between the initial and new shortages. Because this uses a saturating subtraction, a liquidation step that increases the shortage is converted into zero freed margin rather than rejected. The controller then increments free margin by that value and continues into the exit-or-bankruptcy branch, so the health-worsening case is not distinguished from a liquidation that simply freed no margin.

programs/velocity/src/controller/liquidation.rs

```
pub fn calculate_margin_freed([...]) -> VelocityResult<(u64, MarginCalculation)> {
    [...]
    let margin_freed = initial_margin_shortage
        .saturating_sub(new_margin_shortage)
        .cast::<u64>()?;
    Ok((margin_freed, margin_calculation_after))
}
```

`calculate_margin_freed` saturates a worsening shortage to zero freed margin.

The transfer-sizing helper does not fully protect this route. `calculate_liability_transfer_to_cover_margin_shortage` returns an effectively unbounded transfer amount when the asset weight is at least the liability weight, or when the liquidation-multiplier-adjusted asset component is at least the liability component. In the quote-collateral PnL-for-deposit case, that equal-weight or multiplier-overlap regime can leave the caller relying on other caps rather than on a bound that guarantees the transfer reduces the shortage.

programs/velocity/src/math/liquidation.rs

```
pub fn calculate_liability_transfer_to_cover_margin_shortage([...]) ->
VelocityResult<u128> {
    // If unsettled pnl asset weight is 1 and quote asset is 1, this calculation breaks
    if asset_weight >= liability_weight {
        return Ok(u128::MAX);
```

```
    }

    let (numerator_scale, denominator_scale) = if liability_decimals > 6 {
        (10_u128.pow(liability_decimals - 6), 1)
    } else {
        (1, 10_u128.pow(6 - liability_decimals))
    };

    let liability_weight_component =
    liability_weight.cast::<u128>()?.safe_mul(10)?; // multiply market weights by
    extra 10 to increase precision

    let asset_weight_component = asset_weight
        .cast::<u128>()?
        .safe_mul(10)?
        .safe_mul(asset_liquidation_multiplier.cast()?)?
        .safe_div(liability_liquidation_multiplier.cast()?)?;

    if asset_weight_component >= liability_weight_component {
        return Ok(u128::MAX);
    }
    [...]
}
```

The transfer-sizing helper returns an unbounded amount in equal-weight or multiplier-overlap cases.

The public handler for this route is gated by the liquidation pause check and by a self-liquidation rejection before dispatching to the controller. It loads the user, liquidator, markets, and oracles, then calls `controller::liquidation::liquidate_perp_pnl_for_deposit` with the caller-supplied market indexes and maximum PnL transfer. Thus, a non-self liquidator can reach the affected controller path when liquidation is not paused.

When a configured perp market's liquidator fee exceeds the effective liquidation buffer, this flow may allow a liquidator to consume the user's quote collateral while leaving the account with a larger margin shortage, making an already-unhealthy account strictly less healthy while consuming its quote deposit, pushing accounts closer to bankruptcy and reducing collateral available for later liquidation/socialization.

Remediation

Add an explicit post-transfer invariant to asset-transferring liquidation paths, including `liquidate_perp_pnl_for_deposit`, that rejects the transaction when the recalculated margin shortage is greater than the initial margin shortage. For non-cancel liquidation steps, require a positive margin improvement unless the user exits liquidation under an explicitly intended condition. Also constrain configuration so the liquidator-fee discount for quote-collateral PnL-for-deposit liquidations cannot exceed the effective liquidation buffer.

Patch

Resolved in PR#408.

`liquidate_perp_with_fill` Budgets IF/Protocol Fees on a Stale Fee Basis

ID	Severity	Status
OS-VEP-ADV-55	Medium	Resolved

Description

`liquidate_perp_with_fill` derives the insurance-fund and protocol liquidation fee budget from the market's base liquidator fee before applying the time-based liquidation fee adjustment. In the with-fill path, this same base fee is also passed into the margin-shortage base-amount calculation, so both the fee budget and the position size needed to cover the shortage are calculated from the un-aged fee basis in `liquidate_perp_with_fill`.

programs/velocity/src/controller/liquidation.rs

```rust
pub fn liquidate_perp_with_fill([...]) -> VelocityResult {
    [...]
        .price;
    let liquidator_fee = market.liquidator_fee;
    // total insurance-side budget with the cap raised to if + protocol rates,
    // split IF-first (see liquidate_perp for rationale)
    let total_if_side_fee = calculate_perp_if_fee(
        intermediate_margin_calculation.tracked_market_margin_shortage(margin_shortage)?,
        user_base_asset_amount,
        margin_ratio_with_buffer,
        liquidator_fee,
        oracle_price,
        quote_oracle_price,
        market
            .if_liquidation_fee
            .safe_add(market.protocol_liquidation_fee)?,
    )?;
    let if_liquidation_fee = total_if_side_fee.min(market.if_liquidation_fee);
    let protocol_liquidation_fee = total_if_side_fee.safe_sub(if_liquidation_fee)?;
    let base_asset_amount_to_cover_margin_shortage = standardize_base_asset_amount_ceil(
        calculate_base_asset_amount_to_cover_margin_shortage(
            margin_shortage,
            margin_ratio_with_buffer,
            liquidator_fee,
            [...]
        )?,
        market.order_step_size,
    )?;
    drop(market);
    [...]
}
```

`liquidate_perp_with_fill` uses the base market liquidator fee for fee budgeting and shortage sizing.

Later in the same `liquidate_perp_with_fill` execution, the function computes an adjusted liquidation fee using the user's `last_active_slot` and the current slot, then passes that adjusted fee only into the temporary liquidation order parameters. This means the forced order can be priced with the aged liquidation discount while the IF/protocol budget and shortage sizing remain based on the lower base fee.

programs/velocity/src/controller/liquidation.rs

```rust
pub fn liquidate_perp_with_fill([...]) -> VelocityResult {
    [...]
    let liquidator_fee_adjusted = get_liquidation_fee(
        liquidator_fee,
        max_liquidation_fee,
        user.last_active_slot,
        slot,
    )?;
    let order_params = get_liquidation_order_params(
        market_index,
        existing_direction,
        base_asset_amount,
        oracle_price,
        liquidator_fee_adjusted,
    )?;
    [...]
}
```

`liquidate_perp_with_fill` applies the adjusted fee only when constructing the liquidation order.

The direct `liquidate_perp` path does not have the same inconsistency. It computes the time-adjusted liquidator fee before deriving the total IF-side fee and before calculating the base asset amount needed to cover the margin shortage, then reuses that single fee basis through both calculations.

`get_liquidation_fee` returns the base fee only during the grace period and increases the fee after enough slots have elapsed, capped by the market's maximum liquidation fee. The with-fill mismatch therefore appears only once this helper can return a value greater than the base fee.

programs/velocity/src/math/liquidation.rs

```rust
pub fn get_liquidation_fee(
    base_liquidation_fee: u32,
    max_liquidation_fee: u32,
    last_active_user_slot: u64,
    current_slot: u64,
) -> VelocityResult<u32> {
    let slots_elapsed = current_slot.safe_sub(last_active_user_slot)?;
    if slots_elapsed < LIQUIDATION_FEE_ADJUST_GRACE_PERIOD_SLOTS {
```

```
            return Ok(base_liquidation_fee);
        }
        [...]
    }
```

get_liquidation_fee increases the fee after the grace period.

Because `calculate_perp_if_fee` subtracts the passed liquidator fee from margin headroom, and `calculate_base_asset_amount_to_cover_margin_shortage` also uses the passed liquidation fee in its sizing formula, using the smaller base fee can overstate the fee budget available to IF/protocol relative to the aged fee basis used to price the liquidation order. For a thin account past the fee-adjustment grace period, the with-fill route may debit IF/protocol fees using the lower-fee budget while also charging the higher aged execution discount, potentially pushing an already-unhealthy account closer to or into bankruptcy.

Remediation

Compute `liquidator_fee_adjusted` before the IF/protocol fee budget and base-amount sizing in `liquidate_perp_with_fill`, and pass that single adjusted fee to `calculate_perp_if_fee`, `calculate_base_asset_amount_to_cover_margin_shortage`, and `get_liquidation_order_params`.

Patch

Resolved in PR#408.

`liquidate_perp_with_fill` Executes a Real Fill During Global FillPaused

ID	Severity	Status
OS-VEP-ADV-56	Medium	Resolved

Description

`handle_liquidate_perp_with_fill` is entered through the liquidation pause guard only. As shown in `handle_liquidate_perp_with_fill`, the access control checks `liq_not_paused` and then loads the liquidatee, liquidator, market maps, oracle map, and maker/referrer accounts before dispatching to the liquidation controller; there is no adjacent `fill_not_paused` guard on this handler.

programs/velocity/src/instructions/keeper.rs

```
#[access_control(
liq_not_paused(&ctx.accounts.state)
)]
pub fn handle_liquidate_perp_with_fill<'c: 'info, 'info>(
    ctx: Context<'info, LiquidatePerp<'info>>,
    market_index: u16,
) -> Result<()> {[...]}
```

`handle_liquidate_perp_with_fill` is guarded only by `liq_not_paused`.

The liquidation controller then turns the liquidation into an actual order fill. `liquidate_perp_with_fill` calls `fill_perp_order` with `FillMode::Liquidation`, so this route reuses the same fill engine that mutates order and position state. `fill_perp_order` does not independently enforce the global fill pause before it proceeds. The checks shown in `fill_perp_order` require the perp market to be `Active` or `ReduceOnly` and require the market-scoped `PerpOperation::Fill` flag to be unpaused, but they do not check `ExchangeStatus::FillPaused` from the global state.

programs/velocity/src/controller/orders.rs

```
pub fn fill_perp_order([...]) -> VelocityResult<(u64, u64)> {
    [...]
    validate!(
        matches!(
            market.status,
            MarketStatus::Active | MarketStatus::ReduceOnly
        ),
        ErrorCode::MarketFillOrderPaused,
        "Market not active",
    )?;
    validate!(
        !market.is_operation_paused(PerpOperation::Fill),
        ErrorCode::MarketFillOrderPaused,
        "Market fills paused",
```

```
    )?;
    [...]
}
```

`fill_perp_order` enforces market-level fill availability but not the global fill pause.

The missing check matters because the global guard exists separately in `fill_not_paused`. That helper rejects execution when `ExchangeStatus::FillPaused` is present, and ordinary keeper fill handlers are wrapped with it, but this liquidation-with-fill path reaches `fill_perp_order` through `liq_not_paused` instead.

programs/velocity/src/instructions/constraints.rs

```
pub fn fill_not_paused(state: &AccountLoader<State>) -> anchor_lang::Result<()> {
    let state = state.load()?;
    if state
        .get_exchange_status()?
        .contains(ExchangeStatus::FillPaused)
    {
        return Err(ErrorCode::ExchangePaused.into());
    }
    Ok(())
}
```

`fill_not_paused` is the global `FillPaused` guard used outside this path.

As a result, when the exchange has `FillPaused` set while `LiqPaused` remains unset, a public liquidation-with-fill invocation can still advance through the fill path if the market-level fill controls allow it. Based on the controller flow, that can consume maker liquidity, and where the fill logic permits AMM participation, AMM liquidity, while mutating the liquidatee and counterparties through the fill. The impact is bounded by the liquidation preconditions and by the operator's ability to also set `LiqPaused`, but it violates the expectation that a standalone global fill pause stops public fill-engine execution.

Remediation

Gate `handle_liquidate_perp_with_fill` with both `liq_not_paused` and `fill_not_paused`, or move the global `ExchangeStatus::FillPaused` check into `fill_perp_order` and explicitly allow only the fill modes that should remain available during a global fill pause.

Patch

Resolved in PR#270.

Swap-Backed Spot Liquidation Bypasses Spot Deposit/Withdraw Pauses

ID	Severity	Status
OS-VEP-ADV-57	Medium	Resolved

Description

The swap-backed spot liquidation route validates that both the asset and liability markets have not paused `SpotOperation::Liquidation`, but it does not also validate the global deposit or withdraw pause bits, nor the market-local `SpotOperation::Withdraw` and `SpotOperation::Deposit` bits. This controller behavior is shown in `liquidate_spot_with_swap_begin`, where the checks are limited to liquidation pauses for the two spot markets.

programs/velocity/src/controller/liquidation.rs

```rust
pub fn liquidate_spot_with_swap_begin([...]) -> VelocityResult {
    [...]

    let asset_spot_market = spot_market_map.get_ref(&asset_market_index)?;

    validate!(
        !asset_spot_market.is_operation_paused(SpotOperation::Liquidation),
        ErrorCode::InvalidLiquidation,
        "Liquidation operation is paused for market {}",
        asset_market_index
    )?;

    let liability_spot_market = spot_market_map.get_ref(&liability_market_index)?;

    validate!(
        !liability_spot_market.is_operation_paused(SpotOperation::Liquidation),
        ErrorCode::InvalidLiquidation,
        "Liquidation operation is paused for market {}",
        liability_market_index
    )?;

    [...]
}
```

Liquidation-swap controller checks only liquidation pauses

The begin instruction wrapper for `handle_liquidate_spot_with_swap_begin` is gated by `liq_not_paused`, which covers the liquidation-wide pause rather than the spot custody pause controls. In the surrounding handler, the instruction initializes flash-loan accounting for the asset and liability markets and sends the requested asset amount from the asset spot market vault to the liquidator-controlled asset token account, so this path performs spot-vault egress while relying on the liquidation gate.

programs/velocity/src/instructions/keeper.rs

```rust
#[access_control(
    liq_not_paused(&ctx.accounts.state)
)]
pub fn handle_liquidate_spot_with_swap_begin<'c: 'info, 'info>(
    ctx: Context<'info, LiquidateSpotWithSwap<'info>>,
    [...]
}
```

Begin handler is gated only by liquidation pause

The paired `handle_liquidate_spot_with_swap_end` wrapper uses the same `liq_not_paused` access control, and finalizes `liquidate_spot_with_swap_end`, so the route also performs spot-vault ingress without the deposit pause checks used elsewhere.

programs/velocity/src/instructions/keeper.rs

```rust
#[access_control(
    liq_not_paused(&ctx.accounts.state)
)]
pub fn handle_liquidate_spot_with_swap_end<'c: 'info, 'info>(
    ctx: Context<'info, LiquidateSpotWithSwap<'info>>,
    [...]
}
```

End handler is gated only by liquidation pause

The generic user swap path applies the custody pause controls that are absent from the liquidation-swap route. In `handle_end_swap`, the code rejects transactions while either global `DepositPaused` or `WithdrawPaused` is set, rejects withdrawals from an input market with `SpotOperation::Withdraw` paused, and rejects deposits to an output market with `SpotOperation::Deposit` paused.

As a result, when standalone spot deposit or withdraw controls are paused while liquidation remains enabled, a public liquidator could still use the swap-backed liquidation flow to move tokens out of the asset market vault and into the liability market vault through the external swap sequence. The impact is limited to accounts that are eligible for liquidation and remains bounded by the route's whitelisted-swap, boundary, oracle, margin, and vault-invariant checks, and operators can still block this specific path by pausing liquidation globally or pausing `SpotOperation::Liquidation` on the affected markets.

Remediation

Apply the same custody pause checks used by the generic spot swap path to `liquidate_spot_with_swap_begin` and `liquidate_spot_with_swap_end`, either in the instruction wrappers or in the liquidation controller. Specifically, reject the route when global `DepositPaused` or `WithdrawPaused` is active, reject asset-market egress when `SpotOperation::Withdraw` is paused, and reject liability-market ingress when `SpotOperation::Deposit` is paused. If liquidation swaps

need a separate operational policy, add dedicated pause bits for token-moving liquidation swaps so operators can keep internal liquidations available without permitting external spot-vault movement during a custody halt.

Patch

Resolved in PR#272.

Permissionless `set_user_status_to_being_liquidated` Ignores Market-Local Liquidation Pauses

ID	Severity	Status
OS-VEP-ADV-58	Medium	Ack'd

Description

`handle_set_user_status_to_being_liquidated` is exposed as a standalone status-setting instruction and is gated by the global liquidation pause check, then loads the user and market maps before calling the liquidation controller. The instruction does not perform a perp-market `PerpOperation::Liquidation` or spot-market `SpotOperation::Liquidation` pause check before delegating to `set_user_status_to_being_liquidated`.

The corresponding Anchor account context only requires the state account, a mutable user account, and a bare signer authority. As shown in `SetUserStatusToBeingLiquidated`, there is no relationship constraint tying the signer to the user, such as a `can_sign_for_user` check, so any signer can invoke this status setter for the supplied user account.

programs/velocity/src/instructions/keeper.rs

```
3346 pub struct SetUserStatusToBeingLiquidated<'info> {
3347     pub state: AccountLoader<'info, State>,
3348     #[account(mut)]
3349     pub user: AccountLoader<'info, User>,
3350     pub authority: Signer<'info>,
3351 }
```

The controller function then treats liquidation status as a margin-calculation result. As shown in `set_user_status_to_being_liquidated`, it rejects bankrupt or already-liquidating users, calculates the liquidation margin requirement, and enters cross-margin or isolated-margin liquidation when the relevant margin requirement is not met, but it does not check whether liquidation operations are paused on the affected perp or spot markets.

This differs from direct liquidation paths in the same controller, which read the target perp or spot market and reject when `PerpOperation::Liquidation` or `SpotOperation::Liquidation` is paused for that market. The status setter can therefore mark an underwater user as being liquidated while direct liquidation for the relevant market is unavailable because of a market-local pause, provided the global liquidation pause remains disabled.

Once the status is set, non-liquidation perp order placement runs the being-liquidated validation before continuing, as shown in `place_perp_order`. That validation can clear the flag only when the margin calculation indicates the user can exit liquidation; otherwise it returns `UserIsBeingLiquidated`, so the effect is a temporary order-placement block against a user who is already below the applicable liquidation margin requirement rather than a durable lock or direct loss of funds.

programs/velocity/src/controller/orders.rs

```rust
pub fn place_perp_order([...]) -> VelocityResult<PlaceOrderResult> {
    [...]
    if !options.is_liquidation() {
        validate_user_not_being_liquidated(
            user,
            perp_market_map,
            spot_market_map,
            oracle_map,
            state.liquidation_margin_buffer_ratio,
        )?;
    }
    [...]
}
```

Remediation

Make `set_user_status_to_being_liquidated` enforce the same market-local liquidation pause semantics as direct liquidation. Before entering isolated liquidation, check the isolated perp market for `PerpOperation::Liquidation` before entering cross-margin liquidation, ensure the perp and spot markets whose positions or liabilities contribute to the liquidation condition are not paused for liquidation, or require a privileged keeper/admin path when any relevant market-local liquidation pause is active.

Patch

This issue was acknowledged by Velocity Exchange.

IF Stake Rebase Floors a Pending Unstake Request to Zero and Strands the Stake

ID	Severity	Status
OS-VEP-ADV-59	Medium	Resolved

Description

Insurance fund stake accounts are lazily rebased when the market insurance fund `shares_base` has advanced past the stake account's `if_base`. In `apply_rebase_to_insurance_fund_stake`, the same `rebase_divisor` used to scale the staker's share balance is also applied to `last_withdraw_request_shares`. Because this is integer division, any nonzero pending withdraw request smaller than the divisor is rounded down to zero while the account is being brought to the current base.

The cancel path applies the market and stake rebases before it checks whether a withdraw request is still in progress. As shown in `cancel_request_remove_insurance_fund_stake`, a request that was nonzero on disk can become zero during the lazy rebase and then fail the cancel validation, so the function never reaches the later state reset that would clear `last_withdraw_request_shares`, `last_withdraw_request_value`, and `last_withdraw_request_ts`.

programs/velocity/src/controller/insurance.rs

```rust
pub fn cancel_request_remove_insurance_fund_stake([...]) -> VelocityResult {
    apply_rebase_to_insurance_fund(insurance_vault_amount, spot_market)?;
    apply_rebase_to_insurance_fund_stake(insurance_fund_stake, spot_market)?;
    let if_shares_before = insurance_fund_stake.checked_if_shares(spot_market)?;
    let total_if_shares_before = spot_market.insurance_fund.total_shares;
    let user_if_shares_before = spot_market.insurance_fund.user_shares;
    validate!(
        insurance_fund_stake.if_base == spot_market.insurance_fund.shares_base,
        ErrorCode::InvalidIFRebase,
        "if stake base != spot market base"
    )?;
    validate!(
        insurance_fund_stake.last_withdraw_request_shares != 0,
        ErrorCode::InvalidIFUnstakeCancel,
        "No withdraw request in progress"
    )?;
    [...]
}
```

Cancel validates the request only after applying the lazy rebase.

The remove path has the same ordering issue. In `remove_insurance_fund_stake`, the stake rebase is applied before `n_shares` is loaded from `last_withdraw_request_shares` and validated, so a pending request that rounds to zero is rejected as though no valid unstake request had been submitted.

programs/velocity/src/controller/insurance.rs

```rust
pub fn remove_insurance_fund_stake([...]) -> VelocityResult<u64> {
    [...]
    apply_rebase_to_insurance_fund(insurance_vault_amount, spot_market)?;
    apply_rebase_to_insurance_fund_stake(insurance_fund_stake, spot_market)?;
    let if_shares_before = insurance_fund_stake.checked_if_shares(spot_market)?;
    let total_if_shares_before = spot_market.insurance_fund.total_shares;
    let user_if_shares_before = spot_market.insurance_fund.user_shares;
    let n_shares = insurance_fund_stake.last_withdraw_request_shares;
    [...]
}
```

Remove loads and validates the rebased request share count.

Because a failed instruction does not persist the in-memory rebase updates, the stale stake account can continue to contain its original nonzero pending request and old `if_base`. Retrying cancel or remove repeats the same rebase-then-reject sequence, so those user-facing exit paths do not clear the request.

The add entrypoint also does not provide a recovery path for this state. `handle_add_insurance_fund_stake` rejects any stake account with a recorded withdraw request, using both the pending request shares and value as its in-progress condition, so a staker cannot add stake while the stale request remains recorded.

programs/velocity/src/instructions/if_staker.rs

```rust
pub fn handle_add_insurance_fund_stake<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    validate!(
        insurance_fund_stake.last_withdraw_request_shares == 0
            && insurance_fund_stake.last_withdraw_request_value == 0,
        ErrorCode::IFWithdrawRequestInProgress,
        "withdraw request in progress"
    )?;
    [...]
}
```

Add rejects stake accounts with a withdraw request in progress.

The request-remove entrypoint similarly blocks overwriting the stale request. As shown in `handle_request_remove_insurance_fund_stake`, it requires `last_withdraw_request_shares` to be zero before it computes and records a new request. Together, these checks can leave the staker unable to cancel, remove, re-request, or add through the exposed IF staking flows after a market-level rebase floors the pending request to zero during lazy account processing.

programs/velocity/src/instructions/if_staker.rs

```rust
pub fn handle_request_remove_insurance_fund_stake(
    ctx: Context<RequestRemoveInsuranceFundStake>,
    market_index: u16,
    amount: u64,
) -> Result<()> {
    [...]
    validate!(
        insurance_fund_stake.last_withdraw_request_shares == 0,
        ErrorCode::IFWithdrawRequestInProgress,
        "Withdraw request is already in progress"
    )?;
    [...]
}
```

Request-remove rejects an account that already records request shares.

Remediation

Update the stake rebase handling for pending IF unstake requests so a nonzero request cannot be silently converted into an uncleared zero-share request. For example, derive the rebased request from `last_withdraw_request_value`, preserve a minimum of one share when a nonzero pending request survives the rebase, or explicitly clear dust requests that round to zero before cancel, remove, add, and request-remove validations run.

Patch

Resolved in PR#266.

IF-Add Auto-Settles Revenue While Global WithdrawPaused Blocks the Direct Settle Path

ID	Severity	Status
OS-VEP-ADV-60	Medium	Resolved

Description

`handle_add_insurance_fund_stake` can reach insurance-fund revenue settlement while the exchange is globally withdraw-paused. The handler validates the stake amount, market identity, market status, absence of an in-progress IF withdraw request, and the market-scoped `InsuranceFundOperation::Add` pause flag, then calls `attempt_settle_revenue_to_insurance_fund` before minting additional IF stake. This entrypoint is not shown with an `access_control` guard for `withdraw_not_paused`, so its auto-settlement path is governed by different pause logic than the direct settlement instruction.

The direct keeper path for `handle_settle_revenue_to_insurance_fund` is explicitly wrapped in `withdraw_not_paused`, meaning the protocol has a withdraw-pause policy for this revenue movement when it is requested through the dedicated settlement instruction.

programs/velocity/src/instructions/keeper.rs

```
#[access_control(
    withdraw_not_paused(&ctx.accounts.state)
)]
pub fn handle_settle_revenue_to_insurance_fund<'c: 'info, 'info>(
    ctx: Context<'info, SettleRevenueToInsuranceFund<'info>>,
    [...]
}
```

The direct keeper settlement path is guarded by `withdraw_not_paused`.

`withdraw_not_paused` rejects any state whose exchange status contains `ExchangeStatus::WithdrawPaused`. As a result, the direct keeper settlement path refuses to proceed during `WithdrawPaused`, while the IF-add path described above does not apply the same global status check before reaching the settlement helper.

programs/velocity/src/instructions/constraints.rs

```
129 pub fn withdraw_not_paused(state: &AccountLoader<State>) -> anchor_lang::Result<()>
    {
130     let state = state.load()?;
131     if state.get_exchange_status()?.contains(ExchangeStatus::WithdrawPaused)
132     {return Err(ErrorCode::ExchangePaused.into());}
133     Ok(())
134 }
```

`withdraw_not_paused` rejects `ExchangeStatus::WithdrawPaused`.

The shared helper performs the actual revenue settlement when the spot market's revenue-settle timer is due. In that case, `attempt_settle_revenue_to_insurance_fund` computes the settlement amount, sends tokens from the spot market vault to the insurance fund vault when the amount is positive, and advances `last_revenue_settle_ts`.

This creates a pause-policy inconsistency: an IF staker with an initialized stake account can use the add-stake flow to trigger a spot-vault decrease, IF-vault increase, revenue-pool settlement, and settlement timestamp update that the direct keeper path blocks during `WithdrawPaused`. The impact is limited because `handle_remove_insurance_fund_stake` is also guarded by `withdraw_not_paused`.

Remediation

Apply the same withdraw-pause decision to every public path that can settle spot-market revenue into an insurance fund vault. Concretely, either add `withdraw_not_paused` to `handle_add_insurance_fund_stake` before it calls `attempt_settle_revenue_to_insurance_fund`, or move an explicit `WithdrawPaused` check into `attempt_settle_revenue_to_insurance_fund` with a caller mode if some callers are intentionally exempt. The direct keeper settlement path and IF-add auto-settlement path should behave identically while `ExchangeStatus::WithdrawPaused` is set.

Patch

Resolved in PR#272.

Revenue-Pool Deposit Credits at Stale Cumulative Deposit Interest

ID	Severity	Status
OS-VEP-ADV-61	Medium	Resolved

Description

`handle_deposit_into_spot_market_revenue_pool` credits the spot market's `revenue_pool` balance before receiving the donated tokens, but it does not first refresh the market's cumulative deposit interest. The handler checks that the amount is nonzero and that the market is not in settlement, then directly calls the revenue-pool balance update helper using the current `spot_market` state. If that state contains a stale cumulative deposit interest value, the scaled deposit balance for the revenue pool is computed against the stale value.

programs/velocity/src/instructions/user.rs

```rust
pub fn handle_deposit_into_spot_market_revenue_pool<'c: 'info, 'info>(
    ctx: Context<'info, RevenuePoolDeposit<'info>>,
    amount: u64,
) -> Result<()> {
    [...]
    let mut spot_market = load_mut!(ctx.accounts.spot_market)?;
    let remaining_accounts_iter = &mut ctx.remaining_accounts.iter().peekable();
    let mint = get_token_mint(remaining_accounts_iter)?;
    validate!(
        !spot_market.is_in_settlement(Clock::get()?.unix_timestamp),
        ErrorCode::DefaultError,
        "spot market {} not active",
        spot_market.market_index
    )?;
    controller::spot_balance::update_revenue_pool_balances(
        amount.cast::<u128>()?,
        &SpotBalanceType::Deposit,
        &mut spot_market,
    )?;
    [...]
}
```

`handle_deposit_into_spot_market_revenue_pool` credits the revenue pool without first refreshing cumulative deposit interest.

The ordinary `handle_deposit` path uses a different ordering. After loading the market and oracle data and validating deposit eligibility, it calls `update_spot_market_cumulative_interest` before the user's spot balance is credited. This makes ordinary deposits use the current cumulative deposit interest when converting token amounts into scaled balances, while the revenue-pool deposit path shown above does not perform the same refresh.

programs/velocity/src/instructions/user.rs

```rust
pub fn handle_deposit<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    validate!(
        !spot_market.is_operation_paused(SpotOperation::Deposit),
        ErrorCode::MarketActionPaused,
        "spot market {} deposits paused",
        market_index
    )?;
    controller::spot_balance::update_spot_market_cumulative_interest(
        &mut spot_market,
        Some(&oracle_price_data),
        now,
    )?;
    [...]
}
```

`handle_deposit` refreshes cumulative deposit interest before crediting the user deposit.

`update_revenue_pool_balances` copies `spot_market.revenue_pool`, passes it into `update_spot_balances`, and writes it back to the market. The linked helper does not refresh cumulative interest itself; it relies on the caller-provided `spot_market` accounting state. As a result, a revenue-pool deposit made while the market's stored cumulative deposit interest is below the current value can receive more scaled deposit balance than it would have received after the same refresh performed by normal deposits.

programs/velocity/src/controller/spot_balance.rs

```rust
pub fn update_revenue_pool_balances(
    token_amount: u128,
    update_direction: &SpotBalanceType,
    spot_market: &mut SpotMarket,
) -> VelocityResult {
    let mut spot_balance = spot_market.revenue_pool;
    update_spot_balances(
        token_amount,
        update_direction,
        spot_market,
        &mut spot_balance,
        false,
    )?;
    spot_market.revenue_pool = spot_balance;

    Ok(())
}
```

Delegation of balance conversion to `update_spot_balances` using the provided market state.

During revenue settlement, `settle_revenue_to_insurance_fund` refreshes cumulative deposit interest before valuing `spot_market.revenue_pool.scaled_balance` as a token amount. This ordering can revalue the over-credited scaled balance at the refreshed cumulative deposit interest, allowing a fresh revenue-pool donation to participate in deposit interest that accrued before the donation existed. The impact is bounded by market staleness and deposit-rate accrual, and the realistic magnitude may be small because cumulative deposit interest is refreshed on many spot-market interactions.

programs/velocity/src/controller/insurance.rs

```
pub fn settle_revenue_to_insurance_fund([...]) -> VelocityResult<u64> {
    update_spot_market_cumulative_interest(spot_market, None, now)?;
    if spot_market.insurance_fund.revenue_settle_period == 0 {
        // revenue pool not configured to settle, ending early
        return Ok(0);
    }
    let depositors_claim =
        validate_spot_market_vault_amount(spot_market, spot_market_vault_amount)?;
    let mut token_amount = get_token_amount(
        spot_market.revenue_pool.scaled_balance,
        spot_market,
        &SpotBalanceType::Deposit,
    )?;
    [...]
}
```

`settle_revenue_to_insurance_fund` refreshes cumulative deposit interest before valuing the revenue-pool scaled balance.

Remediation

Call `update_spot_market_cumulative_interest` at the start of `handle_deposit_into_spot_market_revenue_pool` before `update_revenue_pool_balances`, matching the ordinary deposit ordering. The refresh should use the same market timing and oracle context required by the cumulative-interest update so the revenue-pool deposit is converted to scaled balance at current cumulative deposit interest.

Patch

Resolved in PR#266.

Direct Revenue-to-IF Settlement Ignores the Market-Scoped SpotOperation::Withdraw Pause

ID	Severity	Status
OS-VEP-ADV-62	Medium	Resolved

Description

`handle_settle_revenue_to_insurance_fund` is protected by the exchange-wide withdrawal pause, but the handler does not check whether withdrawals are paused on the specific `SpotMarket` being settled. After validating the market account, revenue settlement period, and settlement timing, it calls the insurance controller to calculate the settlement amount and then transfers that amount from the spot market vault to the insurance fund vault.

The controller path confirms that the settlement amount is derived from the market's `revenue_pool` balance, capped by vault and insurance fund conditions, and then applied back to the spot market by updating revenue pool balances as a borrow. This means the settlement path is not merely accounting metadata in isolation; it determines an amount of market revenue to move into the insurance fund path, as shown in `settle_revenue_to_insurance_fund`.

By contrast, the normal user withdrawal balance update path enforces the market-scoped withdrawal pause. `update_spot_balances_and_cumulative_deposits_with_limits` checks the market status and then rejects when `spot_market.is_operation_paused(SpotOperation::Withdraw)` is true, so ordinary withdrawals observe the market-level pause bit that the revenue settlement path bypasses.

programs/velocity/src/controller/spot_position.rs

```
pub fn update_spot_balances_and_cumulative_deposits_with_limits(
    token_amount: u128,
    update_direction: &SpotBalanceType,
    spot_market: &mut SpotMarket,
    user: &mut User,
) -> VelocityResult {
    [...]
    validate!(
        !spot_market.is_operation_paused(SpotOperation::Withdraw),
        ErrorCode::MarketWithdrawPaused,
        "Spot Market {} withdraws are currently paused",
        spot_market.market_index
    )?;
    [...]
}
```

Normal spot withdrawals reject when the market-scoped withdraw operation is paused.

`SpotOperation::Withdraw` is one of the spot-market operation bits, separate from the global pause guard used by the settlement handler. Because the revenue-to-insurance-fund handler

only applies the global withdrawal guard and never evaluates this spot operation bit, pausing withdrawals for a market does not pause this direct spot-vault-to-insurance-fund egress path.

programs/velocity/src/state/paused_operations.rs

```rust
#[derive(Clone, Copy, PartialEq, Debug, Eq)]
pub enum SpotOperation {
    UpdateCumulativeInterest = 0b00000001,
    Fill = 0b00000010,
    Deposit = 0b00000100,
    Withdraw = 0b00001000,
    Liquidation = 0b00010000,
}
```

`SpotOperation::Withdraw` is a spot-market operation pause bit.

As a result, when an operator pauses `SpotOperation::Withdraw` on a spot market to stop withdrawals from that market's vault, revenue-pool tokens may still be moved from the market vault into the insurance fund vault through this settlement instruction. The account context for `SettleRevenueToInsuranceFund` does not require a caller signer or market authority signer, so the path is callable without market-authority authorization. The moved assets are protocol revenue routed to another protocol vault rather than direct user theft, but the bypass can still undermine the intended emergency control and change insurance fund vault balances and share value while market withdrawals are paused.

Remediation

Apply the target spot market's `SpotOperation::Withdraw` pause to `handle_settle_revenue_to_insurance_fund` before calculating or transferring the settlement amount, or add a distinct market-scoped revenue-settlement pause if revenue settlement is intended to be controlled separately from user withdrawals.

Patch

Resolved in PR#272.

Revenue-Pool Deposit Bypasses the Market-Scoped SpotOperation::Deposit Pause

ID	Severity	Status
OS-VEP-ADV-63	Medium	Resolved

Description

`handle_deposit_into_spot_market_revenue_pool` is protected by the exchange-wide `deposit_not_paused` access control and rejects zero amounts, but it does not check whether the target `SpotMarket` has its market-local deposit operation paused.

The direct user deposit path applies the missing market-scoped circuit breaker before proceeding with its deposit accounting. In `handle_deposit`, the spot market is loaded from the market map and the handler rejects deposits when `SpotOperation::Deposit` is paused for that specific market, which makes the direct route stricter than the revenue-pool route for the same class of token ingress.

programs/velocity/src/instructions/user.rs

```
pub fn handle_deposit<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    validate!(
        !spot_market.is_operation_paused(SpotOperation::Deposit),
        ErrorCode::MarketActionPaused,
        "spot market {} deposits paused",
        market_index
    )?;[...]
}
```

The direct deposit handler rejects deposits when the spot market's deposit operation is paused.

`SpotOperation::Deposit` is one of the per-spot-market pause bits defined in the paused-operations state. Because the revenue-pool handler never consults this bit, pausing deposits on a specific spot market does not consistently prevent all deposit-like flows into that market's vault and accounting state.

programs/velocity/src/state/paused_operations.rs

```
#[derive(Clone, Copy, PartialEq, Debug, Eq)]
pub enum SpotOperation {
    UpdateCumulativeInterest = 0b00000001,
    Fill = 0b00000010,
    Deposit = 0b00000100,
    Withdraw = 0b00001000,
    Liquidation = 0b00010000,
}
```

`SpotOperation::Deposit` is defined as a spot-market pause bit.

The result is a market-scoped emergency-control bypass: an operator may pause `SpotOperation::Deposit` to stop token ingress for a market, while a caller can still fund a revenue-pool deposit that transfers tokens into the spot vault and increases `spot_market.revenue_pool`. Since this remains backed by caller-supplied tokens and still passes the later vault and max-token checks, the impact is limited to pause-boundary bypass and inconsistent enforcement.

Remediation

Add the same market-local pause validation used by `handle_deposit` to `handle_deposit_into_spot_market_revenue_pool` before `update_revenue_pool_balances` and before receiving tokens. The check should reject the call when `spot_market.is_operation_paused(SpotOperation::Deposit)` is true and return the appropriate paused-market error.

Patch

Resolved in PR#272.

DLOB Match Quoter Reprices Oracle-Offset Makers Against a Zero Oracle

ID	Severity	Status
OS-VEP-ADV-64	Medium	Resolved

Description

In `fulfill_perp_order_step`, the unified match path builds a `QuoteContext` after AMM setup, but the context's oracle reference is backed by `OraclePriceData::default()` rather than the oracle price already available to the function. As shown in the matcher context construction, this stub oracle is then shared with the quoters used by `match_take`.

programs/velocity/src/controller/orders.rs

```
pub fn fulfill_perp_order_step([...]) -> VelocityResult<(u64, u64, u64)> {
    [...]
    let stats_snapshot = market.market_stats;
    let oracle_stub = OraclePriceData::default();
    let ctx = QuoteContext {
        stats: &stats_snapshot,
        oracle: &oracle_stub,
        mm_oracle: None,
        oracle_validity: None,
        [...]
    };
    [...]
}
```

Match context uses a default oracle

The default oracle value is material because `OraclePriceData` derives `Default` and its `price` field is a plain signed integer. A default instance therefore supplies a zero oracle price to any downstream code that reads `ctx.oracle.price`, as shown by the oracle data structure definition.

programs/velocity/src/state/oracle.rs

```
#[derive(Default, Clone, Copy, Debug)]
pub struct OraclePriceData {
    pub price: i64,
    pub confidence: u64,
    pub delay: i64,
    pub has_sufficient_number_of_data_points: bool,
    pub sequence_id: Option<u64>,
}
```

Default oracle price data

`DlobOrderQuoter::effective_price` recomputes the maker order's limit price from the `QuoteContext` oracle. For oracle-offset orders, this means the maker is quoted from the default context oracle instead of the real oracle price, so the effective maker price can become the signed offset alone rather than the intended oracle-derived limit.

programs/velocity/src/state/quoter.rs

```
411 fn effective_price(&self, ctx: &QuoteContext) -> VelocityResult<Option<u64>> {
412     // Slot 0 + valid_oracle_price = ctx.oracle.price; this drives
413     // get_limit_price's auction / oracle-offset handling.
414     self.order
415         .get_limit_price(Some(ctx.oracle.price), None, ctx.slot, ctx.tick.max(1))
416 }
```

DLOB quoter recomputes from context oracle

This differs from the earlier maker discovery flow in `fill_perp_order`, where the code computes whether the oracle price is valid for order pricing and then calls `get_maker_orders_info` with the real `oracle_price`. The maker selected for a `PerpFulfillmentMethod::Match` is therefore routed with a maker price discovered from the live oracle context, not from the later default matcher context.

The resulting mismatch breaks the route-and-fill invariant for oracle-offset makers: discovery stores a maker price based on the real oracle, while the DLOB quoter inside the Match branch may produce a fill price based on a zero oracle. The AMM JIT quoter is initialized with the routed maker price and `valid_oracle_price`, but the DLOB maker quoter still uses the shared `QuoteContext` oracle when it recomputes its effective price.

Settlement then validates the DLOB match fill against `match_maker_price`, which is the frozen maker price selected during routing. If the DLOB quoter reprices an oracle-offset maker against zero, the fill can fail this validation rather than committing at the originally selected maker price.

programs/velocity/src/controller/orders.rs

```
1  fn settle_dlob_match_fill([...]) -> VelocityResult<(u64, u64, u64)> {
2      [...]
3      validate_fill_price(
4          fill.quote_filled,
5          fill.base_filled,
6          BASE_PRECISION_U64,
7          maker_direction,
8          match_maker_price,
9          false,
10     )?;
11     [...]
12 }
```

Settlement validates against routed maker price

Oracle-offset perp maker liquidity can therefore become unfillable through the standard DLOB Match path when a crossing taker is routed to such a maker: the match can be quoted against the wrong oracle context and then rejected by settlement.

Remediation

Thread the same safe oracle price used for maker discovery into the Match branch's `QuoteContext` instead of constructing it with `OraclePriceData::default()`. Alternatively, make the DLOB maker quoter commit fills at the routed `match_maker_price` for the Match path rather than recomputing the limit from `ctx.oracle.price`.

Patch

Resolved in PR#274.

Market-Scoped AmmImmediateFill Pause Bit Is Defined and Settable but Never Enforced

ID	Severity	Status
OS-VEP-ADV-65	Medium	Resolved

Description

Velocity defines `PerpOperation::AmmImmediateFill` as a market-scoped pause operation and includes it in the complete perp operation list used for pause logging. This makes the bit part of the market `paused_operations` model rather than an unused numeric gap, as shown by the enum and `ALL_PERP_OPERATIONS` membership.

programs/velocity/src/state/paused_operations.rs

```
#[derive(Clone, Copy, PartialEq, Debug, Eq)]
pub enum PerpOperation {
    UpdateFunding = 0b00000001,
    AmmFill = 0b00000010,
    Fill = 0b00000100,
    SettlePnl = 0b00001000,
    SettlePnlWithPosition = 0b00010000,
    Liquidation = 0b00100000,
    AmmImmediateFill = 0b01000000,
    SettleRevPool = 0b10000000,
}
const ALL_PERP_OPERATIONS: [PerpOperation; 8] = [
    PerpOperation::UpdateFunding,
    PerpOperation::AmmFill,
    PerpOperation::Fill,
    PerpOperation::SettlePnl,
    PerpOperation::SettlePnlWithPosition,
    PerpOperation::Liquidation,
    PerpOperation::AmmImmediateFill,
    PerpOperation::SettleRevPool,
];

```

The immediate-fill auction-skip decision is made in `PerpMarket::can_skip_auction_duration`. It blocks only when `state.amm_immediate_fill_paused()` is true, then otherwise allows the skip path when the AMM revenue and inventory conditions are satisfied; it does not consult the market's own `paused_operations` field for `PerpOperation::AmmImmediateFill`.

`state.amm_immediate_fill_paused()` reads `ExchangeStatus::AmmImmediateFillPaused` from the global `State.exchange_status` bitfield. This reinforces that the pause enforced by `can_skip_auction_duration` is exchange-wide, not the per-market bit that operators can set on an individual perp market.

programs/velocity/src/state/state.rs

```rust
pub fn amm_immediate_fill_paused(&self) -> VelocityResult<bool> {
    Ok(self
        .get_exchange_status()?
        .contains(ExchangeStatus::AmmImmediateFillPaused))
}
```

The AMM fill path separately enforces the market-scoped `PerpOperation::AmmFill` pause before evaluating oracle and risk gates, but does not include a corresponding check for `PerpOperation::AmmImmediateFill`. In `amm_can_fill_order`, non-low-risk orders that pass immediate-fill oracle and AMM inventory checks call `can_skip_auction_duration`, so the market-scoped immediate-fill bit is not applied before the immediate AMM route is accepted.

As a result, pausing only `AmmImmediateFill` on a single market may not disable the immediate AMM route for that market. Public fillers can continue to route non-low-risk orders straight into the standalone immediate AMM path, mutating that market's AMM reserves through the route the control was meant to disable. During a market-specific incident, operators would have to use the broader global `AmmImmediateFillPaused` control or the market `AmmFill` pause, which also affects behavior beyond the intended per-market immediate-fill circuit breaker.

Remediation

Enforce `PerpOperation::AmmImmediateFill` on the immediate-fill path before the profitability and inventory checks in `PerpMarket::can_skip_auction_duration`, returning `Ok(false)` when the market operation is paused.

Patch

Resolved in PR#270.

Permissionless Perp Fee Sweep Under-Reserves Expiry-Price Claims During Settlement

ID	Severity	Status
OS-VEP-ADV-66	Medium	Resolved

Description

The `handle_sweep_perp_market_fees` keeper entrypoint can run for a perp market that is already in `Settlement`. Its access control uses `perp_market_valid` and `exchange_not_paused`, and the linked constraint helper only rejects `Delisted` markets. The `SweepPerpMarketFees` account context also does not include a signer account, so the sweep is available to any caller who supplies the required market, spot market, and oracle accounts, as shown in the entrypoint guard.

programs/velocity/src/instructions/keeper.rs

```
#[access_control(
    perp_market_valid(&ctx.accounts.perp_market)
    exchange_not_paused(&ctx.accounts.state)
)]
pub fn handle_sweep_perp_market_fees(
    ctx: Context<SweepPerpMarketFees>,
    perp_market_index: u16,
) -> Result<()> {
    let clock = Clock::get()?;
    [...]
}
```

Settlement markets are not rejected by the sweep entrypoint guard.

Inside `handle_sweep_perp_market_fees`, the amount reserved for users is derived from `calculate_net_user_pnl` using the current oracle price. That is appropriate for live PnL accounting, but in settlement the users' remaining claims are no longer priced by a live oracle; they are later settled against the market's fixed expiry price. The live-oracle reservation path is shown where the handler computes `net_user_pnl` and passes it into `sweep_market_fees`.

programs/velocity/src/instructions/keeper.rs

```
pub fn handle_sweep_perp_market_fees([...]) -> Result<()> {
    [...]
    let net_user_pnl = calculate_net_user_pnl(
        &perp_market.amm,
        oracle_price,
        perp_market.quote_asset_amount,
        perp_market.net_unsettled_funding_pnl,
    )?;
    let (if_swept, protocol_swept, amm_provision_tokenized) =
        controller::perp_pools::sweep_market_fees(
```

```
            perp_market,
            spot_market,
            net_user_pnl,
            now,
            false,
        )?;
    [...]
}
```

The sweep reserves claims using live-oracle net user PnL.

`sweep_market_fees` treats `net_user_pnl.max(0)` as the user-claim reserve and then makes the protocol fee transfer before applying the buffer used for the later insurance and AMM-provision fee drains. Because the protocol drain is buffer-exempt and consumes `pending_protocol_fee` from the PnL pool first, an understated `net_user_pnl` can make the function observe surplus that is actually needed for expiry-price claims, as shown in the fee waterfall.

programs/velocity/src/controller/perp_pools.rs

```
pub fn sweep_market_fees([...]) -> VelocityResult<(u128, u128, u128)> {
    [...]
    // not recoverable later anyway)
    let reserved_claims: u128 = net_user_pnl.max(0).cast::<u128>()?;
    let mut available_unbuffered: u128 =
    pnl_pool_tokens.saturating_sub(reserved_claims);
    // 1. protocol's withdrawable cut (buffer-exempt: only user claims reserved)
    let protocol_drain = market
        .fee_ledger
        .pending_protocol_fee
        .min(available_unbuffered);
    if protocol_drain > 0 {
        transfer_spot_balances(
            protocol_drain.cast()?,
            spot_market,
            &mut market.pnl_pool,
            &mut market.protocol_fee_pool,
        )?;
        market.fee_ledger.consume_pending_protocol(protocol_drain)?;
        available_unbuffered = available_unbuffered.safe_sub(protocol_drain)?;
    }
    [...]
}
```

The fee waterfall reserves only positive net user PnL before draining protocol fees.

Expired position settlement uses `perp_market.expiry_price` to value the user's base position before computing the PnL to settle. For an expired market with net-long user exposure, if the

current oracle price is below the fixed expiry price, the sweep's live-oracle PnL can be lower than the liability that `settle_expired_position` will later compute from the expiry price.

programs/velocity/src/controller/pnl.rs

```
pub fn settle_expired_position([...]) -> VelocityResult {
    [...]
    let base_asset_value = calculate_base_asset_value_with_expiry_price(
        &user.perp_positions[position_index],
        perp_market.expiry_price,
    )?;
    [...]
}
```

Expired settlement values the position at the fixed expiry price.

The settlement path then calls `update_pnl_pool_and_user_balance`, which only settles positive user PnL up to the current PnL pool balance and validates that the full requested amount was available. If the earlier sweep moved pending protocol, insurance, or AMM-provision fees out of the PnL pool based on an understated reserve, a later expiry settlement can fail with `InsufficientPerpPnlPool` or remain delayed until the pool is replenished.

programs/velocity/src/controller/perp_pools.rs

```
pub fn update_pnl_pool_and_user_balance(
    market: &mut PerpMarket,
    quote_spot_market: &mut SpotMarket,
    user: &mut User,
    unrealized_pnl_with_fee: i128,
) -> VelocityResult<i128> {
    [...]
    validate!(
        unrealized_pnl_with_fee == pnl_to_settle_with_user,
        ErrorCode::InsufficientPerpPnlPool,
        "pnl_pool_amount doesnt have enough ({} < {})",
        pnl_to_settle_with_user,
        unrealized_pnl_with_fee
    )?;
    [...]
}
```

A short PnL pool causes settlement to revert with `InsufficientPerpPnlPool`.

Remediation

Prevent `sweep_perp_market_fees` and the non-final inline sweep from running while a perp market is in `Settlement` unless all remaining settlement exposure is zero. If sweeping during `Settlement` is required, compute the reserved user claims from the same fixed `expiry_price` used by

`settle_expired_position`, then pass that expiry-price liability into the fee waterfall so pending fees are drained only from true surplus after all expiry claims remain fully backed.

Patch

Resolved in PR#273.

Market-Scoped Settle-PnL Pause Does Not Stop Expired-Position Settlement

ID	Severity	Status
OS-VEP-ADV-67	Medium	Resolved

Description

The settle-PnL handlers distinguish settlement markets from the normal PnL path before dispatching into the controller. In `handle_settle_pnl`, the handler is protected by the global `settle_pnl_not_paused` wrapper, but when the selected perp market is in `MarketStatus::Settlement` it only calls the global AMM pause helper before invoking `settle_expired_position`.

`handle_settle_multiple_pnls` uses the same pattern while iterating over requested markets. For each market already in `Settlement`, it checks only the global AMM pause state and then calls `settle_expired_position`, so the multi-market handler has the same bypass of market-scoped settle-PnL operation pauses.

The normal `settle_pnl` controller path does enforce the market-level pause bits. It rejects settlement when `PerpOperation::SettlePnl` is paused, and separately rejects settlement with nonzero base exposure when `PerpOperation::SettlePnlWithPosition` is paused, which is the control that the settlement-market branch does not reach.

programs/velocity/src/controller/pnl.rs

```
pub fn settle_pnl([...]) -> VelocityResult {
    [...]
    if perp_market.is_operation_paused(PerpOperation::SettlePnl) {
        let msg = format!(
            "Cannot settle pnl under current market = {} status",
            market_index
        );
        return mode.result(
            ErrorCode::InvalidMarketStatusToSettlePnl,
            market_index,
            &msg,
        );
    }

    let base_asset_amount = user.perp_positions[position_index].base_asset_amount;
    if base_asset_amount != 0 {
        if perp_market.is_operation_paused(PerpOperation::SettlePnlWithPosition) {
            let msg = format!(
                "Cannot settle pnl with position under current market = {} operation
                paused",
                market_index
```

```
            );
            return mode.result(
                ErrorCode::InvalidMarketStatusToSettlePnl,
                market_index,
                &msg,
            );
        }
        [...]
    }[...]
}
```

Normal PnL settlement enforces market-scoped settle-PnL pause bits.

`settle_expired_position` performs the expired-position settlement flow without checking `paused_operations`. As a result, when only the market-scoped `SettlePnl` or `SettlePnlWithPosition` pause bits are set, a caller can still progress expired-position settlement through these handlers as long as the global settle-PnL and AMM pause gates are not active.

This undermines the intended market-scoped emergency control for settlement. An operator attempting to pause settlement on one market, for example while reviewing PnL-pool sufficiency or expiry-price correctness, may still see expired positions closed and user balances and the market PnL pool updated through the settlement path.

Remediation

Apply the same market-scoped pause checks used by `settle_pnl` before every call to `settle_expired_position`: reject when the target perp market has `PerpOperation::SettlePnl` paused, and reject with `PerpOperation::SettlePnlWithPosition` when the user's expired position has nonzero base exposure. Prefer placing these checks in a shared helper or inside `settle_expired_position` so both single- and multi-market handlers must pass the same gate before mutating expired positions, the PnL pool, or user quote balances.

Patch

Resolved in PR#270.

TrySettle Soft-Skips a Paused Market's PnL Settle but Still Sweeps Its Revenue Share

ID	Severity	Status
OS-VEP-ADV-68	Medium	Resolved

Description

The public `settle_multiple_pnls` instruction accepts a caller-supplied `SettlePnlMode` and forwards it to the batch settlement handler. This makes `TrySettle` available on the multi-market settlement path rather than forcing every market settlement attempt to fail closed when a market-specific settle-PnL condition rejects settlement.

programs/velocity/src/lib.rs

```rust
pub fn settle_multiple_pnls<'c: 'info, 'info>(
    ctx: Context<'info, SettlePNL>,
    market_indexes: Vec<u16>,
    mode: SettlePnlMode,
) -> Result<()> {
    handle_settle_multiple_pnls(ctx, market_indexes, mode)
}
```

The batch settlement entrypoint forwards the caller-selected mode.

In `handle_settle_multiple_pnls`, the batch instruction is protected by the global settle-PnL pause check, then loads the requested markets, optional revenue-share accounts, and iterates through the supplied market indexes. For non-settlement markets, the handler calls `controller::pnl::settle_pnl` with the caller-provided mode, so the handler's later behavior depends on whether `settle_pnl` returns an error or a successful result for that market.

`settle_pnl` checks the market-level `SettlePnl` pause and, when the user still has base asset exposure, the `SettlePnlWithPosition` pause before proceeding with PnL settlement. In both pause cases, it returns through the settlement mode helper instead of directly returning an error, so these market-specific pause conditions can be converted into success when the selected mode allows soft failure.

programs/velocity/src/controller/pnl.rs

```rust
pub fn settle_pnl([...]) -> VelocityResult {
    [...]
    if perp_market.is_operation_paused(PerpOperation::SettlePnl) {
        let msg = format!(
            "Cannot settle pnl under current market = {} status",
            market_index
        );
        return mode.result(
            ErrorCode::InvalidMarketStatusToSettlePnl,
```

```
            market_index,
            &msg,
        );
    }
    let base_asset_amount = user.perp_positions[position_index].base_asset_amount;
    if base_asset_amount != 0 {
        if perp_market.is_operation_paused(PerpOperation::SettlePnlWithPosition) {
            let msg = format!(
                "Cannot settle pnl with position under current market = {} operation
                paused",
                market_index
            );
            return mode.result(
                ErrorCode::InvalidMarketStatusToSettlePnl,
                market_index,
                &msg,
            );
        }
        [...]
    }[...]
}
```

`settle_pnl` routes market pause failures through the mode helper.

The `SettlePnlMode` helper maps `MustSettle` to an error and `TrySettle` to success for the same rejected settlement condition. As a result, when `settle_multiple_pnls` is called with `TrySettle`, a paused market can produce `Ok(())` even though the PnL settlement did not execute.

programs/velocity/src/state/settle_pnl_mode.rs

```
pub fn result(self, error_code: ErrorCode, market_index: u16, msg: &str) ->
VelocityResult {
    [...]
    match self {
        SettlePnlMode::MustSettle => Err(error_code),
        SettlePnlMode::TrySettle => Ok(()),
    }
}
```

`TrySettle` converts those failures into success.

After the `settle_pnl` call returns, `handle_settle_multiple_pnls` unconditionally proceeds to builder-code side effects for that same market when builder codes are enabled and the optional escrow and revenue-share map are present. The code revokes completed orders and calls `sweep_completed_revenue_share_for_market` without tracking whether the preceding PnL settlement actually settled anything or was soft-skipped by `TrySettle`.

`sweep_completed_revenue_share_for_market` then filters completed or referral revenue-share rows for the same perp market and checks only whether the market PnL pool has enough quote-denominated balance to cover the accrued fees. If it does, the function transfers those accrued fees from the market's PnL pool to the referrer or builder account, so a market-scoped settle-PnL pause does not prevent this PnL-pool value transfer on the soft-skipped batch path.

The impact is limited to configurations where builder codes are enabled, the relevant revenue-share accounts are supplied, and a matching completed or referral revenue-share row has accrued fees. Under those conditions, a market-specific settle-PnL pause can stop user PnL settlement while still allowing completed builder or referrer fees to be swept from that paused market's PnL pool.

Remediation

Track the result of each market settlement attempt separately from the instruction-level `Result`. When `settle_pnl` returns success because `TrySettle` soft-skipped a market-specific pause or other non-settled condition, skip `revoke_completed_orders`, `sweep_completed_revenue_share_for_market`, and any other post-settlement side effects for that market. Also apply the same relevant market operation-pause policy to the revenue-share sweep itself so funds are not transferred from a market PnL pool while that market's settle-PnL or revenue-pool operation is paused.

Patch

Resolved in PR#396.

Expired-Position Closeout Fee Bypasses the Perp Fee-Ledger Waterfall

ID	Severity	Status
OS-VEP-ADV-69	Medium	Resolved

Description

When a perp market is in `Settlement`, `handle_settle_pnl` routes the user through `settle_expired_position` rather than the normal PnL-settlement branch. In that expired-position path, the closeout fee is computed from the expired position value and fee tier, then debited from the user's perp quote amount through the position accounting path, as shown in `settle_expired_position`.

programs/velocity/src/controller/pnl.rs

```rust
pub fn settle_expired_position([...]) -> VelocityResult {
    [...]
    let fee = base_asset_value
        .safe_mul(fee_structure.fee_tiers[0].fee_numerator as i64)?
        .safe_div(fee_structure.fee_tiers[0].fee_denominator as i64)?;

    update_quote_asset_and_break_even_amount(
        &mut user.perp_positions[position_index],
        perp_market,
        -fee.abs(),
    )?;
    [...]
}
```

Expired-position closeout debits the locally computed fee

After debiting the user, `settle_expired_position` continues by settling the resulting quote amount with the market PnL pool. When it emits the synthetic fill event for the expired closeout, the fee appears only as the fill record's taker fee field. The function does not accrue the fee into the market fee ledger before the user and market balances are finalized. Ordinary perp fills follow a different accounting path. In `settle_amm_house_fill`, the taker fee is split into protocol, insurance-fund, and AMM-provision components, and those carveouts are recorded through the market's fee ledger, as shown in the ordinary AMM fill path.

programs/velocity/src/controller/orders.rs

```rust
fn settle_amm_house_fill([...]) -> VelocityResult<(u64, u64)> {
    [...]
    // gross taker fee for analytics plus the explicit protocol / IF / AMM
    // carveouts of the trade-fee remainder. All three accrue as pending quote
    // counters here (the quote spot market isn't in scope at fill); their
    // token value lands in the pnl pool as fills settle and is materialized
    // into revenue_pool / protocol_fee_pool / amm.fee_pool by
```

```
    // `sweep_market_fees`. The AMM provision also grows the lifetime
    // backstop-of-last-resort clawback cap.
    market
        .fee_ledger
        .accrue_fill_fees(user_fee, protocol_fee, if_fee, amm_fee)?;
    [...]
}
```

Ordinary AMM fills accrue fee carveouts into the market ledger

`sweep_market_fees` materializes those ledgered carveouts out of the perp market PnL pool. As shown in below, it returns without transferring anything when the pending protocol, insurance-fund, and AMM-provision counters are all zero, and its later transfers are bounded by those pending counters. A closeout fee that was only debited from the user, without incrementing those counters, therefore has no destination tag for this waterfall.

programs/velocity/src/controller/perp_pools.rs

```
pub fn sweep_market_fees([...]) -> VelocityResult<(u128, u128, u128)> {
    [...]
    if (!force && market.is_operation_paused(PerpOperation::SettleRevPool))
        || (market.fee_ledger.pending_protocol_fee == 0
            && market.fee_ledger.pending_if_fee == 0
            && market.fee_ledger.pending_amm_provision == 0)
    {
        return Ok((0, 0, 0));
    }
    let pnl_pool_tokens = get_token_amount(
        market.pnl_pool.balance(),
        spot_market,
        market.pnl_pool.balance_type(),
    )?;
    // live user claims stay fully backed by every drain; the buffer is a
    // retention margin on top that only the IF and AMM-provision drains
    // respect — the protocol drain is exempt and goes first (its per-settle
    // cadence keeps each drain small, and unlike the other two its value is
    // not recoverable later anyway)
    let reserved_claims: u128 = net_user_pnl.max(0).cast::<u128>()?;
    [...]
}
```

Fee sweeps route only pending fee-ledger counters

The delisting path reinforces the misallocation. Before moving an expired market to `Delisted`, `handle_settle_expired_market_pools_to_revenue_pool` performs one forced fee sweep and then transfers the remaining PnL-pool balance, together with the AMM fee-pool amount, into the quote spot revenue pool. Because the expired-position closeout fee was not booked into the

fee ledger, the forced sweep cannot split it into the protocol, insurance-fund, and AMM-provision destinations, so it remains part of the residual PnL-pool balance handled by this final drain.

As a result, closeout fees paid during expiry settlement bypass the configured perp fee-routing model. The protocol fee pool and AMM fee pool may receive less than intended, while the spot revenue pool and insurance fund may receive more, with the discrepancy scaling with expired open interest and the applicable taker-fee tier. The affected branch is reachable through the `SettlePNL` account context with an unconstrained signer authority, and the expired-position branch uses the supplied user account rather than requiring the signer to match that user.

Remediation

Route the expired-position closeout fee through the same fee-ledger accounting used by ordinary perp fills. In `settle_expired_position`, compute the protocol, insurance-fund, and AMM-provision portions for the closeout fee and call `fee_ledger.accrue_fill_fees` with the gross taker fee and those carveouts before the market can be swept or delisted. Alternatively, implement an equivalent expired-closeout split that increments `pending_protocol_fee`, `pending_if_fee`, and `pending_amm_provision` consistently with the configured fee structure.

Patch

Resolved in PR#273.

Revenue-Share Settlement Credits Any Subaccount of the Recipient Authority

ID	Severity	Status
OS-VEP-ADV-70	Medium	Resolved

Description

`load_revenue_share_map` accepts revenue-share recipient `User` accounts from the remaining-account tail and indexes each writable account by the `authority` stored inside the account data. As shown in the recipient-loading path, the checks performed before insertion cover the account discriminator, writability, data length, and account loader conversion, but they do not bind the supplied `User` to subaccount 0 or compare the account key against a canonical `(authority, 0)` user PDA.

programs/velocity/src/state/revenue_share_map.rs

```rust
pub fn load_revenue_share_map<'a: 'b, 'b>(
    account_info_iter: &mut Peekable<Iter<'a, AccountInfo<'b>>>,
) -> VelocityResult<RevenueShareMap<'b>> {
    [...]
    while let Some(account_info) = account_info_iter.peek() {
        [...]
        if account_discriminator == user_discriminator {
            let user_account_info = account_info_iter.next().safe_unwrap()?;
            let is_writable = user_account_info.is_writable;
            if !is_writable {
                return Err(ErrorCode::UserWrongMutability);
            }

            // Extract authority from User account data (after discriminator)
            let data = user_account_info
                .try_borrow_data()
                .or(Err(ErrorCode::CouldNotLoadUserData))?;
            let expected_data_len = User::SIZE;
            if data.len() < expected_data_len {
                return Err(ErrorCode::CouldNotLoadUserData);
            }
            let authority_slice = array_ref![data, 8, 32];
            let authority = Pubkey::from(*authority_slice);

            let user_account_loader: AccountLoader<User> =
                AccountLoader::try_from(user_account_info)
                    .or(Err(ErrorCode::InvalidUserAccount))?;

            revenue_share_map.insert_user(authority, user_account_loader)?;
            continue;
```

```
        }
        [...]
    }
```

Revenue-share user accounts are inserted by stored authority.

The resulting `RevenueShareMap` lookup API retrieves a mutable `User` solely from the entry stored under a `Pubkey` authority. In `get_user_ref_mut`, once an account has been inserted for an authority, later settlement code receives the loaded `User` for that authority without an additional `sub_account_id` or PDA check at lookup time.

`sweep_completed_revenue_share_for_market` uses that authority-only lookup when settling accrued referral or builder fees. In the builder path, the function obtains the builder authority from the escrow, loads the corresponding `User` and `RevenueShare` entries from `RevenueShareMap`, transfers the accrued quote balance into the loaded user's quote spot position, updates total builder rewards, and emits the loaded user's `sub_account_id`.

`handle_initialize_user` separately enforces that an onboarding referrer must be subaccount 0, establishing subaccount 0 as the expected referrer account in that flow. Settlement does not reapply that invariant when it consumes the remaining accounts, so a caller able to invoke settlement with arbitrary remaining accounts can supply a writable sibling `User` account under the same stored authority and cause the reward credit to land on that sibling subaccount instead of the canonical subaccount-0 recipient.

programs/velocity/src/instructions/user.rs

```
pub fn handle_initialize_user<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    // Only try to add referrer if it is the first user
    if user_stats.number_of_sub_accounts_created == 0 {
        let (referrer, referrer_stats) =
        get_referrer_and_referrer_stats(remaining_accounts_iter)?;
        let referrer = if let (Some(referrer), Some(referrer_stats)) = (referrer,
        referrer_stats) {
            let referrer = load!(referrer)?;
            let mut referrer_stats = load_mut!(referrer_stats)?;
            validate!(referrer.sub_account_id == 0, ErrorCode::InvalidReferrer)?;
            [...]
        }[...]
    }[...]
}
```

Referrer initialization requires subaccount 0.

This misroutes revenue-share rewards within the same authority boundary. The supplied `User` must still carry the beneficiary authority used by the escrow lookup. However, the credited quote balance can be placed on the wrong account-level balance for that authority, including a noncanonical subaccount with different operational use, delegate handling, or deficit state, which

can break the reward-routing expectations used by referrer onboarding and emitted settlement records.

Remediation

When loading revenue-share recipient users, require the supplied account to be the canonical recipient for its stored authority before inserting it into `RevenueShareMap`. This can be done by asserting that the loaded `User` has `sub_account_id == 0`, or by deriving the expected `(authority, 0)` user PDA and comparing it to the supplied account key.

Patch

Resolved in PR#273.

Perp Bankruptcy Resolution Requires Open Interest Even When No Loss Is Socialized

ID	Severity	Status
OS-VEP-ADV-71	Medium	Resolved

Description

The public `resolve_perp_bankruptcy` instruction routes through `handle_resolve_perp_bankruptcy`, loads the requested perp and quote spot market accounts, settles available revenue to the insurance fund, and then calls the liquidation controller with the current insurance-fund vault balance. The account context requires a signer authorized for the liquidator account and rejects self-liquidation, so the affected path is reachable through the normal bankruptcy-resolution flow for a separate bankrupt user.

Inside `resolve_perp_bankruptcy`, the controller first verifies the user is bankrupt with negative perp PnL, then applies the available coverage tranches before determining whether any loss remains to socialize. The function consumes pending insurance fees, draws from the shared insurance-fund vault subject to the market claim and vault balance, and then uses the AMM fee provision when losses remain. After these steps, it computes `loss_to_socialize`, but calls `calculate_funding_rate_deltas_to_resolve_bankruptcy` before checking whether `loss_to_socialize` is negative and before mutating funding rates for socialized loss.

The funding-delta helper enforces a nonzero total base asset amount before doing its funding-rate calculation. That precondition is appropriate when a remaining loss must be distributed across open perp exposure, but it is also reached when the earlier tranches fully cover the bankruptcy and `loss_to_socialize` is zero. In a zero-open-interest market, the helper returns `CantResolvePerpBankruptcy` even though no funding-rate delta is needed.

programs/velocity/src/math/liquidation.rs

```rust
pub fn calculate_funding_rate_deltas_to_resolve_bankruptcy(
    loss: i128,
    market: &PerpMarket,
) -> VelocityResult<i128> {
    let total_base_asset_amount = market
        .base_asset_amount_long
        .abs()
        .safe_add(market.base_asset_amount_short.abs())?;
    validate!(
        total_base_asset_amount != 0,
        ErrorCode::CantResolvePerpBankruptcy,
        "Cant resolve perp bankruptcy when total base asset amount is 0"
    )?;
    loss.abs()
        .safe_mul(AMM_RESERVE_PRECISION_I128)?
        .safe_div_ceil(total_base_asset_amount)?
```

```
311             .safe_mul(FUNDING_RATE_TO_QUOTE_PRECISION_PRECISION_RATIO.cast()?)
312     }
```

Funding-delta helper rejects zero open interest

As a result, a fully covered perp bankruptcy can revert solely because the market has no remaining long or short base asset amount. This creates a liveness failure in bad-debt cleanup: the bankrupt account may remain unresolved even when the designated insurance and fee-pool balances are sufficient to clear the loss without socialization.

Remediation

Move the call to `calculate_funding_rate_deltas_to_resolve_bankruptcy` inside the `loss_to_socialize < 0` branch, and use a zero funding-rate delta when `loss_to_socialize` is zero. This keeps the helper's zero-open-interest rejection limited to cases where a nonzero residual loss actually needs to be socialized across surviving perp exposure.

Patch

Resolved in PR#152.

Perp Bankruptcy Socialized Funding Omits Net Unsettled Seed

ID	Severity	Status
OS-VEP-ADV-72	Medium	Resolved

Description

When `resolve_perp_bankruptcy` reaches the socialization tranche, the remaining bankrupt perp loss is converted into opposite long and short cumulative funding-rate deltas. In `resolve_perp_bankruptcy`, that branch records the social loss and updates the market funding-rate accumulators, but it does not seed `market.net_unsettled_funding_pnl` for the aggregate funding liability that those new deltas create.

programs/velocity/src/controller/liquidation.rs

```rust
pub fn resolve_perp_bankruptcy([...]) -> VelocityResult<u64> {
    [...]
    let loss_to_socialize = losses_remaining.safe_add(amm_tranche_payment)?;
    validate!(
        loss_to_socialize <= 0,
        ErrorCode::InvalidPerpPositionToLiquidate,
        "loss_to_socialize must be non-positive"
    )?;
    let cumulative_funding_rate_delta = if loss_to_socialize < 0 {
        calculate_funding_rate_deltas_to_resolve_bankruptcy(
            loss_to_socialize,
            perp_market_map.get_ref(&market_index)?.deref(),
        )?
    } else {
        0
    };
    // socialize loss
    if loss_to_socialize < 0 {
        let mut market = perp_market_map.get_ref_mut(&market_index)?;
        market.total_social_loss = market
            .total_social_loss
            .safe_add(loss_to_socialize.unsigned_abs())?;
        market.cumulative_funding_rate_long = market
            .cumulative_funding_rate_long
            .safe_add(cumulative_funding_rate_delta)?;
        market.cumulative_funding_rate_short = market
            .cumulative_funding_rate_short
            .safe_sub(cumulative_funding_rate_delta)?;
    }
    [...]
}
```

Bankruptcy socialization updates funding rates without seeding unsettled funding PnL

The normal funding-settlement path accounts for that liability only as individual users settle against the updated cumulative funding rates. In `settle_funding_payment`, a user's funding payment updates the user's perp position and then adjusts `market.net_unsettled_funding_pnl`, so the market-level accumulator is corrected gradually rather than at the moment the socialized funding deltas are written.

programs/velocity/src/controller/funding.rs

```
pub fn settle_funding_payment(
    user: &mut User,
    user_key: &Pubkey,
    market: &mut PerpMarket,
    now: UnixTimestamp,
) -> VelocityResult {
    [...]
    if amm_cumulative_funding_rate
        != user.perp_positions[position_index]
            .last_cumulative_funding_rate
            .cast()?
    {
        [...]
        market_position.last_cumulative_funding_rate =
        amm_cumulative_funding_rate.cast()?;
        update_quote_asset_and_break_even_amount(market_position, market,
        market_funding_payment)?;
        market.net_unsettled_funding_pnl = market
            .net_unsettled_funding_pnl
            .safe_sub(market_funding_payment)?;
    }
    Ok(())
}
```

Funding settlement later adjusts the market unsettled funding accumulator

That delayed accounting matters because aggregate perp PnL includes the market's `quote_asset_amount` together with `net_unsettled_funding_pnl`. The `calculate_net_user_pnl` helper adds the unsettled funding accumulator into the market cost basis, so any missing socialized funding seed can make the aggregate user PnL calculation stale until affected users settle funding.

The public insurance-fund deficit resolver uses that aggregate calculation when deciding whether the market PnL pool is insufficient and again when computing excess user PnL imbalance. In `resolve_perp_pnl_deficit`, the withdrawal sizing path reads `market.net_unsettled_funding_pnl`. If this value has not yet incorporated the socialized bankruptcy funding liability, the resolver can size insurance-fund support from an overstated aggregate user PnL value.

If the stale aggregate can be exercised before funding settlement catches up, the market may receive insurance-fund support for a PnL imbalance that would be smaller or absent once the socialized funding liability is reflected.

Remediation

When `resolve_perp_bankruptcy` writes the bankruptcy socialization funding-rate deltas, update `market.net_unsettled_funding_pnl` in the same socialization branch by the aggregate socialized loss amount that future funding settlements will realize. This seeds the market-level accumulator immediately, keeping `calculate_net_user_pnl` and `resolve_perp_pnl_deficit` consistent before individual users settle funding.

Patch

Resolved in 6bc8709.

Swap-Backed Spot Liquidation Fees Bypass The Margin-Shortage Cap

ID	Severity	Status
OS-VEP-ADV-73	Medium	Resolved

Description

The spot liquidation flow has two lanes that can reduce a user's borrow: the direct `liquidate_spot` path and the external-swap-backed `liquidate_spot_with_swap_begin`/`liquidate_spot_with_swap_end` path. In the direct path, the insurance-fund and protocol fee budget is first passed through `calculate_spot_if_fee` using the tracked market margin shortage and the liability amount, then split IF-first between the insurance fund and protocol fee. This makes the fee applied to the user's borrow reduction margin-shortage-aware rather than simply equal to the market's configured raw fee rates.

programs/velocity/src/controller/liquidation.rs

```
pub fn liquidate_spot([...]) -> VelocityResult {
    [...]
    // if + protocol rates, split IF-first (see liquidate_perp for rationale)
    let (liability_if_liquidation_fee, liability_protocol_liquidation_fee) = {
        let liability_market = spot_market_map.get_ref(&liability_market_index)?;
        (
            liability_market.if_liquidation_fee,
            liability_market.protocol_liquidation_fee,
        )
    };
    let total_if_side_fee = calculate_spot_if_fee(
        intermediate_margin_calculation.tracked_market_margin_shortage(margin_shortage)?,
        liability_amount,
        asset_weight,
        asset_liquidation_multiplier,
        liability_weight_with_buffer,
        liability_liquidation_multiplier,
        liability_decimals,
        liability_price,
        liability_if_liquidation_fee.safe_add(liability_protocol_liquidation_fee)?,
    )?;
    let liquidation_if_fee = total_if_side_fee.min(liability_if_liquidation_fee);
    let liquidation_protocol_fee = total_if_side_fee.safe_sub(liquidation_if_fee)?;

    [...]
}
```

Direct spot liquidation computes a margin-shortage-aware combined fee before splitting it.

The swap-backed path does not carry that same capped fee into `liquidate_spot_with_swap_end`. At swap end, the function reads the liability market's `if_liquidation_fee` and

`protocol_liquidation_fee` directly alongside the oracle price, decimals, and liquidation multiplier, so the later accounting uses the configured rates instead of a value derived from `calculate_spot_if_fee`.

programs/velocity/src/controller/liquidation.rs

```
pub fn liquidate_spot_with_swap_end([...]) -> VelocityResult {
    [...]
    let (
        liability_price,
        liability_decimals,
        liability_liquidation_multiplier,
        liquidation_if_fee,
        liquidation_protocol_fee,
    ) = {
        let liability_market = spot_market_map.get_ref_mut(&liability_market_index)?;
        let (liability_price_data, _validity_guard_rails) =
            oracle_map.get_price_data_and_guard_rails(&liability_market.oracle_id())?;

        let liability_price = liability_price_data.price;

        (
            liability_price,
            liability_market.decimals,
            calculate_liquidation_multiplier(
                liability_market.liquidator_fee,
                LiquidationMultiplierType::Discount,
            )?,
            liability_market.if_liquidation_fee,
            liability_market.protocol_liquidation_fee,
        )
    };
    [...]
}
```

Swap-end reads the liability market's raw liquidation fee rates.

Those raw rates are then multiplied by the measured `liability_transfer`, and both resulting fees are subtracted from `user_liability_reduction` before the user's borrow is reduced. The same block routes the IF fee into the liability market revenue pool and the protocol fee into the protocol fee pool, so any amount above the margin-shortage-aware cap becomes fee income rather than borrow relief for the liquidated user.

programs/velocity/src/controller/liquidation.rs

```rust
pub fn liquidate_spot_with_swap_end([...]) -> VelocityResult {
    [...]
    let if_fee = liability_transfer
        .cast::<u128>()?
        .safe_mul(liquidation_if_fee.cast()?)?
        .safe_div(LIQUIDATION_FEE_PRECISION_U128)?;
    let protocol_fee = liability_transfer
        .cast::<u128>()?
        .safe_mul(liquidation_protocol_fee.cast()?)?
        .safe_div(LIQUIDATION_FEE_PRECISION_U128)?;
    {
        let mut liability_market =
        spot_market_map.get_ref_mut(&liability_market_index)?;
        let user_liability_reduction = liability_transfer
            .cast::<u128>()?
            .safe_sub(if_fee)?
            .safe_sub(protocol_fee)?;
        update_spot_balances_and_cumulative_deposits([...])?;
        update_revenue_pool_balances(if_fee, &SpotBalanceType::Deposit, &mut
        liability_market)?;
        update_protocol_fee_pool_balances(
            protocol_fee,
            &SpotBalanceType::Deposit,
            &mut liability_market,
            false,
        )?;
    }
    [...]
}
```

Swap-end applies the raw rates to the transfer and subtracts them from borrow relief.

As a result, when the raw `if_liquidation_fee` plus `protocol_liquidation_fee` exceeds the shortage-aware fee that the direct path would compute for the same liability amount, a public liquidator using the begin/swap/end lane can deliver less borrow relief than the direct path.

Remediation

Reuse the direct path's `calculate_spot_if_fee` logic for swap-backed spot liquidations. Compute the margin-shortage-aware combined IF-plus-protocol fee during swap-begin sizing, persist or otherwise re-derive the same capped fee in swap-end, and split that capped amount IF-first before applying `if_fee` and `protocol_fee` to the liability transfer.

Patch

Resolved in PR#408.

Cross-Bankruptcy Resolver Ordering Allocates Limited Quote IF

ID	Severity	Status
OS-VEP-ADV-74	Medium	Resolved

Description

The cross-margin bankruptcy predicate treats both quote spot borrows and negative non-isolated perp quote PnL as bankrupting liabilities, while rejecting users that still have deposits, positive perp quote PnL, base exposure, or open orders. In `is_cross_margin_bankrupt`, a user can therefore remain in cross-margin bankruptcy with multiple liability types outstanding at the same time, including a quote borrow and a negative perp PnL balance.

The perp and spot bankruptcy paths are exposed as separate program instructions and use the same `ResolveBankruptcy` account context. The perp handler shown in `handle_resolve_perp_bankruptcy` is gated by `solvency_repair_not_paused`. The surrounding account constraints require signer authority for the liquidator account, but do not impose a deterministic ordering between this instruction and the corresponding spot bankruptcy instruction.

programs/velocity/src/instructions/keeper.rs

```
#[access_control(
    solvency_repair_not_paused(&ctx.accounts.state)
)]
pub fn handle_resolve_perp_bankruptcy<'c: 'info, 'info>(
    ctx: Context<'info, ResolveBankruptcy<'info>>,
    quote_spot_market_index: u16,
    market_index: u16,
) -> Result<()> {
    let clock = Clock::get()?;
    [...]
}
```

Each resolver computes its insurance-fund payment from the current token balance passed into the liquidation controller. In `resolve_perp_bankruptcy`, the shared insurance-fund payment is capped by the current `insurance_fund_vault_balance` after leaving one unit in the vault. The spot resolver read from the same controller file applies the same current-balance cap to the bankrupt borrow before socializing the remainder to depositors.

programs/velocity/src/controller/liquidation.rs

```
pub fn resolve_perp_bankruptcy([...]) -> VelocityResult<u64> {
    [...]
    let if_payment = {
        [...]
            .unsigned_abs()
            .min(insurance_fund_vault_balance.saturating_sub(1).cast()?)
```

```
                .min(max_insurance_withdraw);
            [...]
        };
        [...]
    }
```

When the quote insurance-fund vault cannot cover both the quote borrow and the negative perp PnL, the first resolver consumes the available insurance-fund capacity for its liability class and leaves less capacity for the second resolver. This makes the residual loss allocation order-dependent: resolving spot bankruptcy first can reduce depositor loss while leaving more perp loss to be socialized through funding, while resolving perp bankruptcy first can reduce perp socialization while leaving more spot bad debt for depositors. The total bad debt is not changed by the ordering, but a caller with exposure to one survivor class could prefer the order that protects that class.

Remediation

Resolve all bankrupt liabilities for a cross-margin user in a single deterministic bankruptcy allocation step before applying liability-specific socialization. The allocation should reserve the available quote insurance-fund balance across simultaneous quote spot borrow and quote perp PnL liabilities, for example with a pro-rata rule or another fixed priority that cannot be selected by instruction ordering.

Patch

Resolved in PR#408.

Protocol Sweep Can Unback Floored Pending IF Bankruptcy Tranche

ID	Severity	Status
OS-VEP-ADV-75	Medium	Resolved

Description

The fee sweep for a perp market materializes pending fee carveouts from the market PnL pool, and the permissionless keeper handler calls it after computing `net_user_pnl`. In `sweep_market_fees`, the protocol-fee leg runs first and reserves only positive aggregate user PnL before transferring tokens from `pnl_pool` to `protocol_fee_pool`, as shown in the excerpt. When bankrupt negative PnL offsets surviving positive PnL, this aggregate reservation can be zero even though surviving traders still have positive PnL claims that depend on the same pool tokens.

programs/velocity/src/controller/perp_pools.rs

```
pub fn sweep_market_fees([...]) -> VelocityResult<(u128, u128, u128)> {
    [...]
    // not recoverable later anyway)
    let reserved_claims: u128 = net_user_pnl.max(0).cast::<u128>()?;
    let mut available_unbuffered: u128 =
    pnl_pool_tokens.saturating_sub(reserved_claims);

    // 1. protocol's withdrawable cut (buffer-exempt: only user claims reserved)
    let protocol_drain = market
        .fee_ledger
        .pending_protocol_fee
        .min(available_unbuffered);
    if protocol_drain > 0 {
        transfer_spot_balances(
            protocol_drain.cast()?,
            spot_market,
            &mut market.pnl_pool,
            &mut market.protocol_fee_pool,
        )?;
        market.fee_ledger.consume_pending_protocol(protocol_drain)?;
        available_unbuffered = available_unbuffered.safe_sub(protocol_drain)?;
    }
    [...]
}
```

Protocol fees drain first while reserving only aggregate positive user PnL.

The same sweep later applies the bankruptcy IF floor only to the `pending_if_fee` counter. As shown in the IF-drain portion of `sweep_market_fees`, the drain subtracts `get_bankruptcy_if_floor()` from `pending_if_fee` to leave a standing tranche behind, while the available token balance used by the earlier protocol drain is not reduced by that floored tranche.

The floor therefore preserves accounting capacity for bankruptcy resolution, but does not separately reserve the PnL-pool tokens that make the counter-only tranche economically backed.

programs/velocity/src/controller/perp_pools.rs

```rust
pub fn sweep_market_fees([...]) -> VelocityResult<(u128, u128, u128)> {
    [...]
    let bankruptcy_if_floor = if force {0} else {market.get_bankruptcy_if_floor()?};
    let if_drain = market
        .fee_ledger
        .pending_if_fee
        .saturating_sub(bankruptcy_if_floor)
        .min(available);
    if if_drain > 0 {[...]}
    [...]
}
```

The bankruptcy IF floor is applied only to the pending IF counter.

During perp bankruptcy resolution, `resolve_perp_bankruptcy` consumes `pending_if_fee` before drawing from the shared insurance fund or socializing loss. This first tranche reduces the pending IF counter and the remaining loss, but performs no token transfer into the PnL pool.

programs/velocity/src/controller/liquidation.rs

```rust
pub fn resolve_perp_bankruptcy([...]) -> VelocityResult<u64> {
    [...]
    // pnl pool backing the counterparties this spares from socialization.
    let pending_if_payment: u128 = {
        let mut perp_market = perp_market_map.get_ref_mut(&market_index)?;
        let pending_if_payment = loss
            .unsigned_abs()
            .min(perp_market.fee_ledger.pending_if_fee);
        if pending_if_payment > 0 {
            perp_market
                .fee_ledger
                .consume_pending_if(pending_if_payment)?;
            msg!("bankruptcy pending_if_fee tranche: {}", pending_if_payment);
        }
        pending_if_payment
    };
    [...]
}
```

Bankruptcy consumes pending IF fees without transferring pool tokens.

Together, these paths create a priority inversion between the protocol-fee pool and surviving perp PnL claimants. If a public sweep runs while aggregate `net_user_pnl` is low or zero because a

bankrupt loss offsets surviving positive PnL, protocol fees can be swept out of the PnL pool while the floored `pending_if_fee` counter remains. A later bankruptcy can then consume that counter, record no social loss for the covered amount, and leave the surviving positive PnL claims under-backed by the remaining pool tokens.

Remediation

Reserve token backing for counter-only bankruptcy tranches before computing the protocol-fee drain. At minimum, reduce protocol-drain availability by the amount of `pending_if_fee` that bankruptcy may consume without token movement, capped by the active bankruptcy IF floor, and include any analogous unresolved `pending_amm_provision` amount that can be consumed counter-only. Alternatively, move bankruptcy-reserved tranches ahead of the protocol leg in the sweep waterfall so protocol fees can only be swept from tokens not needed to back surviving PnL after counter-only bankruptcy coverage.

Patch

Resolved in PR#392.

Equity Breaker Does Not Block Trigger Orders

ID	Severity	Status
OS-VEP-ADV-76	Medium	Resolved

Description

A public keeper can trigger a dormant perp order and collect the keeper reward for a healthy subaccount while its authority's equity breaker is tripped, because the trigger path has no `UserStats` account and checks only the local floor. The equity floor breaker is stored on `UserStats` as an authority-wide flag, while a trigger order operates on an individual `User`. In the `keeper` instruction, `handle_trigger_order` loads the target order and calls the order controller with the target `user` and the `filler`, but the `TriggerOrder` account context does not include a target `UserStats` account. As shown in `TriggerOrder`, the handler therefore has no account from which it can read `is_equity_breaker_tripped()` for the authority whose order is being activated.

programs/velocity/src/instructions/keeper.rs

```
3220 pub struct TriggerOrder<'info> {
3221     pub state: AccountLoader<'info, State>,
3222     pub authority: Signer<'info>,
3223     #[account(
3224         mut,
3225         constraint = can_sign_for_user(&filler, &authority)?
3226     )]
3227     pub filler: AccountLoader<'info, User>,
3228     #[account(mut)]
3229     pub user: AccountLoader<'info, User>,
3230 }
```

Trigger order context omits target UserStats

The trigger controller mutates the dormant trigger order into its triggered state, updates the order auction-related state in the surrounding code, and pays the flat keeper reward before the post-trigger risk gate. The check shown in `trigger_order` only evaluates whether the activation increased worst-case liability and whether the local user fails initial margin or is below that user's own equity floor, it does not check the shared `UserStats` breaker flag.

programs/velocity/src/controller/orders.rs

```
1 pub fn trigger_order([...]) -> VelocityResult {
2     [...]
3     // If order increases risk and user is below initial margin or their equity floor,
      cancel it
4     if is_risk_increasing && !user.orders[order_index].reduce_only {
5         let margin_calc =
          calculate_margin_requirement_and_total_collateral_and_liability_info(
6             user,
```

```
            perp_market_map,
            spot_market_map,
            oracle_map,
            MarginContext::standard(MarginRequirementType::Initial),
        )?;

        if !margin_calc.meets_margin_requirement()
            || user.is_below_equity_floor(margin_calc.total_collateral)
        {
            cancel_order(
                order_index,
                user,
                &user_key,
                perp_market_map,
                spot_market_map,
                oracle_map,
                now,
                slot,
                OrderActionExplanation::InsufficientFreeCollateral,
                Some(&filler_key),
                0,
                false,
            )?;
        }
    }[...]
}
```

Post-trigger gate checks local margin and equity floor only

As a result, when one subaccount has tripped the authority-wide breaker, a keeper can still trigger a dormant perp order on another same-authority subaccount that is locally healthy. Fill-time checks continue to honor the breaker for risk-increasing fills, so the impact is limited to bypassing the intended freeze for trigger-order state transitions: the keeper can collect the bounded flat trigger reward and convert the order into a triggered, auction-aged order instead of preserving the dormant pre-breaker state.

Remediation

Add the target `UserStats` account to the `TriggerOrder` context and constrain it with `is_stats_for_user(&user, &user_stats)` so the trigger handler can read the authority-wide breaker state for the order owner. Before mutating the order or paying the keeper reward, reject risk-increasing trigger activation when `user_stats.is_equity_breaker_tripped()` is true, while preserving existing local margin and equity-floor checks.

Patch

Resolved in PR#393.

Delegate Floor Delta Can Move Admin Floor Without Funds

ID	Severity	Status
OS-VEP-ADV-77	Medium	Resolved

Description

An account owner can shift a warm-admin equity floor off a breached subaccount with a zero-amount delegate transfer, dropping the floor before the permissionless breaker can trip and re-enabling risk-increasing actions. `handle_transfer_deposit_by_delegate` accepts both `amount` and `equity_floor_delta`, but applies the floor movement independently of the token movement. Once delegate transfers are allowed and the authority-level breaker flag is not already set, any positive `equity_floor_delta` is moved from `from_user` to `to_user` before `transfer_spot_deposit` is called, and the code does not require `amount > 0` or otherwise bind the delta to the quote value of the transferred deposit.

The lower-level transfer path then processes the supplied `amount` as the spot transfer amount and performs the withdraw-side margin checks after the floor has already been reduced on the source subaccount. The recipient is checked after the transfer only to ensure that its increased floor is backed by its own strict total collateral, so the aggregate floor can be preserved while the breached source subaccount's floor is reduced without a corresponding transfer of funds.

The same handler only rejects delegate transfers when `UserStats` already has `equity_breaker_tripped` set, as shown by the breaker check in `handle_transfer_deposit_by_delegate`. It does not first prove whether the source subaccount is currently below its admin-assigned floor, leaving a pre-trip window in which an account owner with delegate transfer permission can move floor away before `trip_equity_floor_breaker` observes the breach.

programs/velocity/src/instructions/user.rs

```
pub fn handle_transfer_deposit_by_delegate<'c: 'info, 'info>([...]) ->
anchor_lang::Result<()> {
    [...]
    validate!(
        !user_stats.is_equity_breaker_tripped(),
        ErrorCode::EquityBelowFloor,
        "equity floor breaker is tripped for this authority"
    )?;
    [...]
}
```

Breaker gate only checks the already-tripped authority flag

This weakens the warm-admin per-subaccount floor before the permissionless breaker enforces it. Because the breaker trip path requires a subaccount with a positive floor that is below its floor, lowering the breached subaccount's floor can prevent that proof from succeeding. Ordinary risk-

increasing order, withdrawal, and transfer checks then continue to rely on the reduced floor or the not-yet-tripped authority flag.

Remediation

Reject any positive `equity_floor_delta` in `handle_transfer_deposit_by_delegate` unless the transfer moves a strictly positive amount of value, and enforce an on-chain bound tying the allowed floor delta to the strict quote value actually transferred. For quote-market transfers, cap `equity_floor_delta` at `amount` for non-quote markets, either compute a strict oracle-denominated value before allowing the floor movement or disallow explicit floor movement through this delegate transfer path. If the equity floor is intended to remain an admin-controlled risk boundary rather than portable collateral metadata, remove floor rebalancing from delegate transfers and expose it only through the warm-admin equity-floor update flow.

Patch

Resolved in PR#393.

Equity Breaker Does Not Block Generic Spot Swaps

ID	Severity	Status
OS-VEP-ADV-78	Medium	Resolved

Description

The generic spot swap flow does not enforce the authority-wide equity breaker before completing a swap. In `handle_end_swap`, the instruction loads both `User` and `UserStats`, but the visible gate at this point is only the exchange `WithdrawPaused/DepositPaused` state. The function then proceeds without rejecting a tripped `UserStats`.

programs/velocity/src/instructions/user.rs

```
pub fn handle_end_swap<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    let user_key = ctx.accounts.user.key();
    let mut user = load_mut!(&ctx.accounts.user)?;
    let mut user_stats = load_mut!(&ctx.accounts.user_stats)?;
    let exchange_status = state.get_exchange_status()?;
    validate!(
        !exchange_status.contains(ExchangeStatus::DepositPaused |
        ExchangeStatus::WithdrawPaused),
        ErrorCode::ExchangePaused
    )?;
    [...]
}
```

After the swap legs are reconciled, `handle_end_swap` records the input side as a borrow and the output side as a deposit when applicable, then selects the margin type for the resulting position change. The final risk check is a call to `user.meets_withdraw_margin_requirement_swap`, so the swap path is gated by the user's resulting margin state rather than by the shared breaker flag.

programs/velocity/src/instructions/user.rs

```
pub fn handle_end_swap<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    user.meets_withdraw_margin_requirement_swap(
        &perp_market_map,
        &spot_market_map,
        &mut oracle_map,
        margin_type,
    )?;
    [...]
}
```

The margin helper used by the swap path calculates margin and collateral for the local `User`, checks oracle validity for liabilities, validates isolated-tier requirements, enforces the ordinary margin requirement, and checks `self.equity_floor`. It does not inspect `UserStats` or call `is_equity_breaker_tripped`, so it cannot enforce a breaker that is stored at the authority-wide stats level.

programs/velocity/src/state/user.rs

```
pub fn meets_withdraw_margin_requirement_swap(
    &mut self,
    perp_market_map: &PerpMarketMap,
    spot_market_map: &SpotMarketMap,
    oracle_map: &mut OracleMap,
    margin_requirement_type: MarginRequirementType,
) -> VelocityResult<bool> {
    [...]
    validate!(
        !self.is_below_equity_floor(calculation.total_collateral),
        ErrorCode::EquityBelowFloor,
        "total collateral {} below equity floor {}",
        calculation.total_collateral,
        self.equity_floor
    )?;
    Ok(true)
}
```

This leaves a path where an authority whose breaker has been tripped by one subaccount can still use a healthy sibling subaccount to complete a generic spot swap, provided the post-swap position satisfies the ordinary margin and local equity-floor checks. This results in a bypass of the intended authority-wide freeze for this risk-importing path, allowing new spot borrow and deposit balances to be booked under normal margin constraints.

Remediation

Reject generic spot swaps when the loaded `UserStats` has `is_equity_breaker_tripped()` set before `end_swap` books any new borrow or deposit balances, unless the implementation explicitly proves the swap is strictly reducing the relevant risk on both legs.

Patch

Resolved in PR#393.

Transfer Perp Position Skips Recipient Equity Floor

ID	Severity	Status
OS-VEP-ADV-79	Medium	Resolved

Description

`transfer_perp_position` is exposed through the program entrypoint and handled by `handle_transfer_perp_position`, which transfers a perpetual position between two signed-for user accounts associated with the same `UserStats`. After settling funding and applying the sender and recipient position deltas at the standardized oracle price, the handler evaluates the recipient with an initial-margin `MarginContext` and only requires that the resulting margin calculation meets the margin requirement, as shown in the recipient-side check.

programs/velocity/src/instructions/user.rs

```
pub fn handle_transfer_perp_position<'c: 'info, 'info>([...]) ->
anchor_lang::Result<()> {
    [...]
    let to_user_margin_context =
    MarginContext::standard(MarginRequirementType::Initial);
    let to_user_margin_requirement =
        calculate_margin_requirement_and_total_collateral_and_liability_info(
            to_user,
            &perp_market_map,
            &spot_market_map,
            &mut oracle_map,
            to_user_margin_context,
        )?;

    validate!(
        to_user_margin_requirement.meets_margin_requirement(),
        ErrorCode::InsufficientCollateral,
        "to user margin requirement is greater than total collateral"
    )?;
    [...]
}
```

Recipient transfer check only enforces initial margin

The sender receives an additional post-transfer equity-floor validation after its maintenance-margin calculation. The adjacent sender-side check rejects when `from_user.is_below_equity_floor(...)` is true, which makes the missing recipient-side equivalent asymmetric rather than an absence of equity-floor enforcement throughout the handler.

programs/velocity/src/instructions/user.rs

```rust
pub fn handle_transfer_perp_position<'c: 'info, 'info>([...]) ->
anchor_lang::Result<()> {
    [...]
    validate!(
        !
        from_user.is_below_equity_floor(from_user_margin_calculation.total_collateral),
        ErrorCode::EquityBelowFloor,
        "from user total collateral {} below equity floor {}",
        from_user_margin_calculation.total_collateral,
        from_user.equity_floor
    )?;
    [...]
}
```

Sender transfer check also enforces the equity floor

The `User` state defines `equity_floor` as a minimum cross-margin total collateral threshold and `is_below_equity_floor` returns true when a nonzero floor exceeds the computed total collateral. Other user margin checks in the same state module include this floor validation after confirming margin requirements, and the keeper path can trip an authority-wide breaker when a user is below its floor. In this transfer path, however, a recipient can end the transfer initial-margin compliant while below its configured floor because no `to_user.is_below_equity_floor(to_user_margin_requirement.total_collateral)` validation runs after the recipient calculation.

As a result, the recipient's configured floor does not act as an admission condition for this core perp exposure transfer path. For a same-authority caller, this can leave a floor-protected recipient subaccount in a below-floor state through an ordinary transfer while the transaction still succeeds.

Remediation

After calculating `to_user_margin_requirement`, add a recipient equity-floor validation that rejects when `to_user.is_below_equity_floor(to_user_margin_requirement.total_collateral)` is true, using `ErrorCode::EquityBelowFloor` and mirroring the sender-side postcondition.

Patch

Resolved in PR#393.

AMM-Only Limit Cap Uses A Different Reserve Basis Than Execution

ID	Severity	Status
OS-VEP-ADV-80	Medium	Resolved

Description

A taker's AMM-only perp limit order is capped by an analytical inverse that does not always use the same reserve basis as the later swap execution. In `calculate_base_asset_amount_to_trade_to_price`, the inverse calculation chooses the cached directional ask or bid reserve only when `base_spread` is nonzero. When `base_spread` is zero, it uses the raw `base_asset_reserve` instead. This makes the computed maximum fill size depend on raw reserves even when the AMM's cached directional reserves may differ from the raw reserve state.

programs/velocity/src/vlp/amm/math/spread.rs

```rust
pub fn calculate_base_asset_amount_to_trade_to_price(
    amm: &AMM,
    limit_price: u64,
    direction: PositionDirection,
) -> VelocityResult<(u64, PositionDirection)> {
    [...]
    let base_asset_reserve_before = if amm.base_spread > 0 {
        match direction {
            PositionDirection::Long => amm.ask_base_asset_reserve,
            PositionDirection::Short => amm.bid_base_asset_reserve,
        }
    } else {
        amm.base_asset_reserve
    };
    [...]
}
```

Inverse limit sizing falls back to raw reserves when base spread is zero.

The execution path prices the selected base amount differently. As shown in `calculate_base_swap_output`, the quote amount is derived from the cached bid or ask reserves selected by swap direction, and those cached reserves are refreshed from spread and reference-offset state. The function then separately computes the post-swap raw reserves for mutation, so the quoted taker payment can be based on directional reserves even when the inverse limit cap was based on raw reserves.

programs/velocity/src/vlp/amm/controller.rs

```rust
pub(crate) fn calculate_base_swap_output(
    amm: &AMM,
    base_asset_swap_amount: u64,
    direction: SwapDirection,
```

```
) -> VelocityResult<AmmSwapOutput> {
    // first do the swap with spread reserves to figure out how much base asset is acquired
    let (base_asset_reserve_with_spread, quote_asset_reserve_with_spread) = match direction {
        SwapDirection::Add => (amm.bid_base_asset_reserve, amm.bid_quote_asset_reserve),
        SwapDirection::Remove => (amm.ask_base_asset_reserve, amm.ask_quote_asset_reserve),
    };

    let (new_quote_asset_reserve_with_spread, _) = amm::calculate_swap_output(
        base_asset_swap_amount.cast()?,
        base_asset_reserve_with_spread,
        direction,
        amm.sqrt_k,
    )?;

    let quote_asset_amount = calculate_quote_asset_amount_swapped(
        quote_asset_reserve_with_spread,
        new_quote_asset_reserve_with_spread,
        direction,
        amm.peg_multiplier,
    )?;

    let (new_quote_asset_reserve, new_base_asset_reserve) = amm::calculate_swap_output(
        base_asset_swap_amount.cast()?,
        amm.base_asset_reserve,
        direction,
        amm.sqrt_k,
    )?;
    [...]
}
```

Forward swap quote calculation reads cached directional bid or ask reserves.

`fill_amm_only` treats the AMM's `cumulative_size` result as the sufficient taker-limit cap. After checking the current best price against the taker's limit, it caps `target_size` to the reported supply at the price cap, calls `try_fill_solo` for that effective target, and commits the fill. It does not perform a post-quote average-price validation against the taker's effective limit before `commit_fill`. Because the inverse cap and the executed quote can read different reserve bases when `base_spread == 0`, a public filler may be able to fill another user's AMM-only limit order at an average price worse than the taker's specified limit.

Remediation

Make `calculate_base_asset_amount_to_trade_to_price` use the same cached directional reserve basis as `calculate_base_swap_output` for the relevant long or short side, including when

`base_spread == 0`. In addition, after `try_fill_solo` returns the AMM-only quote and before `commit_fill`, validate the resulting average fill price against the taker's effective limit price for both long and short orders so mismatched reserve states cannot bypass the user's limit.

Patch

Resolved in PR#269.

Fill-Triggered Funding Uses Pre-Fill Reserve For Divergence Gate

ID	Severity	Status
OS-VEP-ADV-81	Medium	Resolved

Description

`fill_perp_order` snapshots the AMM reserve price before fulfillment, then performs the fill and attempts a funding update at the end of the trade. Because fulfillment occurs between the snapshot and the funding call, an AMM or AMM-JIT fill can change the market reserves before funding is evaluated, while the funding call still receives the earlier reserve value, as shown in `fill_perp_order`.

programs/velocity/src/controller/orders.rs

```
pub fn fill_perp_order([...]) -> VelocityResult<(u64, u64)>
    [...]
    {
        [...]
        controller::funding::update_funding_rate(
            market_index,
            market,
            oracle_map,
            now,
            slot,
            &state.oracle_guard_rails,
            funding_paused,
            Some(reserve_price_before),
        )?;
    }[...]
}
```

Pre-fill reserve passed into the end-of-trade funding update

`update_funding_rate` accepts that precomputed reserve price and uses it when evaluating whether funding should be blocked. The function refreshes the AMM for the funding gate, but the divergence check is still supplied with the previously selected reserve price; when called from `fill_perp_order`, that value is the pre-fill reserve rather than the reserve after fulfillment.

programs/velocity/src/controller/funding.rs

```
pub fn update_funding_rate([...]) -> VelocityResult<bool> {
    let reserve_price = match precomputed_reserve_price {
        Some(reserve_price) => reserve_price,
        None => market.amm.reserve_price()?,
    };
    refresh_amm_for_funding_gate(market, oracle_map, slot, guard_rails)?;
    // Pause funding if oracle is invalid or if mark/oracle spread is too divergent
```

```
    let block_funding_rate_update = oracle::block_operation(
        market,
        oracle_map.get_price_data(&market.oracle_id())?,
        guard_rails,
        reserve_price,
        slot,
    )?;
    [...]
}
```

Funding gate using the supplied reserve price

The funding-rate calculation later derives its mark inputs from the AMM through a fresh quoter and computes bid and ask prices from the current AMM state. This makes the safety gate and the rate calculation depend on different reserve states within the same instruction: the gate can be evaluated against the pre-fill reserve, while the funding rate and mark TWAP inputs are derived from the post-fill AMM bid and ask path in `update_funding_rate`.

As a result, a fill timed at a funding boundary may allow a funding update to proceed when the post-fill mark would have failed the intended mark-oracle divergence guard. The extractable amount depends on trade sizing and the resulting funding delta.

Remediation

Do not pass the pre-fill reserve snapshot into the post-fill funding safety checks. After fulfillment and after any AMM refresh used by the funding gate, recompute `market.amm.reserve_price()` and use that current reserve for `oracle::block_operation` and oracle TWAP normalization. If the pre-fill reserve remains needed for separate fill accounting, keep it under a distinct parameter and prevent it from being used by the funding divergence gate.

Patch

Resolved in PR#269.

AMM Override Zero-Fill Suppresses Same-Timestamp Maker TWAP

ID	Severity	Status
OS-VEP-ADV-82	Medium	Resolved

Description

The perp fulfillment router can schedule an AMM override immediately before a maker match when the maker quote is worse than the current AMM quote for the taker direction. In `determine_perp_fulfillment_methods`, that override is inserted with the maker price and the local AMM comparison price is then advanced to that maker price before the actual `Match` step is pushed, as shown in the router excerpt.

programs/velocity/src/math/fulfillment.rs

```rust
pub fn determine_perp_fulfillment_methods([...]) ->
VelocityResult<Vec<PerpFulfillmentMethod>> {
    [...]
    for (maker_key, maker_order_index, maker_price) in maker_orders_info.iter() {
        [...]
        if amm_is_available {
            let maker_better_than_amm = match order.direction {
                PositionDirection::Long => *maker_price <= amm_price,
                PositionDirection::Short => *maker_price >= amm_price,
            };
            if !maker_better_than_amm {
                fulfillment_methods.push(PerpFulfillmentMethod::AMM(Some(*maker_price)))
                amm_price = *maker_price;
            }
        [...]
    }
    [...]
}
```

AMM override inserted before the maker match

`fulfill_perp_order` iterates the generated fulfillment methods and calls `fulfill_perp_order_step` for each one with the same `now` value from the fill. Inside `fulfill_perp_order_step`, the market mark TWAP is updated before the selected fulfillment method is executed, and the trade-price hint is the maker price for `Match` steps or the AMM bid/ask-side price for AMM steps, as shown in the TWAP update excerpt.

programs/velocity/src/controller/orders.rs

```rust
pub fn fulfill_perp_order_step([...]) -> VelocityResult<(u64, u64, u64)> {
    [...]
    });
    market.market_stats.update_mark_twap_with_amm_bid_ask(
```

```
        amm_bid_price,
        amm_ask_price,
        amm_base_spread,
        amm_long_spread,
        amm_short_spread,
        now,
        Some(twap_trade_price),
        Some(taker_direction),
        sanitize_clamp_denom,
        order_tick_size,
    )?;
    [...]
}
```

Mark TWAP update occurs before match execution

The AMM override step can still produce no fill. In `fill_amm_only`, the AMM take is capped by the AMM cumulative size at the effective price cap, and the function returns an empty match when the capped target is zero or the eventual fill has zero base, as shown in the AMM-only matching excerpt.

programs/velocity/src/controller/matching.rs

```
pub fn fill_amm_only(
    amm: &mut AmmQuoter,
    ctx: &QuoteContext,
    side: PositionDirection,
    target_size: u64,
    taker_limit_price: Option<u64>,
) -> VelocityResult<Match> {
    [...]
    let supply_at_cap = amm.cumulative_size(ctx, side, price_cap)?;
    let effective_target = target_size.min(supply_at_cap);
    if effective_target == 0 {
        return Ok(Match::empty());
    }

    let Some(fill) = amm.try_fill_solo(ctx, side, effective_target)? else {
        return Ok(Match::empty());
    };
    if fill.base_filled == 0 {
        return Ok(Match::empty());
    }
    [...]
}
```

AMM-only fill can return an empty match

Because the TWAP mutation happens before the match result is known, a zero-base AMM override can stamp the market TWAP timestamp without any corresponding base fill. The later maker match in the same fulfillment loop can execute at the same `now` timestamp, but its pre-match TWAP update has no elapsed time to move the TWAP meaningfully. The post-match code returns early on zero-base matches and accounts only nonzero fills, but the earlier market-stat update is not rolled back in the observed control flow.

As a result, a public filler can cause the funding-relevant mark TWAP to remain anchored to the pre-maker AMM quote while a real same-timestamp DLOB maker fill settles. This can suppress the maker trade price's effect on mark TWAP and may distort same-timestamp funding or risk logic that expects successful fills to be reflected in the mark TWAP. The condition is most relevant when the AMM cumulative size available at the override price rounds or standardizes to zero before the maker match.

Remediation

Move the mark-TWAP update in `fulfill_perp_order_step` until after the fulfillment method returns and a nonzero `total_base_filled` is known, using the actual executed fill context for the trade-price hint. Alternatively, explicitly skip or roll back market-stat updates when `fill_amm_only` or `match_take` returns an empty match.

Patch

Resolved in PR#269.

AMM Scalar Projection Ignores Market Min Order Size K-Down Floor

ID	Severity	Status
OS-VEP-ADV-83	Medium	Resolved

Description

The quote-time AMM projection path constructs a temporary `PerpMarket` from the live `AMM` plus only `market_status` and `market_config`, then passes that synthetic market into the same `project_post_refresh` pipeline used by the full refresh. As shown in `project_post_refresh_scalar`, the synthetic value is completed with `PerpMarket::default()`, so any market-level fields not copied into `ProjectionInputs` are not taken from the real market.

programs/velocity/src/vlp/amm/math/repeg.rs

```rust
pub fn project_post_refresh_scalar(
    amm: &AMM,
    inputs: &ProjectionInputs,
    mm_oracle_price_data: &MMOraclePriceData,
    oracle_validity: Option<OracleValidity>,
) -> VelocityResult<ProjectedAmmState> {
    let synthetic = PerpMarket {
        amm: *amm,
        status: inputs.market_status,
        market_config: inputs.market_config,
        ..PerpMarket::default()
    };
    project_post_refresh(&synthetic, mm_oracle_price_data, oracle_validity)
}
```

`project_post_refresh_scalar` builds a default-backed synthetic market.

`project_post_refresh` delegates the curve decision to `adjust_amm`, and `adjust_amm` uses `market.market_stats.min_order_size` when deciding whether formulaic k reduction is allowed and when computing the lower bound for `sqrt_k`. The k-down branch shown in `adjust_amm` therefore depends on a market-level floor even when reached through the scalar projection interface.

programs/velocity/src/vlp/amm/math/repeg.rs

```rust
pub fn adjust_amm([...]) -> VelocityResult<(Box<PerpMarket>, i128)> {
    [...]
    } else {
        // use full budget peg
        let can_lower_k = market.amm.can_lower_k(market.market_stats.min_order_size)?;
        // equivalent to (but cheaper than) scaling down by .1%
        let adjustment_cost: i128 = if adjust_k
            && can_lower_k
```

```
            && !
            market.has_market_config_flag(MarketConfigFlag::DisableFormulaicKUpdate)
        {
            // always let protocol-owned sqrt_k be either least .1% of lps or the base
            amount / min order
            let new_sqrt_k_lower_bound = market
                .amm
                .get_lower_bound_sqrt_k(market.market_stats.min_order_size)?;
            [...]
    }
    [...]
}
```

`adjust_amm` reads `market_stats.min_order_size` for k-down gating.

`MarketStats::default()` sets `min_order_size` to `1`. Because the scalar projection does not carry the real market stats into its synthetic `PerpMarket`, the scalar path can evaluate the k-down floor against this default value instead of the market's configured `market_stats.min_order_size`.

This creates a divergence between paths that otherwise use the same AMM and oracle inputs. The keeper refresh calls `project_post_refresh` with the real `PerpMarket` through `snap_to_oracle`, while quote setup calls `project_post_refresh_scalar` before applying the projected state to the AMM. Under conditions where the configured min order size would block `can_lower_k`, the scalar path may still project and apply a lower `sqrt_k`, reducing AMM depth and changing fee or budget accounting relative to the canonical keeper refresh.

Remediation

Include the real `market_stats.min_order_size` in `ProjectionInputs` and populate it from `ProjectionInputs::from_market`, then set `market_stats.min_order_size` on the synthetic `PerpMarket` before calling `project_post_refresh`. Alternatively, change `adjust_amm` or the lower-k calculation to accept the min-order-size scalar directly so the scalar projection cannot fall back to `MarketStats::default()`.

Patch

Resolved in PR#269.

AMM Quoter Marks Affordability-Rejected Refreshes Fresh

ID	Severity	Status
OS-VEP-ADV-84	Medium	Resolved

Description

`AmmQuoter::setup` performs the fill-time AMM refresh projection before quote and fill reads use the AMM state. When the projection is not already current for the slot, it computes a post-refresh projection, applies it to the AMM, and then advances `last_update_slot` if the oracle is valid for low-risk AMM fills unless its local `suppress` predicate is true. In `AmmQuoter::setup`, that predicate is based on `projection.cost > 0 && !projection.applied`, so an unapplied projection with zero cost is still allowed to mark the AMM fresh for the current slot.

programs/velocity/src/vlp/amm/quoter.rs

```
fn setup(&mut self, ctx: &QuoteContext) -> VelocityResult<()> {
    [...]
    let projection_current = self.amm.last_update_slot >= ctx.slot;
    if !projection_current {
        [...]
        if let Some(validity) = ctx.oracle_validity {
            if crate::math::oracle::is_oracle_valid_for_action(
                validity,
                Some(crate::math::oracle::VelocityAction::FillOrderAmmLowRisk),
            )? {
                let suppress = projection.cost > 0 && !projection.applied;
                if !suppress {
                    self.amm.last_update_slot = ctx.slot;
                }
            }
        }
    }
    [...]
}
```

`AmmQuoter::setup` suppresses freshness only for positive-cost unapplied projections.

The affordability-rejected projection path returns exactly that zero-cost, unapplied state. In `project_post_refresh`, when the calculated AMM adjustment would debit more than the retained fee surplus permits, the function returns a passthrough `ProjectedAmmState` with the old peg, reserves, and `sqrt_k`, while setting `rejected_due_to_affordability` and leaving the inherited `cost` and `applied` values from the no-op state. This means `projection.apply_to` leaves the AMM curve unchanged, but the stale quoter predicate does not recognize the rejection.

programs/velocity/src/vlp/amm/math/repeg.rs

```rust
pub fn project_post_refresh(
    market: &PerpMarket,
    mm_oracle_price_data: &MMOraclePriceData,
    oracle_validity: Option<OracleValidity>,
) -> VelocityResult<ProjectedAmmState> {
    [...]
    if applied {
        Ok(ProjectedAmmState {[...]})
    } else {
        // Peg/reserves/cost stay at passthrough; flag the affordability
        // rejection so the keeper crank won't mark the AMM fresh.
        Ok(ProjectedAmmState {
            rejected_due_to_affordability: true, ..noop
        })
    }
}
```

`project_post_refresh` returns affordability rejections as zero-cost unapplied passthrough projections.

The keeper refresh path already uses the explicit affordability-rejection flag when deciding whether to advance freshness. In `snap_to_oracle`, `last_update_slot` is written only when the oracle is valid for low-risk fills and the projection was not rejected due to affordability. This creates inconsistent freshness semantics between the keeper refresh path and the fill-time quoter path for the same rejected projection result.

programs/velocity/src/vlp/amm/refresh.rs

```rust
pub fn snap_to_oracle([...]) -> VelocityResult<i128> {
    [...]
    if let Some(validity) = oracle_validity {
        if is_oracle_valid_for_action(validity,
        Some(VelocityAction::FillOrderAmmLowRisk))? {
            if !projection.rejected_due_to_affordability {
                market.amm.last_update_slot = slot;
            }
        }
    }
    Ok(projection.cost)
}
```

`snap_to_oracle` uses the explicit affordability-rejection guard.

`last_update_slot` is used by `is_fresh_at` as an exact same-slot freshness marker, and repository call sites use that marker in same-slot checks such as PnL settlement and market freshness validation. A fill-time setup that stamps `last_update_slot` after an affordability-rejected projection

can therefore make unchanged AMM curve state appear freshly projected for the slot. The impact is limited to freshness gates that rely on this marker.

Remediation

Update `AmmQuoter::setup` to mirror `snap_to_oracle` by advancing `last_update_slot` only when the oracle is valid for low-risk fills and `projection.rejected_due_to_affordability` is false.

Patch

Resolved in PR#269.

AMM JIT Match Fills Bypass The Per-Fill Reserve Throttle

ID	Severity	Status
OS-VEP-ADV-85	Medium	Resolved

Description

The AMM-only fulfillment path applies the market's per-fill AMM liquidity limit before routing the fill. In `fulfill_perp_order_with_method`, the standalone AMM branch validates the AMM, calculates available AMM liquidity for the taker direction, and passes the smaller of `target_size` and that available amount into the AMM-only matcher, as shown in the standalone AMM branch.

programs/velocity/src/controller/orders.rs

```
pub fn fulfill_perp_order_step([...]) -> VelocityResult<(u64, u64, u64)> {
    [...]
    let match_result = match method {
        PerpFulfillmentMethod::AMM(override_fill_price) => {
            [...]
            // and the AMM's hard reserve bound.
            let amm_available = {
                let amm_ref: &crate::vlp::amm::AMM = amm_quoter.amm;
                calculate_amm_available_liquidity(amm_ref, &taker_direction,
                order_step_size)?
            };
            crate::controller::matching::fill_amm_only(
                &mut amm_quoter,
                &ctx,
                taker_direction,
                target_size.min(amm_available),
                effective_taker_limit,
            )?
        }
        [...]
    };
    [...]
}
```

Standalone AMM fulfillment caps the requested target by available AMM liquidity.

The match fulfillment path uses the same surrounding target size and matching context, but its AMM JIT branch constructs an `AmmJitQuoter` directly from the match context using the full taker target and the maker's unfilled amount. It then includes the AMM JIT quoter alongside the DLOB maker and calls `match_take` with the original `target_size`, so this branch does not apply the standalone branch's `calculate_amm_available_liquidity` cap before exposing AMM JIT capacity.

Inside `AmmJitQuoter::from_match_context`, the JIT capacity is derived from the minimum of taker and maker unfilled sizes and `calculate_amm_jit_liquidity`. The quoter's capacity and fill logic

then clamp `max_jit_base` against `max_fillable`, which means the JIT path is limited by the hard AMM reserve bounds but not by the operator-configured per-fill reserve fraction.

The `max_fillable` enforces only hard reserve bounds, not the per-fill fraction. Thus, the JIT quoter's reserve-side limit is only the distance from the current base reserve to the AMM's minimum or maximum base reserve. Because this differs from the standalone AMM path's smaller per-fill cap, a permissionless match can cause the AMM JIT slice to move reserves and `base_asset_amount_with_amm` more in one fill than a standalone AMM fill would allow. This can affect subsequent AMM pricing, inventory, and AMM fee or PnL accounting, especially when the JIT execution price differs from the AMM curve price.

programs/velocity/src/vlp/amm/quoter.rs

```
897 fn max_fillable(&self, side: PositionDirection) -> VelocityResult<u64> {
898     let base = self.amm.base_asset_reserve;
899     let raw_u128 = match side {
900         // Taker Long → AMM sells base → base falls toward min → max = current_base
        - min_base.
901         PositionDirection::Long =>
        base.saturating_sub(self.amm.min_base_asset_reserve),
902         // Taker Short → AMM buys base → base rises toward max → max = max_base -
        current_base.
903         PositionDirection::Short =>
        self.amm.max_base_asset_reserve.saturating_sub(base),
904     };
905     Ok(raw_u128.min(u64::MAX as u128) as u64)
906 }
```

The JIT quoter's fillable amount is bounded by hard min/max reserves.

Remediation

Apply `calculate_amm_available_liquidity` or an equivalent shared per-fill reserve-fraction cap to the AMM JIT capacity before `AmmJitQuoter` exposes liquidity to `match_take`. The cap should be computed for the same taker direction and order step size used by the standalone AMM branch, then applied to `max_jit_base` in or before `AmmJitQuoter::from_match_context`.

Patch

Resolved in PR#269.

Bid/Ask TWAP Can Advance Funding In Settlement Markets

ID	Severity	Status
OS-VEP-ADV-86	Medium	Resolved

Description

The public `handle_update_perp_bid_ask_twap` keeper instruction was reachable for perp markets that passed the generic `perp_market_valid` constraint. That constraint only rejects `Delisted` markets, so it does not exclude markets in `Settlement`. After refreshing oracle-derived market state and computing the bid/ask TWAP inputs, `handle_update_perp_bid_ask_twap` calls the shared funding writer, as shown in the TWAP crank handler.

programs/velocity/src/instructions/keeper.rs

```rust
pub fn handle_update_perp_bid_ask_twap<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    let funding_paused =
        state.funding_paused()? ||
        perp_market.is_operation_paused(PerpOperation::UpdateFunding);
    controller::funding::update_funding_rate(
        perp_market.market_index,
        perp_market,
        &mut oracle_map,
        now,
        slot,
        &state.oracle_guard_rails,
        funding_paused,
        None,
    )?;
    [...]
}
```

Bid/ask TWAP crank also invokes the funding writer

The direct `handle_update_funding_rate` keeper instruction applies an explicit lifecycle check before invoking the same shared funding writer. That handler only permits funding updates when the market status is `Active` or `ReduceOnly`, which shows that settlement markets are intentionally outside the direct funding update path.

Because the TWAP crank skipped that lifecycle check but still invoked `controller::funding::update_funding_rate`, a caller whose `keeper_stats` account satisfied the bid/ask TWAP permission and quote insurance-fund stake checks could advance cumulative funding after a market had entered settlement. The settlement path calls `settle_funding_payment` before final expired-position settlement, so users settling expired positions could consume a post-expiry funding delta before their positions are closed against the expiry price, shifting settlement value between long and short positions during the settlement-only lifecycle.

Remediation

Decouple funding updates from `handle_update_perp_bid_ask_twap` so the bid/ask TWAP crank only updates TWAP-related market state. Keep the direct funding handler's `Active | ReduceOnly` lifecycle gate, and preferably enforce the same market-status requirement inside the shared `controller::funding::update_funding_rate` function so no future caller can advance cumulative funding for settlement markets by bypassing the instruction-level check.

Patch

Resolved in PR#213.

Perp Funding Settlement Paths Bypass Funding Pause

ID	Severity	Status
OS-VEP-ADV-87	Medium	Resolved

Description

The funding settlement controller applies cumulative funding deltas directly to a user's perp position and the corresponding market accounting, but `settle_funding_payment` only receives the user, user key, market, and timestamp. The helper updates cumulative user funding, the position's last cumulative funding rate, quote and break-even amounts, and `net_unsettled_funding_pnl` without checking the exchange funding pause state.

The direct funding-settlement instruction is protected by `funding_not_paused`, which returns an exchange pause error when the state reports funding as paused. This gate is present around `handle_settle_funding_payment`, but the shared helper itself has no equivalent pause input, so any other public instruction that reaches the helper can still apply funding while the direct funding entrypoint is disabled.

`fill_perp_order` reaches funding settlement before the later route fulfillment result determines whether anything was filled. In `fulfill_perp_order`, an empty set of fulfillment methods logs that no route was found and returns success with zero filled amounts, so the earlier taker funding settlement can be committed even when the order ultimately has no executable route.

programs/velocity/src/controller/orders.rs

```
fn fulfill_perp_order([...]) -> VelocityResult<(u64, u64)> {
    [...]
    if fulfillment_methods.is_empty() {
        msg!("no fulfillment methods found");
        return Ok((0, 0));
    }
    [...]
}
```

A no-route perp fulfillment returns success with zero filled amounts.

The same missing funding-specific gate appears in the related settlement paths described by the finding. The repository callsites show `transfer_perp_position`, special position transfer, PnL settlement, and liquidation routines invoking `settle_funding_payment` while the public handlers are gated by operation-specific pause checks such as fill, settle-PnL, or liquidation pause rather than `funding_not_paused`. During an emergency funding pause, this means user funding state may still change through those paths, affecting quote collateral, per-position cumulative funding, and market unsettled funding accounting.

Remediation

Require the funding pause gate on every public instruction that can call `settle_funding_payment` or `settle_funding_payments`, including fill, same-authority position transfer, PnL settlement, ex-

pired-position settlement, and liquidation paths. Alternatively, pass the relevant pause state into the shared funding controller and reject or skip funding settlement when `FundingPaused` is active.

Patch

Resolved in PR#404.

Market Funding Pause Does Not Stop Bid/Ask TWAP Mutation

ID	Severity	Status
OS-VEP-ADV-88	Medium	Resolved

Description

`handle_update_perp_bid_ask_twap` is an alternate keeper path for refreshing a perp market's bid/ask TWAP inputs. Its access control verifies that the perp market and oracle are valid and that exchange-wide funding is not paused, but it does not check whether that specific market has paused `PerpOperation::UpdateFunding`.

programs/velocity/src/instructions/keeper.rs

```
#[access_control(
    perp_market_valid(&ctx.accounts.perp_market)
    funding_not_paused(&ctx.accounts.state)
    valid_oracle_for_perp_market(&ctx.accounts.oracle, &ctx.accounts.perp_market)
)]
pub fn handle_update_perp_bid_ask_twap<'c: 'info, 'info>(
    ctx: Context<'info, UpdatePerpBidAskTwap<'info>>,
) -> Result<()> {[...]}
```

Bid/ask TWAP crank access control omits the market-scoped funding pause.

After passing those checks, the handler calls `update_mark_twap_crank`. That crank selects bid and ask prices from the AMM and optional DLOB inputs and then calls `update_mark_twap`, so the mark, bid, and ask TWAP state can still advance through this path while the market-level funding operation is paused.

The direct `handle_update_funding_rate` path computes a `funding_paused` flag from both the exchange-wide funding pause and the market's `PerpOperation::UpdateFunding` pause bit before invoking the funding controller. This means the final funding-rate update path observes the market-scoped pause, while the bid/ask TWAP crank that updates funding inputs does not.

As a result, a market-scoped `UpdateFunding` pause freezes the direct cumulative funding write but does not freeze the TWAP inputs that funding later consumes. Callers able to satisfy the bid/ask TWAP crank's existing keeper-stat checks may continue advancing or influencing that state during the pause, so when funding is resumed it may use TWAP values that changed while the market-specific funding pause was intended to be in effect.

Remediation

Add the same market-scoped `PerpOperation::UpdateFunding` pause enforcement to `handle_update_perp_bid_ask_twap` before any oracle-stat refresh, AMM quote-state refresh, or `update_mark_twap_crank` call that can mutate funding-relevant TWAP inputs.

Patch

Resolved in PR#404.

Funding Updates Accept Stale Or Insufficient Oracles

ID	Severity	Status
OS-VEP-ADV-89	Medium	Ack'd

Description

Velocity uses action-specific oracle validity policies, and the funding policy is more permissive than the policy used for margin calculations. In `is_oracle_valid_for_action`, the `VelocityAction::UpdateFunding` branch accepts oracle states that include stale-for-margin and insufficient-data classifications, so those classifications are treated as acceptable for the funding gate shown in this snippet.

programs/velocity/src/math/oracle.rs

```
pub fn is_oracle_valid_for_action([...]) -> VelocityResult<bool> {
    let is_ok = match action {
        Some(action) => match action {
            [...]
            // relax oracle staleness, later checks for sufficiently recent amm slot update for funding update
            VelocityAction::UpdateFunding => {
                matches!(
                    oracle_validity,
                    OracleValidity::Valid
                        | OracleValidity::StaleForAMM { .. }
                        | OracleValidity::InsufficientDataPoints
                        | OracleValidity::StaleForMargin
                )
            }
            VelocityAction::OracleOrderPrice => {[...]}
            [...]
        },
        [...]
    };
    [...]
}
```

The same action-dispatch function applies a stricter rule for `VelocityAction::MarginCalc`. That branch rejects `StaleForMargin`, which means an oracle value can be too stale for margin-sensitive calculations while still being considered usable by the funding update path, as shown by the margin policy in this snippet.

programs/velocity/src/math/oracle.rs

```
pub fn is_oracle_valid_for_action([...]) -> VelocityResult<bool> {
    let is_ok = match action {
```

```
        Some(action) => match action {
            [...]
            }
            VelocityAction::MarginCalc => !matches!(
                oracle_validity,
                OracleValidity::NonPositive
                    | OracleValidity::TooVolatile
                    | OracleValidity::TooUncertain
                    | OracleValidity::StaleForMargin
            ),
          [...]
        },
        [...]
    };
    [...]
}
```

The funding gate in `block_operation` obtains the oracle status and then checks it using `VelocityAction::UpdateFunding`. Because this call uses the permissive funding policy, `block_operation` does not block solely because the oracle is `StaleForMargin` or `InsufficientDataPoints`. It only blocks when the selected funding validity check fails, the mark/oracle spread is too divergent, the AMM update is too old, or funding is paused.

programs/velocity/src/math/oracle.rs

```
pub fn block_operation(
    market: &PerpMarket,
    oracle_price_data: &OraclePriceData,
    guard_rails: &OracleGuardRails,
    reserve_price: u64,
    slot: u64,
) -> VelocityResult<bool> {
    let OracleStatus {
        oracle_validity,
        mark_too_divergent: is_oracle_mark_too_divergent,
        oracle_reserve_price_spread_pct: _,
        ..
    } = get_oracle_status(market, oracle_price_data, guard_rails, reserve_price)?;
    let is_oracle_valid =
        is_oracle_valid_for_action(oracle_validity,
        Some(VelocityAction::UpdateFunding))?;
    [...]
}
```

The public `update_funding_rate` instruction has no signer in its account context, reads the market oracle, refreshes market oracle-derived stats, and calls the controller funding update. In the

controller, `block_operation` is evaluated before the function computes funding, updates oracle and mark TWAPs, mutates `cumulative_funding_rate_long` and `cumulative_funding_rate_short`, updates funding metadata, and emits the funding record. As a result, any caller who reaches a funding boundary may advance cumulative funding using an oracle classification that the protocol's own margin-calculation policy would reject, causing later funding settlement to use cumulative rates derived from that weaker oracle condition.

Remediation

Make the funding gate reject `OracleValidity::StaleForMargin` and `OracleValidity::InsufficientDataPoints` before any cumulative funding, funding TWAP, or funding metadata mutation occurs. This can be done by tightening the `VelocityAction::UpdateFunding` policy or by adding an explicit funding-specific freshness and data-sufficiency check in the controller before `update_funding_rate` mutates market state.

Patch

This issue was acknowledged by Velocity Exchange.

Equity Breaker Does Not Block Liquidator Actions

ID	Severity	Status
OS-VEP-ADV-90	Medium	Resolved

Description

The equity-floor breaker is intended to freeze risk-increasing activity across every subaccount for an authority after a keeper proves that one subaccount is below its equity floor. The keeper handler for tripping the breaker sets `UserStats.equity_breaker_tripped`, and the flag is authority-wide. The direct spot liquidation account context, however, authenticates only the liquidator `User` against the signing authority and does not load the liquidator's `UserStats`, so that route has no account from which it can enforce the authority-wide breaker before changing the liquidator's risk state.

programs/velocity/src/instructions/keeper.rs

```
3357 pub struct LiquidateSpot<'info> {
3358     pub state: AccountLoader<'info, State>,
3359     pub authority: Signer<'info>,
3360     #[account(
3361         mut,
3362         constraint = can_sign_for_user(&liquidator, &authority)?
3363     )]
3364     pub liquidator: AccountLoader<'info, User>,
3365     #[account(mut)]
3366     pub user: AccountLoader<'info, User>,
3367 }
```

Direct spot liquidation omits the liquidator UserStats account.

In `liquidate_spot`, the controller transfers spot balances between the liquidatable user and liquidator, then checks whether the liquidator meets the ordinary initial margin requirement. As shown in the spot liquidation gate, the validation is based on the liquidator's margin calculation and does not consult an equity-breaker flag. A same-authority subaccount that is still healthy enough to satisfy initial margin can therefore continue receiving discounted spot exposure while the authority's breaker is tripped.

programs/velocity/src/controller/liquidation.rs

```
1 pub fn liquidate_spot([...]) -> VelocityResult {
2     [...]
3     let liquidator_meets_initial_margin_requirement =
4         calculate_margin_requirement_and_total_collateral_and_liability_info(
5             liquidator,
6             perp_market_map,
7             spot_market_map,
8             oracle_map,
```

```
            liq_margin_context,
        )
        .map(|calc| calc.meets_margin_requirement())?;
    validate!(
        liquidator_meets_initial_margin_requirement,
        ErrorCode::InsufficientCollateral,
        "Liquidator doesnt have enough collateral to take over borrow"
    )?;
    [...]
}
```

Spot liquidation validates only the liquidator's initial margin.

The direct perp liquidation route does receive both the user and liquidator `UserStats` accounts in its instruction context, but the controller gate shown in `liquidate_perp` likewise validates only the liquidator's initial margin status. Because the available check does not reject `liquidator_stats.is_equity_breaker_tripped()`, a frozen authority may still use a healthy sibling subaccount as liquidator for perp liquidations. This bypass is limited by the remaining initial-margin checks.

programs/velocity/src/controller/liquidation.rs

```
pub fn liquidate_perp([...]) -> VelocityResult {
    [...]
    let liquidator_meets_initial_margin_requirement =
        meets_initial_margin_requirement(liquidator, perp_market_map, spot_market_map,
        oracle_map)?;

    validate!(
        liquidator_meets_initial_margin_requirement,
        ErrorCode::InsufficientCollateral,
        "Liquidator doesnt have enough collateral to take over perp position"
    )?;
    [...]
}
```

Perp liquidation uses the same initial-margin-only liquidator gate.

Remediation

Load and authenticate the liquidator `UserStats` account in every liquidation route that can mutate liquidator risk or value state, including direct spot liquidation where it is currently absent. Before performing liquidator balance, position, PnL, or fee mutations, reject the action when the authenticated liquidator stats have `equity_breaker_tripped` set.

Patch

Resolved in PR#393.

Oracle Validity Cache Ignores Market Context

ID	Severity	Status
OS-VEP-ADV-91	Medium	Resolved

Description

`OracleMap` identifies an oracle by the oracle account and source, as shown by the `OracleIdentifier` alias. The `OracleMap` state also keeps a `validity` map keyed by this identifier, so entries for markets that share the same oracle account and source are stored under the same cache key.

programs/velocity/src/state/oracle_map.rs

```
1  pub(crate) type OracleIdentifier = (Pubkey, OracleSource);
```

In `get_price_data_and_validity`, the cached-price branch first retrieves the oracle price data and then checks `self.validity` with only `oracle_id`. When a cached validity entry exists, the function returns that stored `OracleValidity` instead of calling `oracle_validity` again with the current `market_type`, `market_index`, TWAP, confidence multiplier, and staleness overrides.

The `oracle_validity` function accepts market-specific and per-call inputs, including market type, market index, last oracle TWAP, the confidence interval multiplier, oracle source, and staleness override values. Its body uses these values to classify the oracle as valid, too volatile, too uncertain, stale, or otherwise invalid, so the result is not solely a property of the oracle account and source.

This can affect margin calculations because `calculate_margin_requirement_and_total_collateral_and_liability_info` iterates over spot and perpetual positions and calls `get_price_data_and_validity` for each market using that market's TWAP, confidence multiplier, and override settings. If an earlier market caches a shared oracle as `Valid`, a later market using the same oracle identifier may reuse that result even though its own parameters would have produced a stricter invalid classification. In a value-release path such as `withdraw`, this may allow the margin check to proceed with collateral leaving the vault when the liability market's oracle validity gate should have blocked the action.

Remediation

Do not cache `OracleValidity` solely by `OracleIdentifier`. Either recompute validity on every `get_price_data_and_validity` call after retrieving cached price data, or include every validity input in the cache key, including market type, market index, last TWAP, confidence multiplier, oracle source, and staleness override values.

Patch

Resolved in PR#391.

PnL Settlement Accepts Stale Or Insufficient Oracles

ID	Severity	Status
OS-VEP-ADV-92	Medium	Resolved

Description

PnL settlement uses a settlement-specific oracle policy that treats several degraded oracle states as acceptable. In `is_oracle_valid_for_action`, the `VelocityAction::SettlePnl` branch accepts not only `Valid`, but also `StaleForAMM`, `InsufficientDataPoints`, and `StaleForMargin`, as shown in the settlement oracle-policy branch.

programs/velocity/src/math/oracle.rs

```
pub fn is_oracle_valid_for_action([...]) -> VelocityResult<bool> {
    let is_ok = match action {
        Some(action) => match action {
            [...]
            ),
            VelocityAction::SettlePnl => matches!(
                oracle_validity,
                OracleValidity::Valid
                    | OracleValidity::StaleForAMM { .. }
                    | OracleValidity::InsufficientDataPoints
                    | OracleValidity::StaleForMargin
            ),
            [...]
        },
        [...]
    };
    [...]
}
```

SettlePnl accepts stale and insufficient oracle classes

The public `settle_pnl` controller relies on that permissive policy when the market's recent oracle check is not healthy. After this gate, the same controller computes claimable PnL from the oracle price, updates PnL-pool and quote spot balances, updates the perp position quote asset amount, records settled PnL, and emits a settlement record, so a degraded oracle class accepted at this point can still drive accounting mutation.

The controller also restricts third-party settlement only when the settlement amount is non-negative, the PnL pool is not in excess, the user is not covered by the liquidation exception, and the signer is neither the user's authority nor delegate. Because the condition is tied to `pnl_to_settle_with_user >= 0`, the guard does not block a third-party signer from settling a negative-PnL debit for a user who otherwise passes the negative-PnL margin requirement.

programs/velocity/src/controller/pnl.rs

```rust
pub fn settle_pnl([...]) -> VelocityResult {
    [...]
    let user_must_settle_themself = pnl_to_settle_with_user >= 0
        && max_pnl_pool_excess <= 0
        && !(pnl_to_settle_with_user > 0 && base_asset_amount == 0 &&
        user.is_being_liquidated())
        && !(user.authority.eq(authority) || user.delegate.eq(authority));

    if user_must_settle_themself {
        let msg = format!(
            "Market = {} user must settle their own unsettled pnl when its positive and
            pnl pool not in excess",
            market_index
        );
        return mode.result(
            ErrorCode::UserMustSettleTheirOwnPositiveUnsettledPNL,
            market_index,
            &msg,
        );
    }
    [...]
}
```

Third-party self-settlement restriction is limited to non-negative PnL

The result is that stale-for-margin or insufficient oracle data can be used to crystallize a user's negative PnL through the normal settlement path. This may misassign quote value between the user, the quote spot market, and the perp PnL pool at a price that the protocol has already classified as unsuitable for stricter oracle-sensitive actions such as margin freshness.

Remediation

Tighten the settlement oracle gate so `VelocityAction::SettlePnl` rejects `OracleValidity::StaleForMargin` and `OracleValidity::InsufficientDataPoints`, or add an explicit settlement-only freshness and data-sufficiency check before any PnL, quote spot balance, or PnL-pool accounting mutation.

Patch

Resolved in PR#391.

Prelaunch Oracle Stale Price Reports Zero Delay

ID	Severity	Status
OS-VEP-ADV-93	Medium	Resolved

Description

For markets using `OracleSource::Prelaunch`, `get_oracle_price` dispatches to `get_prelaunch_price`, which loads the `PrelaunchOracle` account and returns an `OraclePriceData` record. In that prelaunch path, the returned `delay` is computed from `amm_last_update_slot` using `saturating_sub(slot)`, so once the current slot is greater than the stored AMM update slot, the elapsed delay is reported as `0` instead of the number of slots since the source was last refreshed.

programs/velocity/src/state/oracle.rs

```
pub fn get_prelaunch_price(
    price_oracle: &AccountInfo,
    slot: u64,
) -> VelocityResult<OraclePriceData> {
    let oracle: Ref<PrelaunchOracle> =
    load_ref(price_oracle).or(Err(UnableToLoadOracle))?;

    Ok(OraclePriceData {
        price: oracle.price,
        confidence: oracle.confidence,
        delay: oracle.amm_last_update_slot.saturating_sub(slot).cast()?,
        has_sufficient_number_of_data_points: true,
        sequence_id: None,
    })
}
```

Prelaunch price reports delay with reversed slot subtraction

`PrelaunchOracle::update` derives the synthetic price and confidence from the perp market state, stores `perp_market.amm.last_update_slot()` in `amm_last_update_slot`, and separately records the current update slot in `last_update_slot`. Therefore, the value later used by `get_prelaunch_price` is the AMM source slot rather than necessarily the slot at which the prelaunch oracle account was updated.

programs/velocity/src/state/oracle.rs

```
pub fn update(&mut self, perp_market: &PerpMarket, slot: u64) -> VelocityResult {
    [...]
    self.amm_last_update_slot = perp_market.amm.last_update_slot();
    self.last_update_slot = slot;
    [...]
}
```

Prelaunch update stores the AMM source slot separately

The stale-delay value returned by the prelaunch adapter is then part of the normal oracle flow. `OracleMap::get_price_data_and_validity` obtains price data through `get_oracle_price` and passes the resulting `OraclePriceData` into `oracle_validity`, so the prelaunch delay is consumed by the same validity logic used for other market oracle checks.

In `oracle_validity`, `oracle_delay` is compared against AMM immediate, AMM low-risk, and margin staleness thresholds before the function classifies the oracle as stale or valid. If a stale prelaunch source reports delay `0`, these elapsed-slot comparisons do not trip solely because time has advanced, so the prelaunch oracle can appear fresh to the staleness gates despite the AMM-derived source slot being old.

For any perp market configured with `OracleSource::Prelaunch`, this can allow public fills, margin checks, and liquidations that depend on oracle validity to continue using the obsolete synthetic price instead of rejecting it as stale. The resulting decisions may be based on an old prelaunch price rather than on a currently refreshed source.

Remediation

Compute the prelaunch oracle delay in elapsed-slot order, using the current slot minus the stored AMM source slot, such as `slot.saturating_sub(oracle.amm_last_update_slot)`, so a later current slot produces a positive delay.

Patch

Resolved in PR#391.

Pyth Lazer Confidence Can Be Understated By Caller-Selected Properties

ID	Severity	Status
OS-VEP-ADV-94	Medium	Resolved

Description

The Pyth Lazer update handler verifies the signed message, deserializes the payload, and then iterates over each feed's `properties` before updating the corresponding `PythLazerOracle` account. In that property loop, `handle_update_pyth_lazer_oracle` only handles the best bid, best ask, exponent, and feed update timestamp fields; all other property variants are ignored. As shown in the property matching logic, this means a signed `Confidence` property in the Lazer payload is not consumed when computing the cached oracle confidence.

programs/velocity/src/instructions/pyth_lazer_oracle.rs

```
pub fn handle_update_pyth_lazer_oracle<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    for (account, payload_data) in remaining_accounts.iter().zip(data.feeds.iter()) {
        [...]
        for property in &payload_data.properties {
            match property {
                PayloadPropertyValue::BestBidPrice(price) => best_bid_price = *price,
                PayloadPropertyValue::BestAskPrice(price) => best_ask_price = *price,
                PayloadPropertyValue::Exponent(exp) => exponent = Some(*exp),
                PayloadPropertyValue::FeedUpdateTimestamp(timestamp) => match timestamp
                {
                    Some(timestamp) => next_timestamp = Some(timestamp.as_micros()),
                    None => continue,
                },
                _ => {}
            }
        }
        [...]
    }
    [...]
}
```

Pyth Lazer property parsing ignores unmatched properties, including confidence.

The handler then derives `conf` from bid and ask prices rather than from the signed confidence value. As shown in the cache update path, it initializes confidence to `price / 500`, and only replaces that value when both bid and ask are present and ordered with bid below ask. If the submitted property set has missing, one-sided, or inverted bid and ask data, the oracle account is updated with the fixed 20 bps fallback confidence.

programs/velocity/src/instructions/pyth_lazer_oracle.rs

```rust
pub fn handle_update_pyth_lazer_oracle<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    for (account, payload_data) in remaining_accounts.iter().zip(data.feeds.iter()) {
        [...]
        // Default to 20bps of the price for conf if bid > ask or one-sided market
        let mut conf: i64 = price.safe_div(500)?;
        if let (Some(bid), Some(ask)) = (best_bid_price, best_ask_price) {
            if bid.mantissa_i64() < ask.mantissa_i64() {
                conf = ask.mantissa_i64() - bid.mantissa_i64();
            }
        }
        pyth_lazer_oracle.price = price;
        pyth_lazer_oracle.posted_slot = Clock::get()?.slot;
        pyth_lazer_oracle.publish_time = next_timestamp.unwrap();
        pyth_lazer_oracle.exponent = exponent.cast::<i32>()?;
        pyth_lazer_oracle.conf = conf.cast::<u64>()?;
        msg!("Price updated to {}", price);
        [...]
    }
    [...]
}
```

The cached confidence falls back to 20 bps unless usable bid and ask prices are present.

Downstream oracle loading reads `PythLazerOracle.conf` into `OraclePriceData.confidence`, and `oracle_validity` compares that confidence against the configured confidence interval limit to determine `TooUncertain`. An understated cached confidence can therefore make a Lazer price appear less uncertain than the signed feed data indicates, allowing an otherwise too-uncertain price to pass the uncertainty gate and be used by protocol flows that rely on oracle validity.

Remediation

Require the Lazer payload used for protocol oracle updates to include the signed `Confidence` property, parse that property explicitly in `handle_update_pyth_lazer_oracle`, and persist it to `PythLazerOracle.conf`. If bid/ask spread is retained as an additional uncertainty signal, store the maximum of the signed confidence and the bid/ask-derived spread, and reject updates whose property set cannot provide the confidence data required by the oracle validity policy.

Patch

Resolved in PR#391.

Fee Sweep Can Starve Pending Revenue-Share Claims

ID	Severity	Status
OS-VEP-ADV-95	Medium	Resolved

Description

`sweep_market_fees` computes the PnL-pool amount available for fee movement by reserving only positive net user PnL before sweeping accrued fees out of the market PnL pool. The protocol-fee drain is applied first, uses this unbuffered availability, and is not reduced by the market fee-pool buffer that later applies to the insurance-fund and AMM-provision drains. As shown in `sweep_market_fees`, this reservation does not include already-accrued builder or referrer revenue-share amounts that are also paid from the PnL pool.

programs/velocity/src/controller/perp_pools.rs

```
pub fn sweep_market_fees([...]) -> VelocityResult<(u128, u128, u128)> {
    [...]
    // not recoverable later anyway)
    let reserved_claims: u128 = net_user_pnl.max(0).cast::<u128>()?;
    let mut available_unbuffered: u128 =
    pnl_pool_tokens.saturating_sub(reserved_claims);
    // 1. protocol's withdrawable cut (buffer-exempt: only user claims reserved)
    let protocol_drain = market
        .fee_ledger
        .pending_protocol_fee
        .min(available_unbuffered);
    if protocol_drain > 0 {
        transfer_spot_balances(
            protocol_drain.cast()?,
            spot_market,
            &mut market.pnl_pool,
            &mut market.protocol_fee_pool,
        )?;
        market.fee_ledger.consume_pending_protocol(protocol_drain)?;
        available_unbuffered = available_unbuffered.safe_sub(protocol_drain)?;
    }
    [...]
}
```

`sweep_market_fees` reserves user PnL before the buffer-exempt protocol drain.

`sweep_completed_revenue_share_for_market` pays completed builder or referral revenue-share orders from the same market PnL pool. Before each payment, it checks the current PnL-pool token amount against the order's accrued fees; when the pool is smaller than the accrued fee amount, the function logs the condition and breaks out of the loop rather than reverting the earlier fee movement or preserving the claim's priority.

programs/velocity/src/controller/revenue_share.rs

```rust
pub fn sweep_completed_revenue_share_for_market<'a>([...]) ->
crate::error::VelocityResult<()> {
    [...]
    for i in 0..orders_len {
        [...]
        if pnl_pool_token_amount < fees_accrued as u128 {
            msg!(
                "market {} PNL pool has insufficient balance to sweep fees for builder.
                pnl_pool_token_amount: {}, fees_accrued: {}",
                market_index,
                pnl_pool_token_amount,
                fees_accrued
            );
            break;
        }
        [...]
    }
    [...]
}
```

`sweep_completed_revenue_share_for_market` exits when the PnL pool cannot cover accrued fees.

`settle_pnl` reaches the same fee-sweep path through `update_pool_balances`, which settles the user's PnL against the pool and then calls `sweep_market_fees`.

programs/velocity/src/controller/pnl.rs

```rust
pub fn settle_pnl([...]) -> VelocityResult {
    [...]
    let pnl_to_settle_with_user = update_pool_balances(
        &mut perp_market,
        &mut spot_market,
        user_quote_token_amount,
        user_unsettled_pnl,
        net_user_pnl,
        now,
    )?;

    [...]
}
```

`settle_pnl` reaches the fee sweep through `update_pool_balances`.

Because the protocol-fee sweep can materialize pending protocol fees before completed revenue-share claims are reserved or paid, a market with tight PnL-pool headroom can have a previously payable builder or referrer claim become temporarily unpayable. This results in delayed

revenue-share payout until the PnL pool is replenished, with protocol-fee materialization taking priority over already-accrued third-party revenue-share claims.

Remediation

When computing the PnL-pool reservation in `sweep_market_fees`, include completed builder and referrer revenue-share fees that are payable from the market's PnL pool, or settle/reserve those completed revenue-share claims before any buffer-exempt protocol-fee drain is applied.

Patch

Resolved in PR#392.

Fee Sweep Accepts Stale Margin Insufficient Oracles

ID	Severity	Status
OS-VEP-ADV-96	Medium	Ack'd

Description

The oracle policy used for `SettlePnl` accepts oracle states that are weaker than the quality normally required for margin-sensitive aggregate accounting. In `is_oracle_valid_for_action`, the `SettlePnl` branch treats `StaleForMargin` and `InsufficientDataPoints` as acceptable, so callers using that action may continue with a stale-for-margin or low-data oracle rather than rejecting it outright.

programs/velocity/src/math/oracle.rs

```
pub fn is_oracle_valid_for_action([...]) -> VelocityResult<bool> {
    let is_ok = match action {
        Some(action) => match action {
            [...]
            ),
            VelocityAction::SettlePnl => matches!(
                oracle_validity,
                OracleValidity::Valid
                    | OracleValidity::StaleForAMM { .. }
                    | OracleValidity::InsufficientDataPoints
                    | OracleValidity::StaleForMargin
            ),
            [...]
        },
        [...]
    };
    [...]
}
```

SettlePnl accepts stale-for-margin and insufficient-data oracle states.

The permissionless `sweep_perp_market_fees` entry point reaches `handle_sweep_perp_market_fees`, where the keeper path validates a non-healthy oracle using the same `VelocityAction::SettlePnl` policy before proceeding. That ties the sweep's oracle acceptance to a policy designed for PnL settlement behavior, even though this instruction can move accrued protocol, insurance-fund, and AMM fee carveouts out of the market's PnL pool.

After accepting the oracle price, `handle_sweep_perp_market_fees` computes aggregate `net_user_pnl` at that price and passes it into `sweep_market_fees`. The pool controller reserves only positive user PnL claims, using `max(net_user_pnl, 0)`, before transferring pending fees from the PnL pool to the protocol fee pool, revenue pool, and AMM fee pool. If the accepted stale-for-margin or insufficient-data oracle understates aggregate user PnL, the sweep can reserve too little for live user claims before those fee balances leave the PnL pool.

programs/velocity/src/instructions/keeper.rs

```rust
pub fn handle_sweep_perp_market_fees([...]) -> Result<()> {
    [...]
    let net_user_pnl = calculate_net_user_pnl(
        &perp_market.amm,
        oracle_price,
        perp_market.quote_asset_amount,
        perp_market.net_unsettled_funding_pnl,
    )?;
    let (if_swept, protocol_swept, amm_provision_tokenized) =
        controller::perp_pools::sweep_market_fees(
            perp_market,
            spot_market,
            net_user_pnl,
            now,
            false,
        )?;
    [...]
}
```

The sweep sizes user PnL reserves from the accepted oracle price.

Because the sweep is exposed as a public keeper instruction and the controller documentation describes it as an on-demand materialization of pending fee carveouts, this can leave fewer tokens backing later user PnL settlement when a permissive oracle state is accepted for the aggregate reserve calculation. While the impact is limited to the fee amounts, the sweep is able to move under the controller's existing accounting and buffer rules, but those outflows can still reduce the PnL pool backing available to unsettled users.

Remediation

Use a stricter oracle action or a dedicated fee-sweep oracle gate for `handle_sweep_perp_market_fees` before computing `net_user_pnl`. That gate should reject `StaleForMargin` and `InsufficientDataPoints` states for the aggregate PnL-pool reserve calculation, while preserving any separate policy needed for ordinary single-user PnL settlement.

Patch

This issue was acknowledged by Velocity Exchange.

SettleRevPool Pause Does Not Stop Revenue-Share Sweep

ID	Severity	Status
OS-VEP-ADV-97	Medium	Resolved

Description

The market-scoped `PerpOperation::SettleRevPool` pause is enforced on the direct market-fee sweep path. In `sweep_market_fees`, the non-forced path returns without moving funds when the market has `SettleRevPool` paused, so ordinary keeper or streaming fee sweeps respect the market-level pause.

programs/velocity/src/controller/perp_pools.rs

```
pub fn sweep_market_fees([...]) -> VelocityResult<(u128, u128, u128)> {
    [...]
    if (!force && market.is_operation_paused(PerpOperation::SettleRevPool))
        || (market.fee_ledger.pending_protocol_fee == 0
            && market.fee_ledger.pending_if_fee == 0
            && market.fee_ledger.pending_amm_provision == 0)
    {[...]}
    [...]
}
```

Direct fee sweep respects `SettleRevPool` unless forced.

The revenue-share cleanup path does not apply the same market pause policy. In `sweep_completed_revenue_share_for_market`, the function loads the perp market and quote spot market, filters completed builder or referrer orders for the requested perp market, checks only whether the raw PnL-pool token amount covers the accrued fees, and then later transfers those fees from the market PnL pool to the builder or referrer quote position.

`handle_settle_multiple_pnls` invokes the revenue-share cleanup after each requested market settlement path whenever builder codes are enabled and the relevant accounts are supplied. Because this call is outside the actual `settle_pnl` result handling and follows the settlement attempt for each market, the cleanup can still run in cases where `TrySettle` returns without completing a user PnL settlement.

This creates an emergency-control bypass for paused markets, enabling a public settlement caller to trigger builder or referrer revenue-share cleanup that debits quote from a market PnL pool while the dedicated market revenue-pool sweep is paused. The impact is limited to accrued revenue-share amounts that pass the function's existing order and balance checks, but those debits come from the same PnL pool used in user-PnL accounting.

Remediation

Apply the same `PerpOperation::SettleRevPool` policy to `sweep_completed_revenue_share_for_market` before it can transfer from `perp_market.pnl_pool`.

Also make the settlement handlers skip revenue-share cleanup when `TrySettle` soft-skips the underlying `settle_pnl` operation, so cleanup only follows an actual settlement path.

Patch

Resolved in PR#272.

Signed-Message Taker Orders Bypass The Global Exchange Pause

ID	Severity	Status
OS-VEP-ADV-98	Medium	Resolved

Description

`handle_place_signed_msg_taker_order` is exposed through the program entrypoint for signed-message taker orders and loads the exchange state before building market and oracle maps, loading the taker accounts, optionally loading builder escrow, and calling `place_signed_msg_taker_order`. The handler has no adjacent `access_control` wrapper that calls `exchange_not_paused`, so a globally paused exchange does not block this entrypoint before it begins signed-message order processing.

The normal multi-order placement path is protected differently. `handle_place_orders` is declared with an `exchange_not_paused(&ctx.accounts.state)` access-control check, and the nearby scale-order placement handler follows the same pattern, so ordinary user order placement is rejected at the instruction boundary when the global exchange pause condition is active.

programs/velocity/src/instructions/user.rs

```
2653        exchange_not_paused(&ctx.accounts.state)
2654    )]
2655    pub fn handle_place_orders<'c: 'info, 'info>(
2656        ctx: Context<'info, PlaceOrder>,
2657        params: Vec<OrderParams>,
2658    ) -> Result<()> {
2659        place_orders(&ctx, PlaceOrdersInput::Orders(params))
2660    }
```

Normal order placement is wrapped with `exchange_not_paused`.

The missing guard matters because `exchange_not_paused` is the function that converts an all-bits `ExchangeStatus` pause state into `ErrorCode::ExchangePaused`. Without the same guard on the signed-message taker handler, the signed-message path can continue through Ed25519 verification, signed-message account duplicate checks, and the shared `controller::orders::place_perp_order` call. The controller performs order-level, user, and market checks, but it does not enforce the global all-bits exchange pause itself. As a result, signed-message taker orders may be accepted during a global pause and remain queued for execution once pause-related blocking is removed, undermining the intended emergency freeze for new risk.

programs/velocity/src/instructions/constraints.rs

```
159 pub fn exchange_not_paused(state: &AccountLoader<State>) -> anchor_lang::Result<()>
    {
160     let state = state.load()?;
161     if state.get_exchange_status()?.is_all() {
162         return Err(ErrorCode::ExchangePaused.into());
```

```
163         }
164         Ok(())
165     }
```

Global pause guard returns `ExchangePaused` when all exchange status bits are set.

Remediation

Add the same `exchange_not_paused(&ctx.accounts.state)` access-control guard to `handle_place_signed_msg_taker_order` that is used by normal order placement.

Patch

Resolved in PR#271.

Delegate Can Shrink Authority Signed-Message Replay PDA

ID	Severity	Status
OS-VEP-ADV-99	Medium	Resolved

Description

The signed-message replay account can be shrunk by a signer that is only the delegate of the supplied `user` account. In `handle_resize_signed_msg_user_orders`, the resize handler only blocks shrinking when the payer is neither the supplied authority nor the loaded user's delegate; when the payer is that delegate, the handler proceeds to resize `signed_msg_order_data`, so entries beyond the new length are removed.

programs/velocity/src/instructions/user.rs

```
pub fn handle_resize_signed_msg_user_orders<'c: 'info, 'info>(
    ctx: Context<'info, ResizeSignedMsgUserOrders<'info>>,
    num_orders: u16,
) -> Result<()> {
    let signed_msg_user_orders = &mut ctx.accounts.signed_msg_user_orders;
    let user = load!(ctx.accounts.user)?;
    if ctx.accounts.payer.key != ctx.accounts.authority.key
        && ctx.accounts.payer.key != &user.delegate.key()
    {
        validate!(
            num_orders as usize >= signed_msg_user_orders.signed_msg_order_data.len(),
            ErrorCode::InvalidSignedMsgUserOrdersResize,
            "Invalid shrinking resize for payer != user authority or delegate"
        )?;
    }
    [...]
}
```

The account context for `ResizeSignedMsgUserOrders` ties the replay PDA to the unchecked `authority` account and constrains the supplied `user` to have that same authority. Because `authority` is not required to sign, a delegate for one subaccount under that authority can satisfy the resize authorization through that `user` while the resized PDA remains scoped to the shared authority.

programs/velocity/src/instructions/user.rs

```
4470 pub struct ResizeSignedMsgUserOrders<'info> {
4471     #[account(
4472         mut,
4473         seeds = [SIGNED_MSG_PDA_SEED.as_bytes(), authority.key().as_ref()],
4474         bump,
4475         realloc = SignedMsgUserOrders::space(num_orders as usize),
```

```
4476            realloc::payer = payer,
4477            realloc::zero = false,
4478        )]
4479        pub signed_msg_user_orders: Box<Account<'info, SignedMsgUserOrders>>,
4480        /// CHECK: authority
4481        pub authority: UncheckedAccount<'info>,
4482        #[account(
4483            has_one = authority
4484        )]
4485        pub user: AccountLoader<'info, User>,
4486        #[account(mut)]
4487        pub payer: Signer<'info>,
4488    }
```

`PlaceSignedMsgTakerOrder` also derives the signed-message replay account from the taker user's authority rather than from the individual user account. This means orders for sibling subaccounts under the same authority consult the same replay storage, so resizing it through one subaccount affects the UUID rows used to protect the others.

programs/velocity/src/instructions/keeper.rs

```
1  pub struct PlaceSignedMsgTakerOrder<'info> {
2      [...]
3      #[account(
4          mut,
5          seeds = [SIGNED_MSG_PDA_SEED.as_bytes(), user.load()?.authority.as_ref()],
6          bump,
7      )]
8      [...]
```

The replay logic loads this account during signed-message order placement, scans the retained `SignedMsgOrderId` rows for a matching UUID that has not expired, and records the UUID only after accepting the order. If a delegate shrinks the shared account so that another subaccount's active UUID row is truncated, the duplicate scan no longer sees that UUID, and the same authority-signed payload may be accepted again while its normal placement checks and `max_slot` condition still pass.

Remediation

Require the authority signer for any resize that reduces the signed-message replay account length. Alternatively, scope the replay PDA by subaccount rather than only by authority. If delegate-initiated resizing remains supported, restrict delegates to growth-only resizes or to pruning entries that the program proves are expired, and add a regression test showing that a delegate for one subaccount cannot remove an unexpired UUID row protecting a sibling subaccount.

Patch

Resolved in PR#271.

Signed Messages Lack Deployment Domain Separation

ID	Severity	Status
OS-VEP-ADV-100	Medium	Ack'd

Description

Non-delegate signed-message orders are verified as signatures over the supplied message bytes, but the decoded order intent is not bound to the Velocity deployment. In `deserialize_into_verified_message`, the non-delegate branch accepts a payload with an eight-byte prefix, decodes the remaining bytes as `SignedMsgOrderParamsMessage`, and carries the decoded order parameters, subaccount id, slot, uuid, optional trigger and builder fields, and signature into `VerifiedMessage`. The surrounding verifier checks that the Ed25519 instruction signed the same payload bytes and that the signer matches the taker authority, but the fixed prefix is treated as a manual discriminator for decoding rather than as an on-chain-validated deployment domain.

The taker entrypoint then uses the decoded non-delegate message fields in the current program context. In `place_signed_msg_taker_order`, the non-delegate path derives the expected user PDA from the loaded taker authority and the decoded `sub_account_id` under the current Velocity program id, then only checks that this PDA matches the supplied taker account. This confirms the decoded subaccount belongs to the supplied authority in Velocity, but it does not prove the authority signed a Velocity-domain intent rather than another compatible message layout.

programs/velocity/src/instructions/keeper.rs

```
pub fn place_signed_msg_taker_order<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    if is_delegate_signer {
        [...]
    } else {
        // Verify taker passed to the ix matches pda derived from subaccount id + authority
        let taker_pda = Pubkey::find_program_address(
            &[
                "user".as_bytes(),
                &taker.authority.to_bytes(),
                &verified_message_and_signature
                    .sub_account_id
                    .unwrap()
                    .to_le_bytes(),
            ],
            &ID,
        );
        validate!(
            taker_pda.0 == taker_key,
            ErrorCode::SignedMsgUserContextUserMismatch,
```

```
            "Taker key does not match pda"
        )?;
    };
    [...]
}
```

Taker PDA is derived from decoded subaccount data

The freshness check in `place_signed_msg_taker_order` is also limited to rejecting decoded slots that are more than 500 slots behind the current slot. As shown in the slot validation, there is no corresponding rejection for a decoded future slot before the order slot is used to derive later order timing, so this check does not provide a deployment-domain boundary for compatible legacy payloads.

programs/velocity/src/instructions/keeper.rs

```
pub fn place_signed_msg_taker_order<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    let order_slot = verified_message_and_signature.slot;
    if order_slot < clock.slot.saturating_sub(500) {
        msg!(
            "SignedMsg order slot {} is too old: must be within 500 slots of current slot",
            order_slot
        );
        return Err(print_error!(ErrorCode::InvalidSignedMsgOrderParam)().into());
    }
    let market_index = matching_taker_order_params.market_index;
    [...]
}
```

Slot validation only rejects stale decoded slots

As a result, a relayer holding an old compatible Drift non-delegate signed message from an authority that also owns a funded Velocity subaccount may be able to submit it through the Velocity taker entrypoint if the bytes Borsh-decode into a valid Velocity message. The user would not have signed an intent explicitly scoped to the Velocity deployment, which could lead to unintended order placement and potential unintended fills or losses.

Remediation

Add an explicit domain separator to the signed payload (at least the Velocity program id, ideally the state/deployment or cluster domain) and verify it on-chain rather than only skipping it for Borsh decode; add compatibility tests ensuring old Drift-domain payloads are rejected.

Patch

This issue was acknowledged by Velocity Exchange.

Trigger Order Bypasses Market Fill Pause

ID	Severity	Status
OS-VEP-ADV-101	Medium	Resolved

Description

Direct perp fills enforce the market-scoped fill pause before proceeding with order fulfillment. In `fill_perp_order`, the target perp market is loaded, checked for an active or reduce-only status, and then rejected when `PerpOperation::Fill` is paused, as shown by the direct fill guard in snippet 0.

programs/velocity/src/controller/orders.rs

```rust
pub fn fill_perp_order([...]) -> VelocityResult<(u64, u64)> {
    [...]
    validate!(
        !market.is_operation_paused(PerpOperation::Fill),
        ErrorCode::MarketFillOrderPaused,
        "Market fills paused",
    )?;
    [...]
}
```

Direct perp fills reject markets with `PerpOperation::Fill` paused.

The trigger-order instruction does not apply the same market-local fill gate before dispatching into the order controller. In `handle_trigger_order`, the access control only checks that the exchange is not globally paused.

programs/velocity/src/instructions/keeper.rs

```rust
#[access_control(
    exchange_not_paused(&ctx.accounts.state)
)]
pub fn handle_trigger_order<'c: 'info, 'info>(
    ctx: Context<'info, TriggerOrder<'info>>,
    order_id: u32,
) -> Result<()> {[...]}
```

The trigger-order instruction is only gated by the global exchange pause.

Inside `trigger_order`, the controller confirms that the order is open, triggerable, not already triggered, and for a perp market, then validates user health, oracle validity, oracle divergence, and the trigger condition. After those checks, it updates the trigger-order auction parameters, increments open auction accounting when applicable, increases open bids or asks on the user position, and pays the flat perp keeper reward through the loaded perp market.

Because the trigger path mutates trigger and auction state and pays the keeper reward without the same `PerpOperation::Fill` check used by direct fills, a caller can activate a trigger order while direct fills for that market are paused. This can cause the order's auction timing to start and age during the pause, so after the market is unpaused the order may execute with already-aged auction parameters rather than beginning its execution window at that later time.

Remediation

Require `handle_trigger_order` to load the target perp market in the writable market set needed by `trigger_order`, and enforce the same market status and `!market.is_operation_paused(PerpOperation::Fill)` checks before any trigger state, auction parameters, open bid or ask accounting, or keeper rewards are updated.

Patch

Resolved in PR#270.

Trigger Order Accepts Stale Or Uncertain Oracles

ID	Severity	Status
OS-VEP-ADV-102	Medium	Resolved

Description

Trigger-order activation uses a less restrictive oracle validity policy than margin-quality checks. In `is_oracle_valid_for_action`, the `VelocityAction::TriggerOrder` branch rejects only non-positive and too-volatile oracle states, so stale-for-margin, too-uncertain, and insufficient-data-point oracle validity states can still pass this action-specific gate, as shown in the oracle policy excerpt.

programs/velocity/src/math/oracle.rs

```
pub fn is_oracle_valid_for_action([...]) -> VelocityResult<bool> {
    let is_ok = match action {
        Some(action) => match action {
            [...]
            ),
            VelocityAction::TriggerOrder => !matches!(
                oracle_validity,
                OracleValidity::NonPositive | OracleValidity::TooVolatile
            ),
            [...]
        },
        [...]
    };
    [...]
}
```

Trigger-order oracle validity policy

The `trigger_order` flow then reads the perp market oracle price data and validity, evaluates that validity with `VelocityAction::TriggerOrder`, and rejects only if that action-specific check returns false. This means the trigger predicate can be evaluated using an oracle state that would not satisfy stricter freshness or confidence requirements used elsewhere, as shown in the `trigger_order` oracle validation path.

programs/velocity/src/controller/orders.rs

```
pub fn trigger_order([...]) -> VelocityResult {
    [...]
    let mut perp_market = perp_market_map.get_ref_mut(&market_index)?;
    let (oracle_price_data, oracle_validity) = oracle_map.get_price_data_and_validity(
        MarketType::Perp,
        perp_market.market_index,
        &perp_market.oracle_id(),
        perp_market
```

```
            .market_stats
            .historical_oracle_data
            .last_oracle_price_twap,
        perp_market.get_max_confidence_interval_multiplier()?,
        perp_market.oracle_slot_delay_override,
        perp_market.oracle_low_risk_slot_delay_override,
        None,
    )?;
    let is_oracle_valid =
        is_oracle_valid_for_action(oracle_validity,
        Some(VelocityAction::TriggerOrder))?;
    validate!(is_oracle_valid, ErrorCode::InvalidOracle)?;
    [...]
}
```

Trigger-order oracle validation

After the oracle-validity check, `trigger_order` applies only the 5-minute TWAP divergence guard before it computes the trigger price and checks whether the order satisfies its trigger condition.

programs/velocity/src/controller/orders.rs

```
pub fn trigger_order([...]) -> VelocityResult {
    [...]
    let oracle_too_divergent_with_twap_5min = is_oracle_too_divergent_with_twap_5min(
        oracle_price_data.price,
        perp_market
            .market_stats
            .historical_oracle_data
            .last_oracle_price_twap_5min,
        state
            .oracle_guard_rails
            .max_oracle_twap_5min_percent_divergence()
            .cast()?,
    )?;
    validate!(
        !oracle_too_divergent_with_twap_5min,
        ErrorCode::OrderBreachesOraclePriceLimits,
        "oracle price vs twap too divergent"
    )?;
    [...]
}
```

5-minute TWAP divergence guard

Because the order state records the trigger by rewriting the trigger condition, later fill logic can treat the order as already triggered rather than re-evaluating the original predicate against a fresh oracle. A keeper can therefore activate a trigger order and receive the flat reward while the oracle

is stale for margin, too uncertain, or based on insufficient data points, and the order may remain live for later execution even if a fresh oracle would not have satisfied the original trigger condition.

Remediation

Require trigger-order activation to use oracle quality appropriate for a persistent state transition. Specifically, update the `VelocityAction::TriggerOrder` validity policy, or add an explicit guard inside `trigger_order`, so `StaleForMargin`, `TooUncertain`, and `InsufficientDataPoints` are rejected before the trigger condition is evaluated, the order is rewritten to a triggered state, and any filler reward is paid.

Patch

Resolved in PR#347.

Spot Withdraw Pre-Margin TWAP Normalizes Too-Volatile Oracle

ID	Severity	Status
OS-VEP-ADV-103	Medium	Resolved

Description

The spot withdraw path refreshes the target spot market before it performs the withdraw-specific margin check or sends tokens from the market vault. In `handle_withdraw`, the instruction loads the writable spot market, reads its oracle price data, and calls `update_spot_market_cumulative_interest` with that oracle data before later invoking `meets_withdraw_margin_requirement` and the vault transfer, as shown in the withdraw flow.

programs/velocity/src/instructions/user.rs

```
pub fn handle_withdraw<'c: 'info, 'info>([...]) -> anchor_lang::Result<()> {
    [...]
    let spot_market_is_reduce_only = {
        [...]
        controller::spot_balance::update_spot_market_cumulative_interest(
            spot_market,
            Some(oracle_price_data),}
            now,
        )?;
        [...]
    };
    [...]
}
```

Withdraw refreshes cumulative interest before the margin check

That refresh can mutate the oracle TWAP used by later oracle-validity checks. `update_spot_market_cumulative_interest` reaches `update_spot_market_twap_stats`, where the current oracle price is sanitized against the existing `last_oracle_price_twap`, new one-hour and five-minute TWAP values are computed, and the spot market's historical oracle fields are updated when the TWAP changes or enough time has elapsed.

programs/velocity/src/controller/spot_balance.rs

```
pub fn update_spot_market_twap_stats([...]) -> VelocityResult {
    [...]
    if let Some(oracle_price_data) = oracle_price_data {
        let sanitize_clamp_denominator = spot_market.get_sanitize_clamp_denominator()?;
        let capped_oracle_update_price: i64 = sanitize_new_price(
            oracle_price_data.price,
            spot_market.historical_oracle_data.last_oracle_price_twap,
            sanitize_clamp_denominator,
        )?;
```

```
        let oracle_price_twap = calculate_new_twap(
            capped_oracle_update_price,
            now,
            spot_market.historical_oracle_data.last_oracle_price_twap,
            spot_market.historical_oracle_data.last_oracle_price_twap_ts,
            ONE_HOUR,
        )?;
        let oracle_price_twap_5min = calculate_new_twap(
            capped_oracle_update_price,
            now,
            spot_market
                .historical_oracle_data
                .last_oracle_price_twap_5min,
            spot_market.historical_oracle_data.last_oracle_price_twap_ts,
            FIVE_MINUTE as i64,
        )?;
        [...]
    }
    [...]
}
```

TWAP refresh sanitizes and updates oracle TWAP state

The withdraw margin check then calculates margin and liability-oracle validity using the already-updated market state. In `meets_withdraw_margin_requirement`, the margin context ignores invalid deposit oracles but still requires all liability oracles to be valid when the user has a margin requirement or liabilities, so a liability oracle that would have been rejected as `TooVolatile` against the pre-refresh TWAP may be accepted after the same instruction refreshes the TWAP.

programs/velocity/src/state/user.rs

```
pub fn meets_withdraw_margin_requirement(
    &mut self,
    perp_market_map: &PerpMarketMap,
    spot_market_map: &SpotMarketMap,
    oracle_map: &mut OracleMap,
    margin_requirement_type: MarginRequirementType,
) -> VelocityResult<bool> {
    [...]
    if calculation.margin_requirement > 0 || calculation.get_num_of_liabilities()? > 0
    {
        validate!(
            calculation.all_liability_oracles_valid,
            ErrorCode::InvalidOracle,
            "User attempting to withdraw with outstanding liabilities when an oracle is
            invalid"
        )?;
```

```
    }
    [...]
}
```

Withdraw margin check enforces liability-oracle validity after refresh

This weakens the invalid-liability-oracle gate on withdrawals. If the only blocker is the liability oracle's volatility relative to the pre-call TWAP, a user could cause the withdraw instruction to normalize the TWAP before the margin check and then withdraw from the spot market vault in a state that a pre-mutation check would reject, creating bad-debt or insolvency exposure.

Remediation

Evaluate withdraw liability-oracle validity against the spot market's pre-refresh oracle TWAP before calling `update_spot_market_cumulative_interest`, or prevent the withdraw-side refresh from mutating oracle TWAP fields when the pre-refresh oracle is not valid for `MarginCalc`.

Patch

Resolved in PR#391.

Builder Fees Can Be Dropped When The Escrow Row Is Missing, Cleared, Or Stripped By Modify

ID	Severity	Status
OS-VEP-ADV-104	Medium	Resolved

Description

Builder-code fee attribution is represented through a `RevenueShareEscrow` order row instead of immutable fields on the `Order`. As a result, placement and fill behavior depend on whether that side row exists and remains aligned with the live order. The placement logic validates builder parameters first, then calls `add_builder_order` before the order is placed, so failure to reserve the side row can leave no durable builder-fee record for the later fill.

In `add_builder_order`, a missing escrow account or a full escrow order list does not cause builder-coded placement to fail. The function returns no revenue-share order when either condition occurs, including after logging that the escrow is full, which converts an otherwise builder-coded order into one with no escrow row to collect against.

programs/velocity/src/instructions/optional_accounts.rs

```rust
pub fn add_builder_order<'a, 'b>([...]) -> VelocityResult<Option<&'b mut
RevenueShareOrder>> {
    [...]
    match escrow.add_order(RevenueShareOrder::new(
        builder_idx,
        user.sub_account_id,
        order_id,
        builder_fee_bps,
        MarketType::Perp,
        market_index,
        RevenueShareOrderBitFlag::Open as u8,
        new_order_index as u8,
    )) {
        Ok(order_idx) => Ok(escrow.get_order_mut(order_idx).ok()),
        Err(_) => {
            msg!("Failed to add builder order, RevenueShareEscrow is full");
            Ok(None)
        }
    }
}
```

Best-effort escrow insertion returns no builder row

The `place_perp_order` path then sets the order's builder marker only when the mutable revenue-share order reference is present. If `add_builder_order` returned no row, the order is placed without `HasBuilder` the fill path therefore has no builder marker requiring an escrow account and the later fee lookup has no builder fee basis to accrue.

programs/velocity/src/controller/orders.rs

```rust
pub fn place_perp_order([...]) -> VelocityResult<PlaceOrderResult> {
    [...]
    if rev_share_order.is_some() {
        bit_flags = set_order_bit_flag(bit_flags, true, OrderBitFlag::HasBuilder);
    }
    [...]
}
```

Builder marker depends on the escrow row

The escrow row can also become detached from the actual live order slot. The row records a `user_order_index` chosen from the first currently available user order before `place_perp_order` performs its own placement flow. `revoke_completed_orders` later checks that saved index against the current user order and clears rows with no accrued fees when the indexed order is not the expected open order.

The public modify flow provides a separate downgrade path. `modify_order` cancels the existing order, rebuilds `OrderParams` from the old order and requested modifications, and then calls `place_perp_order` with no revenue-share order reference. In `merge_modify_order_params_with_existing_order`, the rebuilt params explicitly omit builder attribution, so modifying an existing builder-coded order can replace it with an otherwise similar order that no longer carries builder fee fields. Theis results in loss of builder or referrer revenue and corresponding taker fee undercharging for builder-coded orderflow.

programs/velocity/src/controller/orders.rs

```rust
fn merge_modify_order_params_with_existing_order([...]) ->
VelocityResult<Option<OrderParams>> {
    [...]
    Ok(Some(OrderParams {
        [...]
        auction_end_price,
        builder_idx: None,
        builder_fee_tenth_bps: None,
    }))
}
```

Modify rebuild drops builder attribution

Remediation

Make builder-fee attribution atomic with the placed order. Prefer storing the builder index and fee on `Order`, or create the `RevenueShareEscrow` row only after `place_perp_order` has selected the final order slot and order id. If builder fee parameters are supplied and no escrow row can be reserved, reject placement instead of returning no builder order. Update `revoke_completed_orders` to reconcile open orders by stable identifiers such as `(sub_account_id, order_id)` rather than relying

on a saved user-order slot, and make modify either preserve the existing builder attribution or require the caller to revalidate and recreate it explicitly.

Patch

Resolved in PR#309.

Self-Approved Builder Fee Bypasses The Initial-Margin Withdrawal Gate

ID	Severity	Status
OS-VEP-ADV-105	Medium	Resolved

Description

`handle_change_approved_builder` prevents the escrow authority from approving itself as the builder, but the check is limited to exact key equality. The surrounding logic then either updates an existing approved builder entry or appends a new `BuilderInfo` with the caller-supplied fee ceiling, so a user who controls a second authority can approve that separate key as a builder for the taker's escrow.

programs/velocity/src/instructions/user.rs

```
444 pub fn handle_change_approved_builder<'c: 'info, 'info>(
445     ctx: Context<'info, ChangeApprovedBuilder<'info>>,
446     builder: Pubkey,
447     max_fee_tenth_bps: u16,
448     add: bool,
449 ) -> Result<()> {
450     validate!(
451         ctx.accounts.escrow.authority != builder,
452         ErrorCode::DefaultError,
453         "Builder cannot be the same as the escrow authority"
454     )?;
455     [...]
456 }
```

`handle_change_approved_builder` only rejects the escrow authority itself.

`validate_builder_fee` verifies that the loaded revenue-share escrow belongs to the taker, loads the approved builder by index, rejects revoked builders, and compares the requested fee only against that builder's configured `max_fee_tenth_bps`. The validation path does not apply a protocol-level fee ceiling or evaluate whether the taker's account could transfer the same value under the withdrawal health rules.

programs/velocity/src/instructions/optional_accounts.rs

```
1 pub fn validate_builder_fee([...]) -> VelocityResult<Option<u16>> {
2     [...]
3     validate!(
4         escrow.fixed.authority == *user_authority,
5         ErrorCode::InvalidUserAccount,
6         "RevenueShareEscrow account must be owned by taker",
7     )?;
8
9     let builder = escrow.get_approved_builder_mut(builder_idx)?;
```

```
10
11      if builder.is_revoked() {
12          return Err(ErrorCode::BuilderRevoked);
13      }
14
15      if builder_fee > builder.max_fee_tenth_bps {
16          return Err(ErrorCode::InvalidBuilderFee);
17      }
18
19      Ok(Some(builder_fee))
```

`validate_builder_fee` checks only escrow ownership, revocation, and the builder's configured maximum.

`calculate_fee_for_fulfillment_with_match` turns the validated builder fee rate into a notional-based fee using the fill quote amount and a 100,000 denominator. Because the accepted rate is a `u16` bounded only by the approved builder's configured maximum, a high-rate can represent a large fraction of fill notional when the taker configured that builder maximum accordingly.

programs/velocity/src/math/fees.rs

```
314 pub fn calculate_fee_for_fulfillment_with_match(
315     taker_stats: &UserStats,
316     maker_stats: &Option<&mut UserStats>,
317     quote_asset_amount: u64,
318     fee_structure: &FeeStructure,
319     order_slot: u64,
320     clock_slot: u64,
321     filler_multiplier: u64,
322     reward_referrer: bool,
323     market_type: &MarketType,
324     fee_adjustment: i16,
325     builder_fee_bps: Option<u16>,
326 ) -> VelocityResult<FillFees> {
327     [...]
328     let builder_fee = if let Some(builder_fee_bps) = builder_fee_bps {
329         Some(
330             quote_asset_amount
331                 .safe_mul(builder_fee_bps.cast()?)?
332                 .safe_div(100_000)?,
333         )
334     }
335     [...]
336 }
```

`calculate_fee_for_fulfillment_with_match` derives the builder fee from fill notional and the approved rate.

The fill path treats builder fees as part of the taker's fill-side fee debit: the order controller accrues the builder fee to the revenue-share escrow order and subtracts the taker fee plus builder fee

from the taker's perp quote asset amount. For non-liquidation perp fills, the same controller selects maintenance margin for position-decreasing taker orders, while `handle_withdraw` applies `MarginRequirementType::Initial` after withdrawal accounting.

As a result, a taker account that is below initial margin but still above maintenance margin may be able to move value through a self-controlled, separately approved builder during a position-decreasing fill even though a direct withdrawal of that value would fail the initial-margin gate. The impact is limited by the fact that the taker bears the corresponding fill-side loss and maintenance/liquidation protections still apply, but the fee rail can defeat the intended distinction between risk-reducing fills and collateral withdrawals.

Remediation

Apply the same transfer-style constraint to builder fees that can materially debit the taker. In particular, after including the requested builder fee in the taker's post-fill accounting, require the taker to satisfy the initial-margin withdrawal standard before accruing or sweeping a large builder fee, or cap builder fees at a level that cannot substitute for collateral transfer.

Patch

Resolved in PR#420.

Signed Message TP/SL Sidecars Survive Main Order Soft Skip

ID	Severity	Status
OS-VEP-ADV-106	Medium	Resolved

Description

The signed-message taker-order path places optional stop-loss and take-profit sidecars before it attempts to place the main signed taker order. In `place_signed_msg_taker_order`, the stop-loss sidecar is constructed as an opposite-direction `TriggerMarket` order with `reduce_only` set and then submitted through `place_perp_order` with margin enforcement disabled. The same handler records the signed-message order id and only then submits the main taker order through `place_perp_order`. Because the main order is sequenced after the sidecar placements, a later soft skip of the main order does not roll back or prevent the already-submitted trigger sidecars within the handler's normal control flow.

`place_perp_order` treats an already-expired nonzero `max_ts` as a successful no-op by returning a default placement result instead of an error. This behavior means an expired main signed taker order can be accepted by the caller as successfully handled without inserting the main order.

programs/velocity/src/controller/orders.rs

```rust
pub fn place_perp_order([...]) -> VelocityResult<PlaceOrderResult> {
    [...]
    if max_ts != 0 && max_ts < now {
        msg!("max_ts ({}) < now ({}), skipping order", max_ts, now);
        return Ok(PlaceOrderResult::default());
    }
    [...]
}
```

`max_ts` soft skip in `place_perp_order`

The signed-message handler separately rejects bundles whose computed `max_slot` has passed, so a relayer can still submit a bundle while the signed-message slot window remains valid even if the main order's timestamp window has expired. In that case, the reduce-only TP/SL trigger sidecars may remain as standalone orders even though the entry order they were intended to protect was not inserted. Those stale triggers could later reduce or otherwise disturb the user's existing or future position, violating the expected atomicity of the signed bundle.

Remediation

Make signed-message bundle placement atomic with respect to the main taker order. Validate the main order's `max_ts` and other soft-skip outcomes before inserting TP/SL sidecars, or place the main order first and insert sidecars only when `place_perp_order` actually creates the main order. Treat default no-op placement results for the main signed order as a reason to skip or revert sidecar insertion.

Patch

Resolved in PR#308.

Signed-Message IOC Limit Orders Can Be Accepted As Resting Orders

ID	Severity	Status
OS-VEP-ADV-107	Medium	Resolved

Description

Signed-message taker placement accepts a verified signed order after checking that the first order targets a perp market and has valid auction parameters. In `place_signed_msg_taker_order`, that validation path does not also reject an order whose parameters carry the immediate-or-cancel flag before the handler records the signed-message order id and forwards the signed order parameters into `place_perp_order`.

programs/velocity/src/instructions/keeper.rs

```rust
pub fn place_signed_msg_taker_order<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    let matching_taker_order_params =
    &verified_message_and_signature.signed_msg_order_params;
    if matching_taker_order_params.market_type != MarketType::Perp
        || !matching_taker_order_params.has_valid_auction_params()?
    {
        msg!("First order must be a perp taker order");
        return Err(print_error!(ErrorCode::InvalidSignedMsgOrderParam)().into());
    }
    [...]
}
```

Signed-message taker placement validates market type and auction parameters without an IOC guard.

The ordinary direct perp placement path applies a different guard before it reaches the same order controller. In `handle_place_perp_order`, IOC parameters are rejected with `InvalidOrderIOC`, with the message indicating that IOC orders are expected to use the place-and-make or place-and-take flows rather than the standard resting-order placement flow.

programs/velocity/src/instructions/user.rs

```rust
pub fn handle_place_perp_order<'c: 'info, 'info>(
    ctx: Context<'info, PlaceOrder>,
    params: OrderParams,
) -> Result<()> {
    [...]
    if params.is_immediate_or_cancel() {
        msg!("immediate_or_cancel order must be in place_and_make or place_and_take");
        return Err(print_error!(ErrorCode::InvalidOrderIOC)().into());
    }[...]
}
```

Direct perp placement rejects IOC orders before calling the order controller.

Once the signed-message handler reaches `place_perp_order`, the controller constructs an open order from the supplied parameters and copies the IOC bit into the stored order. The same controller also defaults `max_ts` to zero for non-market and non-oracle orders when no explicit maximum timestamp is supplied, and the expiry helper treats `max_ts == 0` as non-expiring, so a signed IOC limit order can remain as an open resting order instead of being cancelled immediately.

As a result, a signed IOC limit order that direct placement would reject can be accepted through the signed-message taker path and left open for later matching. Later fills still go through normal order validation and fill controls, including the signer's limit price, so the impact is stale or unintended execution from an unexpected resting order rather than an unrestricted or price-agnostic fill.

Remediation

Apply the same IOC rejection used by direct and batch resting-order placement before `place_signed_msg_taker_order` calls `place_perp_order`, unless the signed order is routed through a place-and-take or place-and-make flow that cancels any unfilled residual.

Patch

Resolved in PR#308.

Trigger Order After Expiry Creates Settleable Keeper Reward

ID	Severity	Status
OS-VEP-ADV-108	Medium	Resolved

Description

`handle_trigger_order` exposes the trigger-order path behind only the exchange-wide pause check. After it derives the order's market type from the user account, it loads account maps with empty writable perp and spot market sets and then calls `trigger_order`. In this handler context, the target perp market is not explicitly required as a writable market before the trigger path can mutate order/open-order accounting or award the filler.

`trigger_order` updates trigger-order parameters, increments open auction and open bid/ask accounting where applicable, and calls the flat keeper reward routine using `state.perp_fee_structure.flat_filler_fee`. It does not perform the settlement or market-status gates that would reject a market in settlement before this state mutation and reward payment.

The reward path debits the order owner's perp quote and credits the filler user's perp quote when the filler is distinct from the order owner. Because the trigger path can run after the market has moved into expiry settlement, this can create a positive quote claim for a keeper without creating corresponding base exposure on that expired market. The expired-position settlement code separately allows a zero-base, positive-quote position to bypass the maintenance-margin check and settles user PnL through the market PnL pool, so the rewarded quote can be converted through the settlement flow described in the finding.

The direct placement path applies a settlement gate before placing a perp order, rejecting markets for which `is_in_settlement(now)` is true. This contrast shows the missing control: normal order placement prevents new order activity in settlement, while the trigger path can still transition dormant trigger orders and pay the keeper reward unless it applies the same market lifecycle checks.

programs/velocity/src/controller/orders.rs

```rust
pub fn place_perp_order([...]) -> VelocityResult<PlaceOrderResult> {
    [...]
    validate!(
        !market.is_in_settlement(now),
        ErrorCode::MarketPlaceOrderPaused,
        "Market is in settlement mode",
    )?;
    [...]
}
```

`place_perp_order` rejects markets already in settlement.

A public keeper could therefore extract the configured flat trigger reward after the market is closed to normal placement and fills, then settle the resulting zero-base positive quote claim from the expired market's PnL pool. Since the per-trigger amount is bounded by the configured flat filler fee and is validated against `OPEN_ORDER_MARGIN_REQUIREMENT`, this issue is limited to bounded extraction per trigger and PnL-pool headroom consumption rather than an unbounded drain.

Remediation

Before any trigger-order mutation or keeper reward payment, apply the same market lifecycle checks used by placement and fills: reject perp markets that are in settlement, require the market status to be active or reduce-only as appropriate, and enforce the market's fill-operation pause status for trigger transitions that make an order fillable. Also require the target perp market to be included in the writable market set whenever trigger handling can update open-order accounting or pay `pay_keeper_flat_reward_for_perps`.

Patch

Resolved in PR#308.

Expired Perp Transfer Live Oracle Splits Settlement PnL

ID	Severity	Status
OS-VEP-ADV-109	Medium	Resolved

Description

`transfer_perp_position` allows one signer to move a perp position between two user accounts that the same authority can sign for. HOwever, `handle_transfer_perp_position` does not reject an expired market, `MarketStatus::Settlement`, or `perp_market.is_in_settlement(now)`. The same handler then records the current oracle price from the oracle map and later uses that value as the basis for the transfer price. This means the post-expiry transfer path can value the transfer at the live oracle rather than the market's fixed settlement price.

After a market has entered settlement, other settlement-aware valuation paths use the market's `expiry_price`: `calculate_perp_position_value_and_pnl` switches to `expiry_price` for `MarketStatus::Settlement`, and `settle_expired_position` requires settlement status and computes the closeout value with the expiry price after the settlement buffer. Because the transfer path lacks the same settlement gate or expiry-price valuation, an authority controlling both accounts may separate a live-oracle gain on the source account from the matching fixed-expiry loss on the destination account. The impact remains constrained by the destination's initial-margin check, which uses the normal margin calculation path and therefore values settlement exposure at `expiry_price`.

Remediation

Reject `transfer_perp_position` whenever the perp market is expired or in settlement, using the same market-state predicate used by settlement-aware paths such as `perp_market.is_in_settlement(now)` or an explicit `MarketStatus::Settlement` check. If post-expiry transfers must remain supported, compute the transfer value with `perp_market.expiry_price` instead of the live oracle price and enforce the same settlement buffer and margin constraints used by `settle_expired_position`.

Patch

Resolved in PR#308.

A Late Vault Reward After The Final User Exit Becomes Manager Shares

ID	Severity	Status
OS-VEP-ADV-110	Medium	Resolved

Description

When the vault has positive equity but no remaining aggregate shares, `apply_rebase` recreates aggregate supply from the current vault equity instead of assigning that value to an explicit prior owner or reserve. As shown in `Vault::apply_rebase`, this zero-supply repair only updates `total_shares`. It does not restore any departed depositor shares or otherwise preserve the ownership epoch that existed before the final user withdrawal.

programs/vaults/src/state/vault.rs

```
pub fn apply_rebase(
    &mut self,
    vault_protocol: &mut Option<RefMut<VaultProtocol>>,
    vault_equity: u64,
) -> Result<Option<u128>> {
    [...]
    if vault_equity != 0 && self.total_shares == 0 {
        self.total_shares = vault_equity.cast::<u128>()?;
    }
    Ok(rebase_divisor)
}
```

Zero-supply rebase repair assigns equity to aggregate shares.

Manager ownership is represented as the residual share balance rather than as a separately stored field. In `get_manager_shares`, the manager share count is calculated from `total_shares` after subtracting user shares and, when present, protocol shares. Therefore, if a late reward makes vault equity positive while `user_shares` and protocol shares remain zero, the repaired `total_shares` are derived entirely as manager shares.

programs/vaults/src/state/vault.rs

```
pub fn get_manager_shares(
    &self,
    vault_protocol: &mut Option<RefMut<VaultProtocol>>,
) -> VaultResult<u128> {
    Ok(match vault_protocol {
        None => self.total_shares.safe_sub(self.user_shares)?,
        Some(vp) => self
            .total_shares
            .safe_sub(self.user_shares)?
            .safe_sub(vp.protocol_profit_and_fee_shares)?,
    })
```

```
363 }
```

Manager shares are derived as the residual supply.

The public `apply_rebase` instruction can trigger this repair without requiring a manager or depositor signer. The `ApplyRebase` account context requires mutable vault, vault depositor, and vault user accounts with relationship constraints, but it does not include a signer account, so any caller that can provide the constrained accounts can cause the rebase to run after the late reward is reflected in vault equity.

programs/vaults/src/instructions/apply_rebase.rs

```
33 #[derive(Accounts)]
34 pub struct ApplyRebase<'info> {
35     #[account(mut)]
36     pub vault: AccountLoader<'info, Vault>,
37     #[account(
38         mut,
39         constraint = is_vault_for_vault_depositor(&vault_depositor, &vault)?
40     )]
41     pub vault_depositor: AccountLoader<'info, VaultDepositor>,
42     #[account(
43         mut,
44         constraint = is_user_for_vault(&vault, &velocity_user.key())?
45     )]
46     pub velocity_user: AccountLoader<'info, User>,
47 }
```

Public rebase account context has no signer account.

This creates a misallocation when the final depositor exits before a reward already fixed to the vault PDA is swept into NAV. The withdrawal path burns the exiting depositor's shares and reduces both `total_shares` and `user_shares`. If the later sweep leaves positive equity with zero supply, the subsequent rebase mints aggregate shares that are treated as manager residual shares. Since the configured manager can then request and withdraw against those shares through the manager withdrawal flow, the realistic impact is limited to the semi-trusted manager redeeming value that should have remained attributable to the prior depositor epoch.

Remediation

Before allowing a withdrawal that would burn the last ownership shares, include any fixed vault-PDA receivables in NAV or move them into an explicit prior-epoch reserve so they remain attributable to the correct owners. Additionally, change the positive-equity and zero-supply branch in `apply_rebase` so it does not implicitly create manager residual shares, and assign the value to an explicit reserve or intended owner.

Patch

Resolved in PR#307.

Management-Fee Rebase Leaves Existing Protocol Shares In The Old Denomination

ID	Severity	Status
OS-VEP-ADV-111	Medium	Resolved

Description

Protocol vault accounting separates depositor shares, manager-owned shares, and protocol-owned profit or fee shares. The vault structure documents manager ownership as the remainder after subtracting `user_shares` and `protocol_profit_and_fee_shares`, so all share balances must remain in the same denomination after a rebase. In `apply_fee`, a protocol vault with an active management fee, zero current `protocol_fee`, and positive depositor equity enters the same no-protocol-fee branch used when no protocol state is present.

That branch calls the local no-protocol-fee handler even though `vault_protocol` is `Some`. Inside the handler, the fee calculation can increase `total_shares` and then invokes `apply_rebase` with `None` for the protocol state, as shown in `handle_no_protocol_fee`. The surrounding `apply_rebase` implementation divides `total_shares` and `user_shares` by the rebase divisor, and only divides `protocol_profit_and_fee_shares` when a `VaultProtocol` is supplied, so this call skips rebasing existing protocol-owned shares.

programs/vaults/src/state/vault.rs

```
pub fn apply_fee([...]) -> Result<VaultFee> {
    [...]
    let mut handle_no_protocol_fee = |vault: &mut Vault| -> Result<()> {
        [...]
        // in case total_shares is pushed to level that warrants a rebase
        vault.apply_rebase(&mut None, vault_equity)?;
        Ok(())
    };
    [...]
}
```

The handler rebases with no protocol state, leaving protocol shares out of the divisor update.

After the fee branch returns, `apply_fee` still validates ownership using the real protocol state. The manager-share calculation subtracts `user_shares` and then subtracts `vp.protocol_profit_and_fee_shares` from the rebased `total_shares`, as shown in `get_manager_shares`. Because the protocol shares can remain in the pre-rebase denomination, this check may underflow and block fee-applying actions, or it may leave the manager and protocol ownership split mis-scaled without immediately reverting.

programs/vaults/src/state/vault.rs

```rust
pub fn apply_fee([...]) -> Result<VaultFee> {
    [...]
    // this will underflow if there is an issue with protocol fee calc
    self.get_manager_shares(vault_protocol)?;
    Ok(VaultFee {
        management_fee_payment: management_fee_payment.cast::<i64>()?,
        management_fee_shares: management_fee_shares.cast::<i64>()?,
        protocol_fee_payment: protocol_fee_payment.cast::<i64>()?,
        protocol_fee_shares: protocol_fee_shares.cast::<i64>()?,
    })
}

pub fn get_manager_shares(
    &self,
    vault_protocol: &mut Option<RefMut<VaultProtocol>>,
) -> VaultResult<u128> {
    Ok(match vault_protocol {
        None => self.total_shares.safe_sub(self.user_shares)?,
        Some(vp) => self
            .total_shares
            .safe_sub(self.user_shares)?
            .safe_sub(vp.protocol_profit_and_fee_shares)?,
    })
}
```

The post-fee manager-share check subtracts the real protocol share balance.

The impact depends on the pre-existing share distribution. An initialization-valid configuration with a 99.9999% management fee, zero `protocol_fee`, and existing protocol shares where a fee-induced 10,000x rebase leaves protocol shares 10,000x too large. In that case, a later full protocol withdrawal can redeem stale-domain shares for far more vault equity than the correctly rebased protocol balance would represent, transferring manager-owned equity to the protocol. Other distributions can instead freeze the fee path through the manager-share underflow check.

Remediation

Pass the actual `vault_protocol` into every rebase a protocol vault performs, regardless of whether the current protocol annual fee is zero, and stop using a closure whose name conflates a zero current fee with absent protocol state.

Patch

Resolved in PR#403.

Manager Fee Updates Skip Strategy-Vault Initialization Bounds

ID	Severity	Status
OS-VEP-ADV-112	Medium	Resolved

Description

The timelocked manager fee-update path can queue fee settings that the vault initialization paths would not accept. In `manager_update_fees`, the non-pending branch validates that the vault is not in liquidation and that the requested timelock is positive and long enough, then stores the incoming management fee, profit share, and hurdle rate directly from the request or falls back to the current vault values. The function does not check those incoming values against the percentage bounds used during initialization before marking the vault as having a pending fee update.

When a pending update is later processed, `try_update_vault_fees` only checks whether the update is pending and whether the current timestamp has reached the queued activation timestamp. Once mature, it emits the applied update event and assigns the stored management fee, profit share, and hurdle rate into the vault, so any out-of-range value accepted by the queueing step becomes the committed vault fee policy.

programs/vaults/src/state/fee_update.rs

```
pub fn try_update_vault_fees(&mut self, now: i64, vault: &mut Vault) -> Result<()> {
    if !self.is_pending() {
        return Ok(());
    }
    if now >= self.incoming_update_ts {
        emit!(FeeUpdateRecord {[...]});
        vault.management_fee = self.incoming_management_fee;
        vault.profit_share = self.incoming_profit_share;
        vault.hurdle_rate = self.incoming_hurdle_rate;
        vault.fee_update_status = FeeUpdateStatus::None as u8;
        self.reset();
    }
    Ok(())
}
```

Mature fee updates assign the queued values directly into the vault.

This bypasses the fee bounds visible in `initialize_vault`, where ordinary vault creation rejects a management fee or profit share at or above 100% before storing either value. Also `initialize_vault_with_protocol` checks the combined manager-plus-protocol management fee and profit-share sums against the same 100% precision threshold and requires a zero hurdle rate for protocol vaults. The manager fee update path does not load a `VaultProtocol` account, so it cannot enforce the combined protocol-vault constraints when queueing or applying the update.

programs/vaults/src/instructions/initialize_vault.rs

```rust
pub fn initialize_vault<'info>(
    ctx: Context<'info, InitializeVault<'info>>,
    params: VaultParams,
) -> Result<()> {
    [...]
    validate!(
        params.management_fee < PERCENTAGE_PRECISION_U64.cast()?,
        ErrorCode::InvalidVaultInitialization,
        "management fee must be < 100%"
    )?;
    vault.management_fee = params.management_fee;

    validate!(
        params.profit_share < PERCENTAGE_PRECISION_U64.cast()?,
        ErrorCode::InvalidVaultInitialization,
        "profit share must be < 100%"
    )?;
    vault.profit_share = params.profit_share;
    [...]
}
```

Vault initialization bounds management fee and profit share below 100%.

The resulting committed state can affect normal fee accounting. `Vault::apply_fee` applies any pending fee update before computing fees, and the non-protocol management-fee branch derives the payment from depositor equity, the vault management fee, and elapsed time since the last fee update, with a cap at depositor equity minus one. An extremely large management fee after a valid activation delay can therefore drive the formula to that cap and mint manager shares that leave the depositor position at a single base unit. Similarly, an out-of-range profit share can over-allocate realized profit or fail through checked arithmetic, and a nonzero hurdle rate can enter a protocol-vault configuration that initialization rejects as unimplemented.

Remediation

Centralize fee-policy validation in a helper that accepts the target vault state, optional protocol-vault state, candidate management fee, candidate profit share, and candidate hurdle rate. Call it from both initialization paths and from `manager_update_fees` before queueing values, and call it again in `try_update_vault_fees` before committing the queued values. The helper should reject management fees and profit shares at or above 100%, reject manager-plus-protocol management-fee and profit-share sums at or above 100% for protocol vaults, and reject nonzero hurdle rates for protocol vaults.

Patch

Resolved in PR#403.

Vault Fee Timelock Applies A Newly Raised Management Fee To The Pre-Activation Epoch

ID	Severity	Status
OS-VEP-ADV-113	Medium	Resolved

Description

Strategy Vault management fee accrual is triggered through `Vault::apply_fee`, which accepts an optional `FeeUpdate` and processes a matured update before settling fees. As shown in `apply_fee`, the pending update path calls `try_update_vault_fees` before the function calculates the fee interval or share dilution, so a matured fee change can become the live vault policy at the start of the same ordinary action that settles accrued fees.

programs/vaults/src/state/vault.rs

```
pub fn apply_fee([...]) -> Result<VaultFee> {
    if let Some(ref mut fee_update) = fee_update {
        fee_update.load_mut()?.try_update_vault_fees(now, self)?;
    }
    [...]
}
```

`apply_fee` activates a pending fee update before fee settlement.

After that replacement, `apply_fee` computes elapsed time from `last_fee_update_ts` to the current timestamp, as shown by the `since_last` calculation in the same function. The management fee payment is derived from depositor equity, the live `management_fee`, and that elapsed interval; therefore, when the live fee was just raised by a matured update, unsettled time from before the timelock activation boundary is charged at the raised rate until `last_fee_update_ts` is stamped to the current action time.

programs/vaults/src/state/vault.rs

```
pub fn apply_fee([...]) -> Result<VaultFee> {
    [...]
    let mut handle_no_protocol_fee = |vault: &mut Vault| -> Result<()> {
        let since_last = now.safe_sub(vault.last_fee_update_ts)?;
        [...]
    };
    [...]
}
```

`apply_fee` then measures the full unsettled interval from `last_fee_update_ts`.

The activation helper does not create a fee epoch boundary when the timelock matures. As shown in `try_update_vault_fees`, it overwrites `management_fee`, `profit_share`, and `hurdle_rate`, clears the pending status, and resets the `FeeUpdate`, but it does not settle the old interval or set `last_fee_update_ts` to `incoming_update_ts` before installing the new parameters.

programs/vaults/src/state/fee_update.rs

```rust
pub fn try_update_vault_fees(&mut self, now: i64, vault: &mut Vault) -> Result<()> {
    if !self.is_pending() {
        return Ok(());
    }
    if now >= self.incoming_update_ts {
        [...]
        vault.management_fee = self.incoming_management_fee;
        vault.profit_share = self.incoming_profit_share;
        vault.hurdle_rate = self.incoming_hurdle_rate;
        vault.fee_update_status = FeeUpdateStatus::None as u8;
        self.reset();
    }
    Ok(())
}
```

`try_update_vault_fees` overwrites the live fee parameters without stamping an epoch boundary.

This makes the queued fee update prospective in delay but retroactive in settlement for any interval that remained unaccrued before activation. The manager is a trusted role and depositors can exit during the timelock, but the ordering still silently transfers depositor value to the manager and can defeat the intended prospective effect of the delay. Because the same activation helper also replaces `profit_share` and `hurdle_rate` before lazy profit-share realization, already-uncrystallized depositor profit may also be evaluated under the newly installed policy.

Remediation

Model fee parameters as explicit epochs. When a pending fee update matures, first settle management fees from `last_fee_update_ts` through `incoming_update_ts` using the old `management_fee`, stamp `last_fee_update_ts` to that activation boundary, then install the new fee parameters and accrue any post-activation remainder from `incoming_update_ts` to `now` under the new rate. For `profit_share` and `hurdle_rate`, define whether updates are prospective; if they are, checkpoint each depositor under the old policy or store a versioned boundary so later lazy crystallization splits profit across the old and new policy periods.

Patch

Resolved in PR#403.

Public Vault Rebase Can Strand A Protocol Withdrawal Request

ID	Severity	Status
OS-VEP-ADV-114	Medium	Resolved

Description

`Vault::apply_rebase` rescales the vault share denomination when nonzero vault equity is below `total_shares`, but it does not rescale every field that stores shares. `apply_rebase` divides `total_shares`, `user_shares`, `VaultProtocol.protocol_profit_and_fee_shares`, and the manager withdrawal request, while leaving `VaultProtocol.last_protocol_withdraw_request.shares` unchanged.

The public `apply_rebase` instruction reaches this logic through a mutable `VaultDepositor` and the vault's `User` account, and the account constraints shown in `ApplyRebase` only bind those accounts to the same vault. The instruction context does not require a signer authority for the rebase itself, so a caller can trigger the share-scale change for a protocol vault when the required same-vault accounts and canonical protocol account are supplied.

programs/vaults/src/instructions/apply_rebase.rs

```
33  #[derive(Accounts)]
34  pub struct ApplyRebase<'info> {
35      #[account(mut)]
36      pub vault: AccountLoader<'info, Vault>,
37      #[account(
38          mut,
39          constraint = is_vault_for_vault_depositor(&vault_depositor, &vault)?
40      )]
41      pub vault_depositor: AccountLoader<'info, VaultDepositor>,
42      #[account(
43          mut,
44          constraint = is_user_for_vault(&vault, &velocity_user.key())?
45      )]
46      pub velocity_user: AccountLoader<'info, User>,
47  }
```

The rebase instruction does not require a signer authority.

Protocol withdrawal requests are stored in `VaultProtocol.last_protocol_withdraw_request`, as shown by `protocol_request_withdraw` setting that field after calculating the protocol's current shares and requested shares. Because the rebase updates `protocol_profit_and_fee_shares` but not the pending request's `shares`, an already-open protocol request can remain denominated in the pre-rebase share scale while the protocol's actual share balance and the vault's total share supply have moved to the post-rebase scale.

programs/vaults/src/state/vault.rs

```rust
pub fn protocol_request_withdraw([...]) -> Result<()> {
    [...]
    if let Some(vp) = vault_protocol {
        vp.last_protocol_withdraw_request.set(
            protocol_shares_before,
            n_shares,
            withdraw_value,
            vault_equity,
            now,
        )?;
        [...]
    }
    Ok(())
}
```

Protocol withdrawal requests are stored on `VaultProtocol`.

After that mismatch, the protocol's normal request lifecycle can be blocked. The cancel path calls `apply_rebase` and then calculates shares lost from `last_protocol_withdraw_request.shares`, which can fail when the stale request is inconsistent with the rebased `total_shares`. The execute path later validates the pending request shares against the protocol's current shares before subtracting them, and the replacement path continues to see an active request because `WithdrawRequest::set` rejects a request while the existing request value is nonzero. The impact is limited to the protocol withdrawal slot.

Remediation

When `Vault::apply_rebase` receives a `VaultProtocol`, apply the same `rebase_divisor` to `vp.last_protocol_withdraw_request.shares` that is applied to `vp.protocol_profit_and_fee_shares`.

Patch

Resolved in PR#403.

Tokenized Vault Redemption Compares Inconsistent Share Domains

ID	Severity	Status
OS-VEP-ADV-115	Medium	Resolved

Description

`redeem_tokens` performs a conservation check around token redemption by comparing the holder vault depositor shares, the tokenized depositor shares, and manager shares before and after the share transfer. The initial manager-share snapshot is taken while the optional `VaultProtocol` account has been loaded and passed into `get_manager_shares`, so protocol-owned profit and fee shares are excluded from the manager-share value in that snapshot.

The redemption path then calls `tokenized_vault_depositor.redeem_tokens`, which returns the optional protocol account after taking it from the caller's option, and proceeds into `transfer_shares`. The subsequent `transfer_shares` call also returns the optional protocol account, but `redeem_tokens` discards that second return value, leaving the local provider option empty for the post-transfer conservation check.

`get_manager_shares` uses different share domains depending on whether the protocol provider is present: with no provider it returns `total_shares - user_shares`, while with a provider it also subtracts `protocol_profit_and_fee_shares`. Because the after snapshot can run with `None` after the discarded return, pre-existing protocol shares are included as manager shares even though they were excluded before, causing the before and after aggregate share totals to differ.

programs/vaults/src/state/vault.rs

```
352 pub fn get_manager_shares(
353     &self,
354     vault_protocol: &mut Option<RefMut<VaultProtocol>>,
355 ) -> VaultResult<u128> {
356     Ok(match vault_protocol {
357         None => self.total_shares.safe_sub(self.user_shares)?,
358         Some(vp) => self
359             .total_shares
360             .safe_sub(self.user_shares)?
361             .safe_sub(vp.protocol_profit_and_fee_shares)?,
362     })
363 }
```

`get_manager_shares` changes behavior based on provider presence

When those aggregates differ, `redeem_tokens` fails the `InvalidVaultSharesDetected` validation before the SPL token transfer and burn are executed. As a result, no tokens are destroyed, but ordinary redemptions may be blocked for protocol vaults that already have `protocol_profit_and_fee_shares`. In the legacy path, redemptions may also be temporarily blocked when fee accrual changes the share accounting between the snapshots.

Remediation

Make the redemption conservation check compare a single, explicit share domain across both snapshots. Either keep the returned `VaultProtocol` provider from `transfer_shares` and pass it into the after-snapshot `get_manager_shares`, or include protocol shares as a separately tracked component in both the before and after totals. Also ensure fee and profit-share mutations that can occur during redemption are either crystallized before the first snapshot or accounted for as expected deltas in the comparison.

Patch

Resolved in PR#403.

Vault Share Transfer Skips Matured Fee Updates

ID	Severity	Status
OS-VEP-ADV-116	Medium	Resolved

Description

Direct deposit follows the vault fee-update gate before mutating depositor accounting. In `deposit`, the instruction derives whether `vault.fee_update_status` has a pending update, loads the optional `FeeUpdate` account from remaining accounts, validates that the account is present and matches the expected PDA when required, and passes the same flag into account-map loading. The shared `vault.apply_fee` path then attempts to apply a matured pending update before calculating management and profit-share effects.

The share-transfer path does not mirror that gate. In `transfer_vault_depositor_shares`, the instruction performs liquidation, self-transfer, amount, and pending-withdraw checks, then calculates vault equity and calls `transfer_shares` however, it loads maps with fee-update handling disabled and passes no `FeeUpdate` account into the transfer accounting. The shared transfer routine still calls `vault.apply_fee`, applies profit share for both depositors, moves vault shares, and shifts the transferred value between the two depositors' `net_deposits`, so these mutations can occur under stale fee parameters if a pending update has already matured but has not yet been applied elsewhere.

The tokenize path repeats the same omission while moving shares into the tokenized depositor. In `tokenize_shares`, the instruction validates the depositor state and tokenized-depositor share base, calculates vault equity, calls `transfer_shares`, and then calls tokenized-depositor `tokenize_shares`. Both accounting calls receive no fee-update account even though both eventually route through `vault.apply_fee`. This allows tokenization-related share and basis accounting to proceed without first applying the matured fee schedule.

The redeem path also disables the fee-update account while unwinding tokenized shares. In `redeem_tokens`, the instruction calculates vault equity, calls tokenized-depositor `redeem_tokens`, and then transfers shares back to the ordinary vault depositor; both calls receive no fee-update account. As with tokenization, these routines perform fee and profit-share accounting through the shared vault methods, but without the pending update that direct value-moving paths require.

As a result, after a fee-update timelock has elapsed but before another instruction applies the update, an appreciated depositor may be able to transfer, tokenize, or redeem shares under the old fee schedule. Because the transfer logic also moves the current withdraw value into the recipient depositor's `net_deposits`, the recipient's basis can be stepped up before a later direct withdrawal sees the new fee configuration.

Remediation

Apply the same pending-fee-update flow used by direct deposit and withdrawal instructions to `transfer_vault_depositor_shares`, `tokenize_shares`, and `redeem_tokens`: derive `has_fee_update` from `vault.fee_update_status`, load the optional `FeeUpdate` account from remaining accounts,

call `vault.validate_fee_update`, pass `has_fee_update` into `ctx.load_maps`, and forward the mutable `fee_update` option into `transfer_shares`, tokenized-depositor `tokenize_shares`, and tokenized-depositor `redeem_tokens` before any share, token, or `net_deposits` mutation occurs.

Patch

Resolved in PR#403.

Protocol Vault Fee Accrual Can Underflow And Freeze Withdrawals

ID	Severity	Status
OS-VEP-ADV-117	Medium	Resolved

Description

`initialize_vault_with_protocol` allows protocol vaults to be created with both a manager management fee and a protocol fee as long as their annual sum is below 100%. That bound prevents an over-100% annual rate at initialization, but it does not bound the absolute amount that can accrue after multiple years without a successful fee update.

programs/vaults/src/instructions/initialize_vault_with_protocol.rs

```rust
pub fn initialize_vault_with_protocol<'info>(
    ctx: Context<'info, InitializeVaultWithProtocol<'info>>,
    params: VaultWithProtocolParams,
) -> Result<()> {
    [...]
    validate!(
        params
            .management_fee
            .saturating_add(params.vault_protocol.protocol_fee.cast::<i64>()?)
            < PERCENTAGE_PRECISION_U64.cast()?,
        ErrorCode::InvalidVaultInitialization,
        "management fee plus protocol fee must be < 100%"
    )?;
    [...]
}
```

In `Vault::apply_fee`, the branch for vaults with both nonzero management and protocol fees computes a total accrued fee from elapsed time, derives the manager's fee payment from that uncapped total, and only then caps the combined amount used for the protocol-side subtraction. If enough time has elapsed, the manager component can exceed depositor equity by itself, causing the later safe subtractions used for share-factor denominators to fail before fee accrual can complete.

Because `last_fee_update_ts` is assigned only after the fee branch completes, a failed fee application leaves the stale timestamp in place. Subsequent calls recompute the same elapsed-time fee amount, so retrying the affected action does not reduce the accrued interval or clear the failure condition.

An ordinary depositor `request_withdraw` instruction loads the vault and optional protocol state, calculates vault equity, and delegates into the depositor withdrawal flow. The surrounding depositor implementation then calls `Vault::apply_fee` before recording the withdrawal request, so this public exit path can hit the same failing fee accrual path.

programs/vaults/src/instructions/request_withdraw.rs

```rust
pub fn request_withdraw<'info>(
    ctx: Context<'info, RequestWithdraw<'info>>,
    withdraw_amount: u64,
    withdraw_unit: WithdrawUnit,
) -> Result<()> {
    [...]
    vault_depositor.request_withdraw(
        withdraw_amount.cast()?,
        withdraw_unit,
        vault_equity,
        vault,
        &mut vp,
        &mut fee_update,
        clock.unix_timestamp,
        oracle.price,
    )?;
    Ok(())
}
```

The same `apply_fee` routine is used by other deposit, withdraw, and share-movement paths in the vault and depositor state code. As a result, once the two-fee accrual reaches the failing condition, deposits and exits may remain blocked until a role-gated fee update lowers the relevant manager or protocol fee inputs. The impact is limited due to the recovery path where a trusted manager or protocol fee reduction can restore operation.

Remediation

In the two-fee `Vault::apply_fee` branch, cap the accrued combined fee before splitting it between the manager and protocol, then allocate the capped amount proportionally. Alternatively, cap each component so neither subtraction can reduce depositor equity below one unit.

Patch

Resolved in PR#403.

Vault Deposit Can Mint Zero Shares

ID	Severity	Status
OS-VEP-ADV-118	Medium	Resolved

Description

The public vault deposit handler calculates the vault equity, optionally clamps the requested amount to the remaining vault capacity, and then calls `VaultDepositor::deposit` before performing the token transfer and Velocity deposit CPI. Because the handler has no caller-provided minimum-share-out argument, the depositor signs only for an input token amount and does not bind the transaction to a minimum number of vault shares received.

`VaultDepositor::deposit` records the positive deposit in depositor and vault accounting and calls `increase_vault_shares(n_shares)` without first requiring `n_shares` to be nonzero. The share conversion used by vault deposits is the shared proportional conversion shown in `vault_amount_to_if_shares`, which returns a proportional share amount based on the input amount, total shares, and vault balance. Since this proportional conversion is floored, therefore, when `amount * total_shares < vault_equity`, the computed share amount can be zero even though the token amount is positive.

programs/velocity/src/math/insurance.rs

```rust
pub fn vault_amount_to_if_shares(
    amount: u64,
    total_if_shares: u128,
    insurance_fund_vault_balance: u64,
) -> VelocityResult<u128> {
    // relative to the entire pool + total amount minted
    let n_shares = if insurance_fund_vault_balance > 0 {
        // assumes total_if_shares != 0 (in most cases) for nice result for user
        get_proportion_u128(
            amount.cast::<u128>()?,
            total_if_shares,
            insurance_fund_vault_balance.cast::<u128>()?,
        )?
    } else {
        validate!(
            total_if_shares == 0,
            ErrorCode::InvalidIFSharesDetected,
            "assumes total_if_shares == 0",
        )?;
        amount.cast::<u128>()?
    };
    Ok(n_shares)
}
```

Shared proportional conversion can return zero shares

In that state, the deposit path can accept and transfer a positive token amount while minting no shares to the depositor. The deposited value then increases vault equity for the benefit of existing shareholders, while the new depositor has no corresponding share balance representing the transferred tokens. This requires a high share-price state and a depositor transaction without share-output protection, so the impact is limited to the affected deposit amount rather than an unrestricted vault drain.

Remediation

After computing `n_shares` for the final `deposit_amount`, require `n_shares > 0` before mutating depositor or vault accounting and before the handler transfers tokens or performs the Velocity deposit CPI. Add a caller-supplied `min_shares_out` parameter to the public deposit instruction and reject deposits whose computed shares are below that value, so the signed transaction commits to an acceptable share result.

Patch

Resolved in PR#307 and PR#339.

Vault Profit Share Can Round To Zero Shares

ID	Severity	Status
OS-VEP-ADV-119	Medium	Resolved

Description

In `VaultDepositorBase::calculate_profit_share_and_update`, the depositor's profit is measured against net deposits plus the realized-profit high-water mark, and manager and protocol profit-share amounts are computed when profit exceeds the hurdle. The function then advances `cumulative_profit_share_amount` and `profit_share_fee_paid` before the fee amounts are converted into vault shares.

`apply_profit_share` calls that helper, sums the manager and protocol token-denominated profit-share amounts, and converts the result into depositor shares with `vault_amount_to_depositor_shares`. The function does not require the converted `profit_share_shares` value to be nonzero before decreasing the depositor's shares and updating vault, manager, and protocol accounting, so a positive fee amount can be recorded even when the share conversion yields zero shares in a high-share-price state.

The ordinary deposit path reaches this helper before minting the depositor's new shares: `deposit` applies pending vault fees and then calls `self.apply_profit_share` using the current `vault_equity`, as shown in the deposit flow. The same shared helper is also used by other depositor accounting paths in the reviewed source, including withdrawal requests, share transfers, explicit profit realization, and tokenized vault depositor flows.

programs/vaults/src/state/vault_depositor.rs

```
pub fn deposit([...]) -> Result<()> {
    [...]
    let VaultFee {
        management_fee_payment,
        management_fee_shares,
        protocol_fee_payment,
        protocol_fee_shares,
    } = vault.apply_fee(vault_protocol, fee_update, vault_equity, now)?;
    let (manager_profit_share, protocol_profit_share) =
        self.apply_profit_share(vault_equity, vault, vault_protocol, now)?;
    [...]
}
```

Deposit flow applies profit share before completing

When a positive performance fee maps to zero shares, manager and protocol performance fees may be marked as paid without any corresponding share transfer. Because the depositor's high-water mark has already advanced, later profit-share calculations may no longer identify the same profit as unpaid, allowing fee undercollection from that depositor in the affected high-share-price state.

Remediation

Convert the profit-share amount to shares before mutating the high-water mark and fee-paid counters, and reject (or carry forward, or round up to one share) any positive fee that maps to zero shares, so fees cannot be recorded as paid unless manager/protocol shares are actually transferred.

Patch

Resolved in PR#403.

Tokenized Vault Redemption Bricks Future Tokenization

ID	Severity	Status
OS-VEP-ADV-120	Medium	Resolved

Description

Tokenized vault issuance relies on `last_vault_shares` as a checkpoint for the shared `TokenizedVaultDepositor`. In `TokenizedVaultDepositor::redeem_tokens`, the method applies rebases and fees, reads the current checked vault-share balance, then refreshes `last_vault_shares` to that same pre-redemption balance before computing the number of shares represented by the tokens being burned.

programs/vaults/src/state/tokenized_vault_depositor.rs

```
pub fn redeem_tokens<'a>([...]) -> Result<(u64, Option<RefMut<'a, VaultProtocol>>)> {
    [...]
    self.last_vault_shares = self.checked_vault_shares(vault)?;
    let shares_to_redeem = depositor_shares_to_vault_amount(
        tokens_to_burn.cast()?,
        mint_supply.cast()?,
        self.last_vault_shares.cast()?,
    )?;
    [...]
}
```

The `redeem_tokens` instruction then calls that helper to obtain `shares_to_transfer` and immediately transfers those shares out of the tokenized depositor into the redeeming vault depositor. The instruction validates total share conservation and the transferred amount, then burns the user's tokens, but the code path does not update `last_vault_shares` after the tokenized depositor's real `vault_shares` balance has been decreased by the transfer.

A later `tokenize_shares` call first transfers shares into the same tokenized depositor and then checks that the previous checkpoint plus the just-transferred shares equals the current pre-mint share balance. After a nonzero redemption, the checkpoint remains at the pre-redemption balance while the actual tokenized depositor balance has been reduced, so this equality no longer holds and the instruction rejects with `InvalidVaultSharesDetected`.

programs/vaults/src/state/tokenized_vault_depositor.rs

```
pub fn tokenize_shares([...]) -> Result<u64> {
    [...]
    let new_last_vault_shares = self.last_vault_shares.safe_add(shares_transferred)?;
    validate!(
        new_last_vault_shares == vault_shares_before,
        ErrorCode::InvalidVaultSharesDetected,
```

```
            "TokenizedVaultDepositor: last_vault_shares + shares_transferred !=
            vault_shares, {} != {}",
            new_last_vault_shares,
            vault_shares_before
        )?;
        let tokens_to_mint = vault_amount_to_depositor_shares(
            shares_transferred.cast()?,
            mint_supply.cast()?,
            self.last_vault_shares.cast()?,
        )?;
        [...]
    }
```

As a result, redeeming any nonzero amount can leave the shared tokenized depositor for that mint in a state where existing redemptions may continue, but future tokenized-share issuance into the same mint is blocked by the stale checkpoint.

Remediation

After the redemption share transfer completes, set `last_vault_shares` to the tokenized depositor's checked post-transfer `vault_shares`, or move the checkpoint refresh out of `TokenizedVaultDepositor::redeem_tokens` and perform it in the instruction after `transfer_shares` returns.

Patch

Resolved in PR#403.

Public Vault Rebase Can Zero Depositor Withdraw Claim

ID	Severity	Status
OS-VEP-ADV-121	Medium	Resolved

Description

The `apply_rebase` instruction can be called without a depositor authority signature. Its account context requires mutable `vault`, `vault_depositor`, and `velocity_user` accounts and checks that the depositor belongs to the vault and that the user belongs to the vault, but it does not include a signer tied to the affected depositor. As a result, any caller who can provide the matching accounts can trigger the lazy rebase path for that depositor.

programs/vaults/src/instructions/apply_rebase.rs

```
#[derive(Accounts)]
pub struct ApplyRebase<'info> {
    #[account(mut)]
    pub vault: AccountLoader<'info, Vault>,
    #[account(
        mut,
        constraint = is_vault_for_vault_depositor(&vault_depositor, &vault)?
    )]
    pub vault_depositor: AccountLoader<'info, VaultDepositor>,
    #[account(
        mut,
        constraint = is_user_for_vault(&vault, &velocity_user.key())?
    )]
    pub velocity_user: AccountLoader<'info, User>,
}
```

Permissionless `apply_rebase` account context

After the instruction computes vault equity, `VaultDepositor::apply_rebase` delegates to the shared depositor rebase and then applies the returned `rebase_divisor` to `last_withdraw_request`. The surrounding shared helper derives a power-of-ten divisor from the vault/depositor share-base difference and divides the depositor's existing share balance before updating the depositor base. The wrapper shown below applies the same divisor to the pending withdrawal request without first checking whether a nonzero request share count would round down to zero.

programs/vaults/src/state/vault_depositor.rs

```
pub fn apply_rebase(
    &mut self,
    vault: &mut Vault,
    vault_protocol: &mut Option<RefMut<VaultProtocol>>,
    vault_equity: u64,
```

```
) -> Result<Option<u128>> {
    if let Some(rebase_divisor) =
        VaultDepositorBase::apply_rebase(self, vault, vault_protocol, vault_equity)?
    {
        self.last_withdraw_request.rebase(rebase_divisor)?;
        Ok(Some(rebase_divisor))
    } else {    Ok(None)    }
}
```

Depositor rebase also rebases the pending request

`WithdrawRequest::rebase` only divides the request's `shares`. It leaves `value` unchanged. The same implementation considers a request pending when either `shares` or `value` is nonzero, so a request with shares below the divisor can become a zero-share request while still appearing active because its recorded value remains nonzero.

programs/vaults/src/state/withdraw_request.rs

```
impl WithdrawRequest {
    pub fn pending(&self) -> bool {
        self.shares != 0 || self.value != 0
    }
    pub fn rebase(&mut self, rebase_divisor: u128) -> VaultResult {
        self.shares = self.shares.safe_div(rebase_divisor)?;
        Ok(())
    }[...]
}
```

`WithdrawRequest` keeps value while rebasing shares

The later `withdraw` path checks the redeem period, calls `apply_rebase` again, and then reads `last_withdraw_request.shares` into `n_shares`. It rejects the withdrawal when `n_shares` is zero, so a request whose shares were floored away by the public rebase cannot complete through the normal withdrawal flow even though its value may still be recorded.

programs/vaults/src/state/vault_depositor.rs

```
pub fn withdraw([...]) -> Result<(u64, bool)> {
    [...]
    let n_shares = self.last_withdraw_request.shares;
    validate!(
        n_shares > 0,
        ErrorCode::InvalidVaultWithdraw,
        "No last_withdraw_request.shares found, must call request_withdraw first",
    )?;
    [...]
}
```

Withdraw rejects zero request shares

This affects depositors with small vault-share balances or small pending withdrawal-share counts when a vault rebase divisor exceeds those counts. In that case, the depositor's share balance and pending withdrawal shares can be rounded to zero by a third-party-triggered lazy rebase, while the request value remains. Cancellation resets the request and subtracts its recorded value from `total_withdraw_requested`, but it does not restore shares already rounded away by the rebase, so the claim represented by those shares may be lost, accrued to surviving shares, or left as dust.

Remediation

Do not let a lazy rebase floor an active depositor or pending withdraw claim to zero while value remains. Convert pending requests to a value-denominated claim before rebase, preserve a minimum one-share claim when the pre-rebase claim was nonzero, reject a public rebase for a depositor whose shares or request shares would floor to zero, or require the affected depositor authority to sign an explicit dust-forfeit action.

Patch

Resolved in PR#403.

Fee Accrual Rebase Can Freeze Vault Depositor Actions

ID	Severity	Status
OS-VEP-ADV-122	Medium	Resolved

Description

The depositor withdrawal lifecycle can synchronize a depositor to the current vault base before fee accrual has finished changing the vault. In `request_withdraw`, the depositor is first lazily rebased against the vault, then `Vault::apply_fee` is called, and the function later reaches `checked_vault_shares` and profit-share logic that require the depositor base to still match the vault base.

programs/vaults/src/state/vault_depositor.rs

```rust
pub fn request_withdraw([...]) -> Result<()> {
    let rebase_divisor = self.apply_rebase(vault, vault_protocol, vault_equity)?;
    let VaultFee {
        management_fee_payment,
        management_fee_shares,
        protocol_fee_payment,
        protocol_fee_shares,
    } = vault.apply_fee(vault_protocol, fee_update, vault_equity, now)?;
    let (manager_profit_share, protocol_profit_share) =
        self.apply_profit_share(vault_equity, vault, vault_protocol, now)?;
    let (withdraw_value, n_shares) = withdraw_unit.get_withdraw_value_and_shares(
        withdraw_amount,
        vault_equity,
        self.get_vault_shares(),
        vault.total_shares,
        rebase_divisor,
    )?;
    validate!(
        n_shares > 0,
        ErrorCode::InvalidVaultWithdrawSize,
        "Requested n_shares = 0"
    )?;
    let vault_shares_before: u128 = self.checked_vault_shares(vault)?;
    let total_vault_shares_before = vault.total_shares;
    [...]
}
```

Withdrawal request synchronizes before fee accrual

`Vault::apply_fee` can increase aggregate shares as part of fee accrual and then invoke a vault-level rebase from inside the fee routine. The shown no-protocol-fee branch assigns the new aggregate share total and immediately calls `Vault::apply_rebase`. The surrounding protocol-fee branches in the same function follow the same pattern when protocol fees are enabled.

programs/vaults/src/state/vault.rs

```rust
pub fn apply_fee([...]) -> Result<VaultFee> {
    [...]
    let mut handle_no_protocol_fee = |vault: &mut Vault| -> Result<()> {
        [...]
        management_fee_shares = new_total_shares
            .cast::<i128>()?
            .safe_sub(vault.total_shares.cast()?)?;
        vault.total_shares = new_total_shares;
        vault.manager_total_fee = vault
            .manager_total_fee
            .saturating_add(management_fee_payment.cast()?);
        // in case total_shares is pushed to level that warrants a rebase
        vault.apply_rebase(&mut None, vault_equity)?;
        Ok(())
    };
    [...]
}
```

Fee accrual can invoke vault rebase

`Vault::apply_rebase` increments `vault.shares_base` when nonzero vault equity is below `total_shares` and the calculated exponent difference is nonzero. Because this rebase happens inside `apply_fee`, after the caller has already synchronized the depositor in the affected lifecycle paths, the vault base can advance without the current depositor account being synchronized to the new base.

The depositor share helpers enforce that the depositor's stored share base equals the vault's current `shares_base` before returning, increasing, or decreasing shares. If fee accrual advances the vault base between the depositor rebase and these helpers, the subsequent `checked_vault_shares` or `decrease_vault_shares` call aborts with `InvalidVaultRebase`.

The `cancel_withdraw_request` and `withdraw` implementations use the same pattern of depositor rebase before fee accrual and then later perform base-checked share operations. Once accrued fees are large enough to trigger this fee-induced aggregate rebase, ordinary request, cancel, and withdraw actions may remain unavailable from the same pre-state until a trusted recovery path or code change corrects the ordering.

Remediation

Ensure fee accrual cannot leave the active depositor at a stale share base before any base-checked share operation. Apply `Vault::apply_fee` before the depositor lazy rebase in depositor lifecycle paths, or have `apply_fee` communicate whether it rebased the vault so callers can immediately re-run depositor and pending-withdraw-request rebase logic before calling `checked_vault_shares`, `increase_vault_shares`, or `decrease_vault_shares`.

Patch

Resolved in PR#403.

Full Insurance-Fund Unstake Cancel Can Burn The Staker's Shares

ID	Severity	Status
OS-VEP-ADV-123	Medium	Resolved

Description

A full insurance-fund unstake request can be canceled through the ordinary IF staker cancel instruction after `last_withdraw_request_shares` has been recorded. In `cancel_request_remove_insurance_fund_stake`, the controller applies rebases, computes `if_shares_lost`, subtracts that amount from the caller's stake, and subtracts the same amount from the market's aggregate insurance-fund share counters before clearing the withdraw-request fields. For a sole staker whose pending request covers the entire share supply, this cancel path can therefore remove the user's whole share position without any corresponding token payout.

The edge case arises from the share recomputation used by `calculate_if_shares_lost`. That helper prices the request as if the requested value were withdrawn and then converted back into shares using the remaining share supply and remaining vault balance. When the request covers the full share supply, the remaining `total_if_shares` passed into `vault_amount_to_if_shares` is zero; because full requests are capped below the vault balance at request time, the remaining vault balance can still be positive. As shown in `vault_amount_to_if_shares`, a positive vault balance follows the proportional-conversion branch, so a zero share supply produces zero restored shares rather than preserving the staker's original shares.

programs/velocity/src/math/insurance.rs

```rust
pub fn vault_amount_to_if_shares(
    amount: u64,
    total_if_shares: u128,
    insurance_fund_vault_balance: u64,
) -> VelocityResult<u128> {
    // relative to the entire pool + total amount minted
    let n_shares = if insurance_fund_vault_balance > 0 {
        // assumes total_if_shares != 0 (in most cases) for nice result for user
        get_proportion_u128(
            amount.cast::<u128>()?,
            total_if_shares,
            insurance_fund_vault_balance.cast::<u128>()?,
        )?
    } else {
        // must be case that total_if_shares == 0 for nice result for user
        validate!(total_if_shares == 0, [...])?;
        amount.cast::<u128>()?
    };
    Ok(n_shares)
}
```

Positive vault-balance conversion returns zero shares when total shares are zero

As a result, a sole staker who changes their mind after requesting a full unstake may have the pending shares treated as lost on cancel. The cancel flow resets the request and updates accounting, but the vault token amount is not paid out by this path. The underlying value remains in the insurance-fund vault while no share claim is restored for the affected staker, leaving the value detached from that staker's position and potentially changing the balance against which later stakers mint shares.

Remediation

Handle the full-supply cancel case before subtracting lost shares. If `last_withdraw_request_shares` equals the market's active insurance-fund share supply and `last_withdraw_request_value` is positive, either restore the original pending share count directly or reject the cancel instead of recomputing restored shares from a zero remaining supply.

Patch

Resolved in PR#339.

Funding Pre-Gate TWAP Normalizes Too-Volatile Oracle

ID	Severity	Status
OS-VEP-ADV-124	Medium	Resolved

Description

The funding-rate update path refreshes PerpMarket oracle-derived state before the funding gate is evaluated. In the surrounding keeper handler, the market calls `update_oracle_derived_stats`, which can update `historical_oracle_data.last_oracle_price_twap`, and then enters `update_funding_rate`. This means the later funding block check can observe an oracle TWAP that was already moved during the same instruction rather than the TWAP that existed at instruction entry.

Inside `update_funding_rate`, the AMM is also refreshed before `oracle::block_operation` is called, and the return value of that gate determines whether the function continues to mutate funding state. The ordering in `update_funding_rate` leaves the funding gate downstream of the refresh work it is intended to guard.

The volatility classification used by the gate compares the current oracle price against `last_oracle_twap` and marks the oracle `TooVolatile` when their ratio exceeds the configured guard rail. Because the funding path can update the stored TWAP before the gate calls this classifier, an oracle that would have been too volatile against the entry-state TWAP may instead be evaluated against a normalized TWAP.

If this occurs, `block_operation` may fail to block a funding update even though the pre-refresh oracle state was too volatile for funding. The resulting impact is limited to cumulative funding movement through the normal funding-rate path and remains bounded by the protocol's funding-rate limits, but the volatility guard does not protect the specific case it is meant to reject.

Remediation

Evaluate the funding volatility gate against the oracle TWAP captured at instruction entry, before `update_oracle_derived_stats`, `refresh_amm_for_funding_gate`, or any other in-instruction refresh can move it. Alternatively, reorder the funding path so the `block_operation` volatility decision is made before the refresh and only refreshes state after the gate passes.

Patch

Resolved in PR#421.

Spot Swap Pre-Band TWAP Normalizes Price Checks

ID	Severity	Status
OS-VEP-ADV-125	Medium	Resolved

Description

The spot swap finalization path in `handle_end_swap` calculates and applies the swap's balance changes before invoking the price-band validation for the swap. This means the band check is not evaluated against an immutable TWAP reference captured at instruction entry. The refresh moves the reference the band is measured against toward the current price, so a swap priced outside the band as it stood at instruction entry is brought inside it by the same instruction that is supposed to be validating it.

This weakens the intended protection against underpriced spot swaps and can allow a user to complete a swap at a price the pre-refresh band would reject, potentially leaving the account with negative fair-value equity that must later be resolved through liquidation or bad debt handling.

Remediation

Snapshot the oracle TWAP at instruction entry and evaluate the price band against that snapshot, or perform any refresh only after the band has passed.

Patch

Resolved in PR#421.

Liquidation Pre-Band TWAP Normalizes Price Checks

ID	Severity	Status
OS-VEP-ADV-126	Medium	Resolved

Description

A public liquidation refreshes the oracle TWAP before its own price-band check, so a liquidation the matched control rejects proceeds and transfers collateral. In `liquidate_perp`, the controller imports both `update_amm_and_check_validity` and `is_oracle_too_divergent_with_twap_5min`, placing the oracle refresh and the price-band check in the same liquidation flow.

Because the same liquidation instruction refreshes the five-minute TWAP before using it as the divergence-band reference, the measured divergence can be reduced relative to the instruction-entry state. In the divergent-oracle condition, this can allow a liquidation that would have failed the band check on the pre-refresh TWAP to proceed through the liquidation path. Since the liquidation path reduces the user's perp position and prices the transfer using liquidation logic, a public liquidator may seize collateral at a liquidation discount when the oracle is depressed relative to realizable external value and the victim would have remained solvent at a fair price.

Remediation

Evaluate the divergence band against the oracle TWAP as it stood at instruction entry, before any refresh this instruction performs.

Patch

Resolved in PR#421.

Perp Fill Pre-Band TWAP Normalizes Price Checks

ID	Severity	Status
OS-VEP-ADV-127	Medium	Resolved

Description

Public perp fills refresh market-derived oracle state before applying the fill path's own oracle-divergence and fill-price band checks. In `fill_perp_order`, the handler calls the market oracle-derived-stats update, then reads `last_oracle_price_twap_5min` into the local `oracle_twap_5min` used by `is_oracle_too_divergent_with_twap_5min` after `fulfill_perp_order` returns a nonzero fill, the same captured value is passed to `validate_fill_price_within_price_bands`. The fulfillment path in `fulfill_perp_order` then routes the order and executes AMM or matched-maker fills after this pre-fill state preparation.

Because the validation reference is captured after the refresh, the same public fill instruction can move the 5-minute oracle TWAP toward the current oracle price before measuring whether that oracle is too divergent and whether the eventual fill price remains inside the TWAP band. This makes the band check consume reference state that the validating instruction has already normalized, instead of the reference state that existed when the instruction began.

The TWAP mutation is performed by `update_oracle_twap`, which normalizes the exchange oracle price against the reserve price, sanitizes the new datapoint, calculates both funding-period and five-minute oracle TWAP values, and writes the new five-minute TWAP back into historical oracle data. As a result, a public filler may be able to complete a perp fill at a price that would have failed the divergence or fill-price band if evaluated against the entry-state TWAP. When that occurs, position and quote balances are transferred between the taker and maker, and fill-related market state is updated, using a reference price condition that the protocol intended to reject before execution.

Remediation

Snapshot the oracle TWAP at instruction entry and evaluate both the divergence band and the fill-price band against that snapshot, or perform any refresh only after the bands have passed.

Patch

Resolved in PR#421.

Interest Pause Catch-Up Credits Late Balances For The Entire Paused Interval

ID	Severity	Status
OS-VEP-ADV-128	Medium	Resolved

Description

Market-level spot interest updates can be paused independently from deposits and withdrawals. In `update_spot_market_cumulative_interest`, the paused path updates TWAP statistics and returns successfully before calling the interest calculation or advancing `last_interest_ts`, so ordinary deposit and withdrawal flows that invoke this updater continue without checkpointing the interest timestamp during the paused interval.

When the pause is lifted, `calculate_accumulated_interest` computes `time_since_last_update` from the frozen `last_interest_ts` and derives borrow and deposit interest from the market's current utilization and cumulative interest indexes. Because deposits and withdrawals may have changed the market balances while interest updates were paused, the catch-up update applies the full elapsed interval to the final balance snapshot rather than to balances that were present throughout that interval.

As a result, a user who deposits immediately before the first unpaused update can receive deposit interest for time when the user did not supply funds, diluting depositors who actually held balances during the pause.

Remediation

When `SpotOperation::UpdateCumulativeInterest` is paused, checkpoint `spot_market.last_interest_ts` at the pause boundary or otherwise ensure the paused interval is not accrued against balances that arrive after the pause begins.

Patch

Resolved in PR#395.

Delisting Expiry Price Counts Unmoved Fee-Pool Value

ID	Severity	Status
OS-VEP-ADV-129	Medium	Resolved

Description

During expired-market settlement, `settle_expired_market` first transfers only the budgeted portion of the AMM fee pool into the market PnL pool, where that budget is capped by positive `total_fee_minus_distributions` and the available fee-pool balance. After that limited transfer, the same function computes `total_excess_balance` from the PnL pool plus the remaining AMM fee pool and passes that value into expiry-price calculation.

The expiry-price solver treats `total_excess_balance` as settlement capacity when deriving the best expiry price against the market's net AMM base and quote amounts. Because the caller can include fee-pool value that was not actually moved into the PnL pool, `calculate_expiry_price` can set an expiry price that assumes more payable balance than the later user settlement path can access.

Expired user positions are settled through `update_pnl_pool_and_user_balance`, which pays positive user PnL only from `market.pnl_pool` and validates that the full requested PnL can be paid. If the PnL pool has been depleted by earlier settlements, a later winner whose claim was priced using unreachable fee-pool value reverts with `InsufficientPerpPnlPool` instead of receiving a partial payout.

This can strand the remaining winning positions during wind-down. The expired-position settlement flow closes user base before settling PnL, but the PnL-pool shortfall causes the transaction to revert, leaving those users with non-zero base positions. The final delisting handler requires long base, short base, and the number of users with base to be zero before moving the market from `Settlement` to `Delisted`, so an unpaid tail winner can keep the market in `Settlement` indefinitely.

Remediation

Make the expiry-price input match the funds that expired-position settlement can actually pay. In `settle_expired_market`, compute `total_excess_balance` from the PnL pool plus the fee-pool amount that was actually transferred into the PnL pool, using the same `min(total_fee_minus_distributions.max(0), fee_pool)` cap, or transfer the full fee-pool amount before solving and settlement if that is the intended source of user payouts.

Patch

Resolved in PR#345.

Equity Breaker Blocks Strictly Reducing Spot Swaps And Forces Liquidation Loss

ID	Severity	Status
OS-VEP-ADV-130	Medium	Resolved

Description

`handle_end_swap` rejects a tripped authority before it classifies whether the swap increases risk or strictly reduces existing spot exposure. The function loads `user_stats` and applies the authority-wide equity breaker near the top, then only later reads the signed input and output spot balances, computes `in_position_is_reduced` and `out_position_is_reduced`, and applies the explicit `SwapReduceOnly` checks. This ordering prevents the function from distinguishing a new borrow or deposit from a swap that consumes an existing deposit to repay an existing borrow.

The result is that the breaker can block the account's own recovery path even when the route is bounded to reduce both affected positions. The impact is a forced liquidation penalty for an otherwise curable account, with value transferred to the third-party liquidator because self-cure is rejected before reducer classification runs.

Remediation

Move the equity-breaker decision until after `handle_end_swap` has computed the signed pre- and post-swap positions and determined whether the input and output legs are strictly reducing. Permit swaps that are bounded by the existing input deposit and output borrow, including the corresponding `SwapReduceOnly` reducer case, and apply the breaker rejection only to routes that create, increase, or import exposure.

Patch

Resolved in PR#415.

Spot Liabilities Are Absent From The Equity-Floor Metric

ID	Severity	Status
OS-VEP-ADV-131	Medium	Resolved

Description

The equity-floor breaker uses the cross-margin `total_collateral` value as the value being compared to `User.equity_floor`. In `handle_trip_equity_floor_breaker`, the handler recalculates margin with an `Initial` margin context under strict pricing, then trips the authority-level breaker only if `user.is_below_equity_floor` reports that this `total_collateral` is below the stored floor. The `UserStats` comments confirm that this permissionless trip sets an authority-wide flag that rejects risk-increasing activity until an admin reset.

The root cause is that `total_collateral` is a margin numerator rather than a net-equity metric. In `calculate_margin_requirement_and_total_collateral_and_liability_info`, spot deposits add token value into cross-margin total collateral, while spot borrows add margin requirement and liability metadata. The borrow branches also track liability value under the `velocity-rs` feature, but the on-chain `total_collateral` accumulator used by the floor check is not reduced by the unweighted spot liability value.

Because the breaker uses `MarginRequirementType::Initial`, the same comparison can also be lower than true equity for accounts with positive unrealized PnL. As shown in `calculate_perp_position_value_and_pnl`, positive weighted unrealized PnL is capped for initial margin, so the value fed into `total_collateral` can reflect initial-margin constraints rather than the account's full unweighted economic equity.

programs/velocity/src/math/margin.rs

```
pub fn calculate_perp_position_value_and_pnl([...]) -> VelocityResult<(u128, i128,
u128, u128)> {
    [...]
    if margin_requirement_type == MarginRequirementType::Initial {
        // safety guard for dangerously configured perp market
        weighted_unrealized_pnl =
        weighted_unrealized_pnl.min(MAX_POSITIVE_UPNL_FOR_INITIAL_MARGIN);
    }
    [...]
}
```

Initial-margin calculations cap positive unrealized PnL before it contributes to collateral.

As a result, an account can appear above the floor to the on-chain equity-floor checks even when spot borrow liabilities have reduced its actual net equity below the configured floor. Since the strict initial-margin value can be asset-weighted, min-priced, and capped for positive unrealized PnL, a permissionless caller may be able to trip the authority-wide breaker against an account whose true net equity is still above the configured floor, freezing sibling subaccounts until the configured admin reset path clears the flag.

Remediation

Use a dedicated net-equity metric for every equity-floor consumer instead of `MarginCalculation.total_collateral`. The metric should value spot assets and PnL on an unweighted economic basis, subtract unweighted spot liability value, and avoid initial-margin-only transformations such as strict min-pricing and the positive-uPnL cap when determining whether the floor covenant has been breached. Apply the same metric consistently in the on-chain floor checks, permissionless breaker, SDK, and guard bot.

Patch

Resolved in PR#415.

Zero Spot-Oracle TWAP Timestamp Collapses The First Price Band

ID	Severity	Status
OS-VEP-ADV-132	Medium	Resolved

Description

A freshly initialized spot market can enter its first price-banded operation with `historical_oracle_data.last_oracle_price_twap_ts` still zero while the stored oracle TWAP is nevertheless treated as the reference for price-band evaluation. The root cause is that this uninitialized timestamp state is not rejected before the banded operation relies on the stored TWAP, so the launch-time reference can degenerate instead of requiring a genuinely initialized TWAP.

The validity layer that consumes the stored TWAP receives only the TWAP value as `last_oracle_twap`. It does not receive or check the corresponding TWAP timestamp. In `oracle_validity`, volatility is classified by comparing the live oracle price against that TWAP value, so a zero or otherwise uninitialized TWAP timestamp is not independently rejected by this layer.

As a result, the first price-banded spot operation after market launch may run without an effective price guard in either direction, enabling that both depressed collateral observations and inflated liability observations to pass during the narrow launch window. The first successful write advances the timestamp, but if the initial write was seeded from a degenerate reference, the resulting TWAP level may remain a corrupted reference for later band evaluations.

Remediation

Treat a zero oracle-TWAP timestamp as an invalid reference and reject price-banded operations until the TWAP has been genuinely initialized, rather than allowing the band to degenerate. Seed the TWAP level as well as the timestamp on first write so the repaired state carries a meaningful reference.

Patch

Resolved in PR#346.

Signerless Tokenized-Vault Rebase Erases Live SPL Backing

ID	Severity	Status
OS-VEP-ADV-133	Medium	Resolved

Description

The vaults program exposes `apply_rebase_tokenized_depositor` as a public instruction and forwards it to the tokenized-depositor rebase handler. The tokenized-depositor rebase instruction carries no signer in its account context: its constraints bind the tokenized depositor and the Velocity user to the Vault, but require no token holder, manager, delegate, or admin. It calls the raw truncating share-rebase trait method (`vault_amount_to_if_shares`) directly, so any caller can drive the tokenized depositor's shared backing shares down to zero through truncation while the tokenized-vault SPL mint supply remains live and outstanding. Once the backing is floored, every holder computes zero redeemable shares against a live token balance, and redemption aborts before it can burn tokens, so the position cannot be exited even after the underlying portfolio recovers.

Any unprivileged caller can permanently strand every holder of a tokenized vault position. The SPL tokens remain in holders' wallets and continue to represent a claim on paper, but the on-chain backing has been floored to zero, so redemption computes zero and aborts rather than burning.

Remediation

Require an appropriate signer on `ApplyRebaseTokenizedDepositor`, such as the vault manager, an authorized delegate, or another authority consistent with the protocol's intended rebase permissions. In addition, route the tokenized-depositor rebase through guarded logic that rejects any rebase which would reduce a positive tokenized depositor share balance, or any positive pending claim represented by live SPL supply, to zero through truncation. The guard should be applied in the tokenized rebase path itself, not only in the ordinary depositor path, so `apply_rebase_tokenized_depositor` cannot call the raw shared rebase method in a way that strands token holders.

Patch

Resolved in PR#347.

Projected Routing Omits Refresh Accounting And Prioritizes A Worse Maker

ID	Severity	Status
OS-VEP-ADV-134	Medium	Resolved

Description

The affected perpetual order flow ranked fulfillment methods using a routing quote that could differ from the quote used during execution. In `fulfill_perp_order`, fulfillment methods are selected before the loop that executes each selected AMM or maker fill. The repaired flow in `fulfill_perp_order` first derives the market-making oracle data, applies the AMM refresh to the real `market.amm`, refreshes cached spread state against the projected reserve price, and only then calls `determine_perp_fulfillment_methods`. Before this change, the routing phase used a scratch AMM projection that matched the refreshed curve fields but omitted refresh accounting transitions, so the ranking could be based on stale spread inputs while execution later quoted from the real AMM after `Quoter::setup`.

`project_and_apply` is the shared fill-path refresh primitive used to make routing and execution observe the same projected AMM state. It returns early when the AMM has already been refreshed for the slot, otherwise computes the post-refresh projection, applies it in place, and advances `last_update_slot` only when the oracle and affordability gates allow it. This matters because the selected fulfillment ranking is not rerun after execution setup; once a maker is ranked ahead of the AMM during routing, later correction of the AMM quote is too late to change that ordering.

programs/velocity/src/vlp/amm/refresh.rs

```
311 pub fn project_and_apply(
312     amm: &mut crate::vlp::amm::AMM,
313     inputs: &repeg::ProjectionInputs,
314     mm_oracle_price_data: &MMOraclePriceData,
315     oracle_validity: Option<OracleValidity>,
316     slot: u64,
317 ) -> VelocityResult<()> {
318     if amm.last_update_slot >= slot {
319         return Ok(());
320     }
321     let projection =
322         repeg::project_post_refresh_scalar(amm, inputs, mm_oracle_price_data,
        oracle_validity)?;
323     projection.apply_to(amm)?;
324     // Same bump gates as `snap_to_oracle`: the oracle must be fresh enough
325     // for low-risk fills and the affordability floor must have accepted the
326     // debit, otherwise the AMM stays marked stale for same-slot freshness
327     // gates downstream.
328     if let Some(validity) = oracle_validity {
```

```
329             if is_oracle_valid_for_action(validity,
                Some(VelocityAction::FillOrderAmmLowRisk))?
330                 && !projection.rejected_due_to_affordability
331             {
332                 amm.last_update_slot = slot;
333             }
334         }
335         Ok(())
336     }
```

`project_and_apply` shares slot-idempotent refresh behavior

The omitted transitions are relevant because the spread calculation consumes both `total_fee_minus_distributions` and `net_revenue_since_last_funding` from the AMM when updating quote state. `ProjectedAmmState::apply_to` writes the refreshed curve fields and debits the refresh cost from both accounting fields. A scratch projection that updated reserves and peg but did not apply those debits could therefore produce different spread state from the real execution path even when the projected reserves matched.

programs/velocity/src/vlp/amm/math/repeg.rs

```
538 pub fn apply_to(&self, amm: &mut AMM) -> VelocityResult<()> {
539     if !self.applied {
540         return Ok(());
541     }
542     amm.apply_k_update(&cp_curve::UpdateKResult {
543         sqrt_k: self.sqrt_k,
544         base_asset_reserve: self.base_asset_reserve,
545         quote_asset_reserve: self.quote_asset_reserve,
546     })?;
547     amm.peg_multiplier = self.peg_multiplier;
548     amm.total_fee_minus_distributions =
549         amm.total_fee_minus_distributions.safe_sub(self.cost)?;
550     amm.net_revenue_since_last_funding = amm
551         .net_revenue_since_last_funding
552         .safe_sub(self.cost.cast::<i64>()?)?;
553     Ok(())
554 }
```

`apply_to` debits refresh cost from spread-relevant accounting

This divergence can allowe a public maker to be prioritized over a cheaper post-refresh AMM quote on the voluntary taker path, causing the taker to pay more than AMM-only execution while the maker received the fill. The same issue is also present in the permissionless `liquidate_perp_with_fill` path.

Remediation

Apply the AMM refresh to the real `market.amm` before fulfillment-method ranking, and use the same slot-idempotent implementation as execution setup so routing and execution share the same peg, reserve, k, `total_fee_minus_distributions`, and `net_revenue_since_last_funding` state.

Patch

Resolved in PR#324.

Same-Slot Oracle Freshness Reuses The Prior Sample's Validity

ID	Severity	Status
OS-VEP-ADV-135	Medium	Resolved

Description

`PerpMarket::is_recent_oracle_valid` reduces oracle freshness to whether the oracle was valid at the last AMM update and whether the AMM update slot equals the current slot. It does not bind that cached validity result to the specific oracle price or confidence sample later consumed by settlement or sweeping, as shown by the same-slot helper.

`settle_pnl` first reads an oracle price for market checks and unrealized PnL, then, when curve updates are enabled, treats `is_recent_oracle_valid` as a healthy-oracle shortcut. If that shortcut returns true, the function does not call the current validity path before continuing with PnL settlement, so a same-slot AMM refresh can cause settlement to rely on the earlier cached oracle verdict rather than the sample currently stored in the oracle map.

`handle_update_pyth_lazer_oracle` verifies a signed Lazer message and then writes the oracle account's price, posted slot, publish time, exponent, and confidence. Because the write stamps `posted_slot` with the current slot and updates price and confidence independently of the AMM's cached `last_oracle_valid`, a later Lazer update in the same slot can replace a previously acceptable sample with one whose confidence would make it `TooUncertain` if evaluated at the point of use.

The same pattern also affects the permissionless `sweep_perp_market_fees` path. It uses the same `is_recent_oracle_valid` shortcut before proceeding to fee sweeping. As a result, PnL settlement and fee sweeping may consume a current oracle price and confidence that the protocol would reject under the normal validity checks, allowing the PnL pool accounting to be settled or swept at a currently too-uncertain price.

Remediation

Recompute oracle validity against the exact oracle sample read by `settle_pnl` and `sweep_perp_market_fees` before using that price for settlement or fee sweeping, even when the AMM was updated in the same slot. Alternatively, make oracle rewrites within a slot invalidate or version the cached AMM validity state so `last_oracle_valid` cannot be reused after the underlying Pyth Lazer price or confidence changes.

Patch

Resolved in PR#344.

Expiry Price Solves Against A Cost Basis That Omits Net Unsettled Funding

ID	Severity	Status
OS-VEP-ADV-136	Medium	Resolved

Description

`calculate_expiry_price` solves the expiry price from the AMM's net base position, the available excess balance, and a caller-supplied `quote_asset_amount`. In `calculate_expiry_price`, the settlement cost-basis input is the raw quote amount, and the function does not include `net_unsettled_funding_pnl` in the value it solves against.

programs/velocity/src/vlp/amm/math/amm.rs

```rust
pub fn calculate_expiry_price([...]) -> VelocityResult<i64> {
    if amm.base_asset_amount_with_amm.abs() < order_step_size.cast::<i128>()? {
        return Ok(target_price);
    }
    [...]
    let best_expiry_price = -(quote_asset_amount
        .safe_sub(total_excess_balance.cast::<i128>()?)?
        .safe_mul(PRICE_TIMES_AMM_TO_QUOTE_PRECISION_RATIO_I128)?
        .safe_div(amm.base_asset_amount_with_amm)?)
    .cast::<i64>()?;
    let expiry_price = if amm.base_asset_amount_with_amm >= 0 {
        // net longs only get as high as oracle_price
        best_expiry_price.min(target_price).saturating_sub(1)
    } else {
        // net shorts only get as low as oracle price
        best_expiry_price.max(target_price).saturating_add(1)
    };
    Ok(expiry_price)
}
```

Expiry-price solve omits unsettled funding

This differs from the canonical valuation helpers. `calculate_net_user_cost_basis` combines `quote_asset_amount` with `net_unsettled_funding_pnl`, and `calculate_net_user_pnl` adds that combined basis to the AMM base-asset value, so the funding component is part of the market's aggregate user PnL accounting. The expiry path then passes only `market.quote_asset_amount` into that solver. In `settle_expired_market`, the function calculates the available pool balance, calls `calculate_expiry_price` with the raw quote amount, and commits the returned value as `market.expiry_price` before moving the market into `Settlement`.

programs/velocity/src/vlp/amm/refresh.rs

```rust
pub fn settle_expired_market([...]) -> VelocityResult {
    [...]
    let expiry_price = amm::calculate_expiry_price(
        &market.amm,
        target_expiry_price,
        total_excess_balance,
        market.quote_asset_amount,
        market.order_step_size,
    )?;
    market.expiry_price = expiry_price;
    market.status = MarketStatus::Settlement;
    [...]
}
```

Market settlement passes raw quote amount

Expired user settlement folds pending funding into the user's position before valuing the expired position. The surrounding `settle_expired_position` flow first calls `settle_funding_payment`, then cancels orders, values the position at `perp_market.expiry_price`, updates the position and market, and attempts to pay the resulting PnL from the PnL pool.

programs/velocity/src/controller/pnl.rs

```rust
pub fn settle_expired_position([...]) -> VelocityResult {
    [...]
    settle_funding_payment(
        user,
        user_key,
        perp_market_map.get_ref_mut(&perp_market_index)?.deref_mut(),
        now,
    )?;
    [...]
}
```

Expired-position settlement settles funding first

Because funding settlement moves pending funding into the user's quote while reducing `market.net_unsettled_funding_pnl`, the claims ultimately honored during expired-position settlement are based on the combined `quote_asset_amount + net_unsettled_funding_pnl` basis, while the committed expiry price was solved using only `quote_asset_amount`. When the solve binds and `net_unsettled_funding_pnl` is positive, the committed price can authorize aggregate winner claims above the backing pools by that omitted funding amount; a negative funding balance instead makes the solve conservative. The inspected PnL-pool payment path validates that the pool can cover the full positive PnL and raises `InsufficientPerpPnlPool` when it cannot.

The `SettleFunding` account context also exposes only `state` and a mutable `user` account in the fixed excerpt, without an authority signer or ownership constraint shown there. The surrounding handler loads the supplied user and remaining perp markets, then calls `settle_funding_payments`, so funding settlement appears callable for an arbitrary user account subject to the other instruction-level checks.

programs/velocity/src/instructions/keeper.rs

```
3554 pub struct SettleFunding<'info> {
3555     pub state: AccountLoader<'info, State>,
3556     #[account(mut)]
3557     pub user: AccountLoader<'info, User>,
3558 }
```

Funding settlement account context

As a result, after a market is finalized with positive net unsettled funding, earlier expired-position settlements may consume the available PnL pool at the inflated expiry price, while later users can fail settlement once the pool cannot cover their full authorized PnL. If those failed users retain non-zero base positions, the market wind-down can remain stuck before delisting, and repricing after early settlements would not recover value already paid out.

Remediation

Pass `calculate_net_user_cost_basis` (with arguements as `(market.quote_asset_amount, market.net_unsettled_funding_pnl)`) into `calculate_expiry_price` instead of passing `market.quote_asset_amount` directly. Change the expiry-pricing API to accept the market or an explicitly named combined cost-basis value so future callers cannot omit the funding component silently.

Patch

Resolved in PR#345.

Cancel-Withdraw Guard Precedes The Sole-Owner Escape Hatch And Forces A Stale-Value Exit

ID	Severity	Status
OS-VEP-ADV-137	Medium	Resolved

Description

`calculate_shares_lost` is responsible for computing how many vault shares a pending withdrawal requester forfeits when canceling after the vault equity has increased above the frozen request value. The function computes the retained share amount against the post-removal pool and rejects cases where that retained amount floors to zero. For a requester whose pending withdrawal covers the entire vault share supply, the post-removal share total is zero; when vault equity has increased above the request value, the residual balance is positive, so the retained-share calculation can produce zero before the caller has a chance to apply the sole-owner exception.

`cancel_withdraw_request` already contains the intended escape hatch for this exact case: it snapshots the depositor and total share counts, computes lost shares, then only deducts those shares when the depositor did not own the entire vault before cancellation. Because `calculate_shares_lost` is invoked before `user_owns_entire_vault` is computed and checked, the zero-share rejection can abort the cancellation before the no-deduction branch is reachable.

The share conversion helpers bootstrap path mints shares one-to-one only when the vault balance argument is zero. This is why the full-supply case deterministically reaches the zero-share guard when equity has risen. In the cancel calculation, the full-owner case passes a zero remaining share supply, while the enclosing profit branch requires the current amount to exceed the frozen request value, leaving a positive residual balance. That combination uses the proportional branch with a zero share total, yielding zero retained shares instead of entering the bootstrap path.

Once cancellation is unavailable, the normal withdraw path in `vault_depositor` pays the lesser of the current value of the requested shares and the frozen withdrawal request value, then burns the requested shares and reduces vault totals. For a depositor whose request represented the entire user share supply, that can leave `total_shares` at zero while equity remains in the vault. `Vault::apply_rebase` then repopulates `total_shares` from nonzero equity when the total share count is zero; given the `get_manager_shares` accounting, residual shares not assigned to users or protocol appear as manager ownership.

Remediation

Move the zero-retained-share rejection out of `WithdrawRequest::calculate_shares_lost` and into each cancel caller after the sole-owner condition is known, so it is enforced only when the requester does not own the entire vault share supply. Alternatively, pass the sole-owner condition into `calculate_shares_lost` and return zero lost shares for that condition instead of erroring. Apply the same ordering to `cancel_withdraw_request`, `manager_cancel_withdraw_request`, and `protocol_cancel_withdraw_request`.

Patch

Resolved in PR#339.

Lending-Yield Carveouts Floor To Zero Under Frequent Permissionless Interest Cranking

ID	Severity	Status
OS-VEP-ADV-138	Medium	Resolved

Description

`update_spot_market_cumulative_interest` splits each accrued deposit-interest increment into insurance-fund, protocol-fee, and lender portions during that single accrual call. The insurance-fund and protocol portions are each computed by multiplying the per-call `deposit_interest` by their configured factors and dividing by `IF_FACTOR_PRECISION`, while lenders receive the remaining deposit interest. Because this division truncates and the function does not carry any discarded remainder forward, any fractional carveout that rounds to zero is instead left in the lender remainder. In the same branch, the function advances `last_interest_ts` after increasing `cumulative_deposit_interest`, so a short interval whose lender portion is positive but whose carveouts round to zero is consumed permanently.

`calculate_accumulated_interest` bases the next deposit-interest increment on the elapsed time since `last_interest_ts` and returns zero only when the current timestamp is not greater than the last recorded timestamp. This makes one-second-separated accruals eligible, and shortening the interval lowers the per-call `deposit_interest` before the fee-factor division is applied. As a result, repeated small accruals can cause carveout products to fall below the precision denominator more often than a single longer accrual over the same elapsed time.

The standalone `UpdateSpotMarketCumulativeInterest` account context contains the state, mutable spot market, oracle, and vault accounts, but no signer account. Its handler is reached through the public instruction and is guarded by spot-market validity, exchange-not-paused, and oracle validation checks, not by an authorization signer. Therefore, any caller who can provide the required valid accounts can trigger the accrual path, subject to those constraints.

programs/velocity/src/instructions/keeper.rs

```
3822 pub struct UpdateSpotMarketCumulativeInterest<'info> {
3823     pub state: AccountLoader<'info, State>,
3824     #[account(mut)]
3825     pub spot_market: AccountLoader<'info, SpotMarket>,
3826     /// CHECK: checked in `update_spot_market_cumulative_interest` ix constraint
3827     pub oracle: UncheckedAccount<'info>,
3828     #[account(
3829         seeds = [b"spot_market_vault".as_ref(),
              spot_market.load()?.market_index.to_le_bytes().as_ref()],
3830         bump,
3831     )]
3832     pub spot_market_vault: Box<InterfaceAccount<'info, TokenAccount>>,
3833 }
```

Permissionless cumulative-interest account context

The precision constants used by this path make the truncation boundary explicit: cumulative spot interest is tracked in `SPOT_CUMULATIVE_INTEREST_PRECISION`, while fee factors use `IF_FACTOR_PRECISION`, which aliases the percentage precision. When the per-call deposit-interest gain is small relative to that fee precision, the integer carveout calculation can produce zero even though the same total elapsed time accrued in a larger interval would have produced nonzero insurance-fund and protocol-fee amounts.

As a result, lending yield intended for the staker-owned insurance-fund revenue pool, and similarly for the protocol-fee pool can be reassigned to depositors through `cumulative_deposit_interest` whenever accruals are cranked frequently enough. Since the timestamp advances for the consumed interval and no remainder is stored, the lost carveout is not recovered by later calls. The same shared accrual function is also invoked from many other instruction and controller paths in the repository, so frequent market activity can exercise the same rounding behavior even without a dedicated caller.

Remediation

Track the truncated insurance-fund and protocol-fee remainders per spot market and include them in the next accrual before dividing by `IF_FACTOR_PRECISION`, or compute the lender share first from the combined non-carveout factor and assign the residual interest to the carveout pools so total rounding loss does not systematically favor lenders. Alternatively, only advance `last_interest_ts` once accrued deposit interest is large enough for configured nonzero carveout factors to produce at least one unit.

Patch

Resolved in PR#395.

Deleting A Sub-Account Permanently Orphans Its Fee-Bearing Builder Rows

ID	Severity	Status
OS-VEP-ADV-139	Medium	Resolved

Description

Builder-coded perp orders are tracked in the taker authority's `RevenueShareEscrow`, while each escrow row records the specific `sub_account_id` and `order_id` it belongs to. The reconciliation path in `revoke_completed_orders` only considers non-referral rows whose stored `sub_account_id` matches the live `User` account supplied to the function; if the row belongs to a different sub-account, it is skipped before order status is checked. In `revoke_completed_orders`, a fee-bearing row is only marked `Completed` after this sub-account match succeeds and the corresponding user order is no longer open.

The matching `sub_account_id` is not reusable under the account allocation logic. `handle_initialize_user` requires the requested sub-account id to equal `UserStats.number_of_sub_accounts_created`, then increments that creation counter. The delete handlers read in the repository decrement `number_of_sub_accounts` and the global live-account count, but do not decrement `number_of_sub_accounts_created`, so a deleted sub-account id is retired rather than made available for a later `User` account.

programs/velocity/src/instructions/user.rs

```
pub fn handle_initialize_user<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    validate!(
        sub_account_id == user_stats.number_of_sub_accounts_created,
        ErrorCode::InvalidUserSubAccountId,
        "Invalid sub account id {}, must be {}",
        sub_account_id,
        user_stats.number_of_sub_accounts_created
    )?;
    user_stats.number_of_sub_accounts_created =
        user_stats.number_of_sub_accounts_created.safe_add(1)?;

    let mut state = ctx.accounts.state.load_mut()?;
    [...]
}
```

`handle_initialize_user` allocates sub-account ids from a monotonic creation counter.

The deletion validator does not inspect the authority's `RevenueShareEscrow` or require outstanding builder revenue-share rows for the sub-account to be reconciled first. In `validate_user_deletion`, a user whose positions and orders are otherwise closed can pass deletion even if an unreconciled fee-bearing builder row remains in the authority-scoped escrow.

Once the `User` account for that sub-account is gone, the skipped row may never reach the completed state required by the sweep path. `sweep_completed_revenue_share_for_market` pays builder rows only when they are completed, belong to the target perp market, and have positive accrued fees. A stranded row that cannot be marked `Completed` therefore leaves the builder's earned fee unpayable even though the fee accounting remains in the market's PnL pool.

The accounting impact extends beyond the individual builder payout because fee sweeps reserve `pending_revenue_share` from the market's available PnL pool. In `sweep_market_fees`, `pending_revenue_share` is included in `reserved_claims`, so an orphaned row whose pending amount is never discharged can continue reducing the amount treated as available for future market fee sweeps. This inflates the market's outstanding revenue-share liability for the lifetime of the affected market.

programs/velocity/src/controller/perp_pools.rs

```
pub fn sweep_market_fees([...]) -> VelocityResult<(u128, u128, u128)> {
    [...]
    //     the tranche the #245 floor promises.
    //   * `pending_revenue_share`: builder/referrer fees already accrued and
    //     owed out of this pnl pool by `sweep_completed_revenue_share_for_market`
    //     — draining protocol fees ahead of them would leave those claims
    //     temporarily unpayable.
    // The buffer is an ADDITIONAL retention margin on top that only the IF and
    // AMM-provision drains respect — the protocol drain is exempt and goes
    // first (its per-settle cadence keeps each drain small, and unlike the
    // other two its value is not recoverable later anyway).
    let reserved_claims: u128 = net_user_pnl
        .max(0)
        .cast::<u128>()?
        .safe_add(market.get_bankruptcy_if_tranche_reservation(force)?)?
        .safe_add(market.pending_revenue_share.cast::<u128>()?)?;
    let mut available_unbuffered: u128 =
    pnl_pool_tokens.saturating_sub(reserved_claims);
    [...]
}
```

`sweep_market_fees` reserves pending revenue-share claims.

The practical result is that deleting a sub-account can permanently strand a non-referral, fee-bearing builder row associated with that sub-account. The value is not destroyed, but the builder cannot receive the accrued fee through the normal completed-order sweep, and the market's pending revenue-share accounting can remain inflated because no later user account can carry the retired `sub_account_id` needed by the existing reconciliation function.

Remediation

Thread the authority's optional `RevenueShareEscrow` into the delete and force-delete validation paths, and reject deletion while it contains any non-referral row for the deleting `sub_account_id`

that is still open, not completed, and has `fees_accrued > 0`. Alternatively, add a permissionless reconciliation path that can mark eligible rows completed using the authority and retired `sub_account_id` without requiring the `User` account to still exist, so already stranded rows can be recovered. Keep `number_of_sub_accounts_created` monotonic to avoid aliasing old escrow rows to newly created sub-accounts.

Patch

Resolved in PR#363.

Third-Party Escrow Initialization Permanently Suppresses Referrals

ID	Severity	Status
OS-VEP-ADV-140	Medium	Resolved

Description

`initialize_user_stats` creates `UserStats` with a default referrer. The referrer is assigned later by `initialize_user`, and only while `number_of_sub_accounts_created` is zero. Independently, `initialize_revenue_share_escrow` copies the current `UserStats.referrer` into the new escrow, making initialization order security-sensitive.

The escrow initializer requires `user_stats` to belong to the supplied authority, but the authority is an `UncheckedAccount` (only the payer must sign). Therefore, after an authority's `UserStats` exists but before its first user is initialized, a third-party payer can initialize that authority's escrow PDA. The instruction snapshots `Pubkey::default()` into the escrow, and the later `initialize_user` call updates only `UserStats.referrer`, leaving the escrow stale.

Referral processing treats the escrow snapshot as authoritative. `has_referrer` returns false for the default key, causing `find_or_create_referral_index` to return `None`. The fill paths consequently disable both the referrer reward and the referred user's fee discount. The resize instruction changes only the order-slot vector, and no production instruction updates or resynchronizes `RevenueShareEscrow.referrer`. Thus, subsequent eligible fills for the affected user permanently omit referral benefits, depriving the referrer of the associated revenue share and the user of the corresponding discounts.

Remediation

Treat `UserStats.referrer` as the authoritative value and, before referral eligibility is evaluated, copy it into `RevenueShareEscrow.referrer` whenever the escrow snapshot is still `Pubkey::default()` and `UserStats.referrer` is non-default. Perform this synchronization before calling `has_referrer` or `find_or_create_referral_index`, so existing griefed escrows are repaired on their next eligible use without changing account layouts or requiring authority-signed reinitialization.

Patch

Resolved in PR#362.

Equity-Floor Gates Count Margin-Invalid Assets At Raw Oracle Value

ID	Severity	Status
OS-VEP-ADV-141	Medium	Resolved

Description

`calculate_user_equity` separately accumulates signed net equity and records whether every spot, perpetual, and quote oracle is valid for `MarginCalc`. For a spot position, however, it adds the token value calculated from the live oracle price regardless of that validity result. Consequently, a positive deposit with a stale, uncertain, or excessively volatile oracle can increase the returned equity even though the function marks the valuation as untrustworthy.

programs/velocity/src/math/margin.rs

```
pub fn calculate_user_equity([...]) -> VelocityResult<(i128, bool)> {
    let mut net_usd_value: i128 = 0;
    let mut all_oracles_valid = true;

    for spot_position in user.spot_positions.iter() {
        if spot_position.is_available() {
            continue;
        }

        let spot_market = spot_market_map.get_ref(&spot_position.market_index)?;
        let (oracle_price_data, oracle_validity) =
        oracle_map.get_price_data_and_validity([...])?;
        all_oracles_valid &=
            is_oracle_valid_for_action(oracle_validity,
            Some(VelocityAction::MarginCalc))?;

        let token_amount = spot_position.get_signed_token_amount(&spot_market)?;
        let oracle_price = oracle_price_data.price;
        let token_value = get_token_value(token_amount, spot_market.decimals,
        oracle_price)?;

        net_usd_value = net_usd_value.safe_add(token_value)?;
    }
    [...]
}
```

`calculate_user_equity` records oracle validity separately from raw valuation

`calculate_net_equity_for_floor` invokes that calculation only for accounts with a positive equity floor, but discards `all_oracles_valid` and returns the numeric result alone. This removes the only indication that the total contains a margin-invalid valuation, leaving downstream floor consumers unable to distinguish reliable equity from equity inflated by an invalid positive asset.

programs/velocity/src/math/margin.rs

```rust
pub fn calculate_net_equity_for_floor(
    user: &User,
    perp_market_map: &PerpMarketMap,
    spot_market_map: &SpotMarketMap,
    oracle_map: &mut OracleMap,
) -> VelocityResult<Option<i128>> {
    if user.equity_floor == 0 {
        return Ok(None);
    }

    let (net_equity, _) =
        calculate_user_equity(user, perp_market_map, spot_market_map, oracle_map)?;

    Ok(Some(net_equity))
}
```

The equity-floor helper discards the aggregate validity flag

`meets_withdraw_margin_requirement` demonstrates the inconsistent treatment of invalid prices. Its ordinary margin calculation is configured to ignore invalid deposit oracles and requires valid liability oracles when liabilities or a margin requirement exist, but its subsequent buffered-floor check uses the validity-blind helper. An account can therefore satisfy ordinary initial margin while using the raw value of collateral excluded from that margin calculation to satisfy the supplemental equity floor.

`handle_withdraw` updates the user's spot balance before calling `meets_withdraw_margin_requirement` and transfers tokens from the program vault only after the checks succeed. A user can deposit an asset whose authentic oracle becomes invalid for margin, retain its raw contribution to floor equity, and withdraw other valid collateral so long as ordinary initial margin still passes. The released value is bounded by the account's ordinary free collateral, the raw-value overstatement, and available vault liquidity.

`handle_trip_equity_floor_breaker` calls `calculate_user_equity` directly and, in the checked revision, explicitly rejects the operation when `all_oracles_valid` is false before comparing equity with the floor. Thus, the breaker cannot arm while the user-induced invalid oracle condition persists, leaving the withdrawal and other action-specific floor gates as the relevant protection during that period.

programs/velocity/src/instructions/keeper.rs

```rust
pub fn handle_trip_equity_floor_breaker<'c: 'info, 'info>(
    ctx: Context<'info, TripEquityFloorBreaker<'info>>,
) -> Result<()> {
    [...]
    let (net_equity, all_oracles_valid) =
```

```
        calculate_user_equity(&user, &perp_market_map, &spot_market_map, &mut
        oracle_map)?;
    // An authority-wide freeze must not arm off an invalid price. The floor
    // gates on withdrawals/fills still hold independently of the breaker.
    validate!(
        all_oracles_valid,
        ErrorCode::InvalidOracle,
        "cannot trip equity floor breaker with an invalid oracle"
    )?;
    validate!(
        user.is_below_equity_floor(net_equity),
        ErrorCode::SufficientCollateral,
        "user net equity {} not below equity floor {}",
        net_equity,
        user.equity_floor
    )?;
    [...]
    user_stats.set_equity_breaker_tripped(true);
    Ok(())
}
```

The breaker refuses to arm while any included oracle is invalid

`force_cancel_orders` also determines whether the user is below the floor using `calculate_net_equity_for_floor` in the surrounding logic. The validity-blind result can therefore affect whether permissionless cancellation is admitted. When cancellation proceeds, the function accumulates cancellation fees and pays the filler as shown. More broadly, valid collateral may leave while reliable equity is below the configured floor, and a later downward oracle refresh may leave less recovery value for counterparties, the insurance fund, and depositors. This is not a high-risk issue because the withdrawal must still satisfy ordinary initial margin and does not, by itself, establish immediate bad debt.

programs/velocity/src/controller/orders.rs

```
pub fn force_cancel_orders([...]) -> VelocityResult {
    [...]
    for order_index in 0..user.orders.len() {
        [...]
        total_fee = total_fee.safe_add(fee)?;
        cancel_order([...])?;
    }
    pay_keeper_flat_reward_for_spot(
        user,
        Some(filler),
        spot_market_map.get_quote_spot_market_mut()?.deref_mut(),
        total_fee,
        slot,
```

```
    )?;

    [...]
}
```

Force-cancel pays the filler after eligible orders are cancelled

Remediation

When `equity_floor` is positive, preserve the validity result from `calculate_user_equity` and make `calculate_net_equity_for_floor` reject the calculation with `InvalidOracle` unless `all_oracles_valid` is true. Apply the same requirement to the delegate transfer source and recipient checks that call `calculate_user_equity` directly. If floor checks must remain available during invalid-oracle periods, instead exclude invalid positive assets and positive PnL while retaining liabilities and negative PnL at conservative risk-side values. Do not substitute the ordinary `total_collateral` numerator because it has different weighting semantics. Add fresh, stale, uncertain, and excessive-volatility tests for withdrawals, both transfer sides, order placement and fills, triggers, breaker trips, and force-cancel behavior.

Patch

Resolved in PR#415.

Same-Market Transfer Normalizes Its Liability TWAP Before Margin Validation

ID	Severity	Status
OS-VEP-ADV-142	Medium	Resolved

Description

Both `handle_transfer_deposit` and `handle_transfer_deposit_by_delegate` move one spot market between same-authority subaccounts through the shared `transfer_spot_deposit` helper. Consequently, the authority and delegated transfer paths share the same ordering defect where the transferred market's oracle history is refreshed before the source subaccount is evaluated.

At the beginning of `transfer_spot_deposit`, the helper calls `update_spot_market_cumulative_interest` with the live oracle sample. It subsequently debits `from_user` in the borrow direction and calls `meets_withdraw_margin_requirement`. The source margin check therefore observes oracle-history state already modified by the sample whose validity and liability value it is intended to assess.

The refresh reaches `update_spot_market_twap_stats`, which sanitizes the live sample against the stored one-hour TWAP, calculates new one-hour and five-minute TWAPs, and writes both values to `historical_oracle_data`. This moves the reference values toward a depressed sample before the transfer's source-side admission check.

programs/velocity/src/controller/spot_balance.rs

```
pub fn update_spot_market_twap_stats([...]) -> VelocityResult {
    [...]
    if let Some(oracle_price_data) = oracle_price_data {
        [...]
        let oracle_price_twap = calculate_new_twap(
            capped_oracle_update_price,
            now,
            spot_market.historical_oracle_data.last_oracle_price_twap,
            spot_market.historical_oracle_data.last_oracle_price_twap_ts,
            ONE_HOUR,
        )?;

        let oracle_price_twap_5min = calculate_new_twap(
            capped_oracle_update_price,
            now,
            spot_market
                .historical_oracle_data
                .last_oracle_price_twap_5min,
            spot_market.historical_oracle_data.last_oracle_price_twap_ts,
            FIVE_MINUTE as i64,
        )?;

```

```
            spot_market.historical_oracle_data.last_oracle_price = oracle_price_data.price;
            spot_market.historical_oracle_data.last_oracle_conf =
            oracle_price_data.confidence;
            spot_market.historical_oracle_data.last_oracle_delay = oracle_price_data.delay;
            [...]
        }
        [...]
    }
```

The refresh updates both stored oracle TWAPs

The margin calculation passes the stored one-hour TWAP to `get_price_data_and_validity` and constructs the strict oracle price using the stored five-minute TWAP. Thus, the same refresh can weaken the volatility comparison and reduce the strict value assigned to the newly created liability. After the source check succeeds, `transfer_spot_deposit` credits the destination subaccount with a deposit and completes only recipient-side market and deposit-cap validations. The admitted transfer can therefore place collateral in a sibling subaccount where it is withdrawable while leaving the source with debt valued using the refreshed TWAP. If the source cannot cover that debt, the resulting shortfall may ultimately be socialized to depositors.

Remediation

Snapshot both `last_oracle_price_twap` and `last_oracle_price_twap_5min` before calling `update_spot_market_cumulative_interest`. Preserve the refreshed values separately, restore the pre-refresh snapshots while debiting and performing all source-side margin and spot-margin validations, and write the refreshed values back only after source admission succeeds.

Patch

Resolved in PR#421.

Token-Unit Share Transfer Inflates The Recipient's Profit Basis

ID	Severity	Status
OS-VEP-ADV-143	Medium	Resolved

Description

For `WithdrawUnit::Token`, `get_withdraw_value_and_shares` preserves the caller-supplied token amount as `withdraw_value` while separately converting that amount into an integer number of Vault shares. Because the proportional conversion truncates fractional shares, converting the resulting `n_shares` back into its current Vault amount may produce a value lower than the original token request.

programs/vaults/src/state/withdraw_unit.rs

```rust
pub fn get_withdraw_value_and_shares(
    &self,
    withdraw_amount: u64,
    vault_equity: u64,
    shares: u128,
    total_shares: u128,
    rebase_divisor: Option<u128>,
) -> VaultResult<(u64, u128)> {
    match self {
        WithdrawUnit::Token => {
            let withdraw_value = withdraw_amount;
            let n_shares: u128 =
                vault_amount_to_depositor_shares(withdraw_value, total_shares,
                vault_equity)?;
            Ok((withdraw_value, n_shares))
        }
        [...]
    }
}
```

`WithdrawUnit::Token` preserves the raw amount while deriving integer shares.

`transfer_shares` applies outstanding fees and profit share before performing this conversion, then transfers only `n_shares` between the depositors. However, it subtracts and credits the unadjusted `withdraw_value` through their respective `net_deposits`. Consequently, the recipient receives a cost basis equal to the raw token request even when the shares actually transferred have a lower canonical current value, causing the share ledger and account-local basis ledger to diverge by the rounding discrepancy.

This discrepancy affects performance-fee accounting because `calculate_profit_share_and_update` computes a depositor's profit by subtracting `net_deposits` and previously recognized profit from the current value of that depositor's shares. An overstated recipient basis therefore reduces the profit recognized on that account and can shelter subse-

quent gains from the configured manager and protocol profit shares until the inflated basis is exceeded.

The public `transfer_vault_depositor_shares` instruction passes the caller-selected `amount` and `withdraw_unit` directly to `transfer_shares`. Its surrounding account validation requires authority over the source depositor and requires the destination depositor to belong to the same Vault, but the displayed invariant checks only that the combined share count is unchanged. It does not ensure that the basis moved between the accounts equals the value of the shares moved, allowing an ordinary depositor to select token amounts near share boundaries and repeat the discrepancy across transfers.

programs/vaults/src/instructions/transfer_vault_depositor_shares.rs

```rust
pub fn transfer_vault_depositor_shares<'info>([...]) -> Result<()> {
    [...]
    let total_shares_before = vault_depositor
        .get_vault_shares()
        .safe_add(to_vault_depositor.get_vault_shares())?;

    vault_depositor.transfer_shares(
        &mut *to_vault_depositor,
        &mut vault,
        &mut vp,
        &mut None,
        amount,
        withdraw_unit,
        vault_equity,
        clock.unix_timestamp,
        oracle.price,
    )?;

    let total_shares_after = vault_depositor
        .get_vault_shares()
        .safe_add(to_vault_depositor.get_vault_shares())?;

    validate!(
        total_shares_after.eq(&total_shares_before),
        ErrorCode::InvalidVaultSharesDetected,
        "Total vault depositor shares before != after"
    )?;
    Ok(())
}
```

The transfer instruction forwards the caller-selected amount and unit.

The `tokenize_shares` instruction reaches the same `transfer_shares` helper with its caller-selected amount and unit. It then mints against the returned `shares_transferred` and validates the actual share totals, while the recipient's `net_deposits` has already been credited using the raw token request. Token-denominated tokenization therefore inherits the same basis mismatch and may reduce manager and protocol performance fees without changing the number or redeemable value of the transferred Vault shares.

Remediation

After deriving and validating `n_shares` in `transfer_shares`, convert those shares back into their canonical current Vault amount using `depositor_shares_to_vault_amount(n_shares, vault.total_shares, vault_equity)`. Subtract and credit that canonical amount, rather than the caller-supplied `withdraw_value`, through both depositors' `net_deposits`.

Patch

Resolved in PR#367.

Invalid-Oracle Dust Suppresses The Authority-Wide Equity Breaker

ID	Severity	Status
OS-VEP-ADV-144	Medium	Resolved

Description

A maker delegate can keep the authority-wide equity breaker unset by depositing a negligible amount of invalid-oracle collateral into the subaccount expected to breach its equity floor. The `Deposit` account context permits a signer controlling the source token account to deposit into the selected user, and `handle_deposit` reads the raw oracle price, allocates or retrieves the spot-position slot, and credits the deposit without requiring that oracle to be valid for margin calculations. Consequently, a nonzero deposit may create a persistent spot position even while the oracle is stale, uncertain, or excessively volatile.

The resulting position has a disproportionate effect on `calculate_user_equity`. As shown in the function, every non-available spot position contributes its oracle validity to the cumulative `all_oracles_valid` predicate before its token value is added to equity. There is no minimum-value exception, so economically negligible positive collateral can make the predicate false even though it has no material bearing on whether the subaccount breached its floor.

programs/velocity/src/math/margin.rs

```
pub fn calculate_user_equity([...]) -> VelocityResult<(i128, bool)> {
    [...]
    for spot_position in user.spot_positions.iter() {
        if spot_position.is_available() {
            continue;
        }
        let spot_market = spot_market_map.get_ref(&spot_position.market_index)?;
        let (oracle_price_data, oracle_validity) =
        oracle_map.get_price_data_and_validity(
            MarketType::Spot,
            spot_market.market_index,
            &spot_market.oracle_id(),
            spot_market.historical_oracle_data.last_oracle_price_twap,
            spot_market.get_max_confidence_interval_multiplier()?,
            -1,
            0,
            Some(LogMode::Margin),
        )?;
        all_oracles_valid &=
            is_oracle_valid_for_action(oracle_validity,
            Some(VelocityAction::MarginCalc))?;
        let token_amount = spot_position.get_signed_token_amount(&spot_market)?;
        let oracle_price = oracle_price_data.price;
        let token_value = get_token_value(token_amount, spot_market.decimals,
        oracle_price)?;
```

```
            net_usd_value = net_usd_value.safe_add(token_value)?;
        }
        [...]
    }
```

Every non-available spot position contributes to the aggregate oracle-validity predicate.

`handle_trip_equity_floor_breaker` treats that cumulative predicate as an all-or-nothing prerequisite. It returns `InvalidOracle` before comparing net equity with the configured floor and before setting the breaker in the authority's `UserStats`. Thus, while the dust position's oracle remains invalid, repeated attempts to prove the material breach cannot activate the authority-wide freeze. Risk-increasing order placement does not independently recover the missing cross-subaccount protection. `meets_place_order_margin_requirement` checks the acting `User` account's margin requirement and buffered equity floor, so an otherwise healthy sibling can continue placing orders while it remains locally compliant. The local equity helper also discards the validity boolean returned by `calculate_user_equity`.

The maker-fill path similarly relies on the authority's breaker bit together with checks calculated from the supplied maker subaccount. A risk-increasing fill is rejected when the selected `UserStats` reports a tripped breaker or when that maker is below its own buffered floor. Because the invalid dust prevents the breaker bit from being set, a healthy sibling can pass these checks using its remaining local margin and equity headroom. `handle_place_and_make_perp_order` makes this path available to a delegate through the surrounding `PlaceAndMake` authorization constraint, which accepts either the user's authority or its configured delegate. The handler places an immediate-or-cancel, post-only limit order for the maker and then fills it against the supplied taker and taker order, allowing the delegate to choose an attacker-controlled counterparty while the authority-wide breaker remains unset.

The attacker-controlled counterparty may therefore receive positive realized PnL while the Velocity-funded sibling absorbs the corresponding mark loss and exposure. Ordinary margin requirements and the sibling's buffered floor continue to bound the trade, but they do not provide the authority-wide freeze intended to apply after any sibling breaches. The delegate can consume that remaining headroom throughout the invalid-oracle window and can preserve the suppression by keeping the dust position open.

Remediation

Prevent an economically negligible invalid-oracle deposit from vetoing the authority-wide transition. At minimum, when a deposit would create a new non-available spot position for a user with a nonzero equity floor, require the market oracle to be valid for the same margin action used by the breaker before crediting the position, including deposits initiated by delegates or external signers. The breaker should also avoid an undifferentiated all-oracles boolean.

Patch

Resolved in PR#380.

Pooled Tokenized Cost Basis Lets A Later Tokenizer Strip Existing Holders' Loss Shelter

ID	Severity	Status
OS-VEP-ADV-145	Medium	Resolved

Description

A `TokenizedVaultDepositor` represents all holders of its fungible token mint through one account-level `vault_shares`, `net_deposits`, and `cumulative_profit_share_amount` balance. Consequently, profit-share accounting has no holder-specific cost basis. When the pooled position is under water, an ordinary depositor can add shares to this shared account and acquire tokens whose economics include part of the loss shelter accumulated by existing holders, as reflected by the shared fields shown in `TokenizedVaultDepositor`.

The loss shelter persists because `calculate_profit_share_and_update` compares the pool's current value against its aggregate deposits and cumulative profit, but updates the checkpoint only when profit exceeds the hurdle. An under-water calculation returns zero without reducing `cumulative_profit_share_amount` or otherwise crystallizing the loss. The excess of aggregate basis over current value therefore remains available to offset a subsequent recovery.

During tokenization, `transfer_shares` first applies profit share to both the source depositor and the destination `TokenizedVaultDepositor`. It then values the transferred shares at current vault equity, moves them into the destination, and adds that current value to the destination's pooled `net_deposits`. The newcomer is issued tokens against the current pooled backing while the earlier loss shelter remains shared across the enlarged token supply, diluting the shelter attributable to incumbent holders.

When the pooled position later recovers sufficiently, `apply_profit_share` converts the calculated fee into vault shares, removes those shares from the `TokenizedVaultDepositor`, and reduces `vault.user_shares`. The manager and protocol fee accounting is still credited in full. The defect changes how that fee burden is allocated among token holders rather than causing the configured recipients to be underpaid.

Redemption crystallizes the tokenized depositor's pooled management and profit-share fees before converting the redeeming holder's tokens into shares and transferring those shares to that holder's `VaultDepositor`. After the transfer, `redeem_tokens` resets `last_vault_shares` to the tokenized depositor's remaining balance. This post-redemption checkpoint permits later tokenization into the same pool, making the dilution sequence repeatable across redemption and tokenization cycles.

programs/vaults/src/instructions/redeem_tokens.rs

```
pub fn redeem_tokens<'info>([...]) -> Result<()> {
    [...]
    let total_supply_before = ctx.accounts.mint.supply;
    let spot_market = spot_market_map.get_ref(&spot_market_index)?;
```

```
    let oracle = oracle_map.get_price_data(&spot_market.oracle_id())?;
    // redeem_tokens crystallizes the management fee and the tokenized depositor's
    profit share.
    let (shares_to_transfer, mut vp) =
    tokenized_vault_depositor.redeem_tokens([...])?;
    [...]
    let manager_shares_before = vault.get_manager_shares(&mut vp)?;
    let protocol_shares_before = vault.get_protocol_shares(&mut vp);
    let total_shares_before = vault_depositor
        .get_vault_shares()
        .safe_add(tokenized_vault_depositor.get_vault_shares())?
        .safe_add(manager_shares_before)?
        .safe_add(protocol_shares_before)?;

    let (shares_transferred, mut vp) =
    tokenized_vault_depositor.transfer_shares([...])?;
    [...]
    tokenized_vault_depositor.checkpoint_vault_shares();
    [...]
}
```

Redemption crystallization and post-transfer checkpointing

The `tokenize_shares` instruction requires the signer to control the source `VaultDepositor` and token account, but does not require authorization from the manager or existing token holders. It transfers the caller's own vault shares into the shared `TokenizedVaultDepositor` and mints pooled tokens in return. Existing holders therefore need not participate in, authorize, or receive advance notice of the operation that dilutes their accumulated shelter.

The resulting fee-burden transfer can irreversibly reduce incumbent holders' backing once profit-share shares are removed from the pool. On recovery, incumbents may bear part of the profit-share charge that the newcomer would have owed under standalone basis accounting, while the newcomer correspondingly avoids that amount. The dilution is caused entirely by the newcomer's permitted tokenization call.

Remediation

Attribute performance-fee basis to holders or token units instead of maintaining a single basis for the entire `TokenizedVaultDepositor`. When minting, record the newcomer's entry basis separately and calculate the minted token amount so it cannot acquire any pre-existing difference between the pool's basis and current value. When burning, remove the basis associated with the burned tokens, for example through a basis-per-token accumulator or separately accounted tranches.

Patch

Resolved in PR#372.

Unpayable Positive-PnL Dust Can Veto Cross-Market Bankruptcy Resolution

ID	Severity	Status
OS-VEP-ADV-146	Medium	Resolved

Description

The cross-margin bankruptcy predicate treats every non-isolated perpetual position with a positive `quote_asset_amount` as evidence that the account is not bankrupt. In `is_cross_margin_bankrupt`, this decision does not consider the amount of positive PnL, the market responsible for paying it, or the liquidity available in that market's PnL pool. Consequently, a single positive quote unit can override a substantially larger negative quote balance in another market.

That positive balance may not be immediately realizable. In `settle_pnl`, the claimable amount and the subsequent pool-balance update are constrained by the paying market's available PnL-pool tokens. If the computed settlement is zero, settlement returns without reducing the user's positive claim. An empty PnL pool may therefore leave the balance recorded on the user while providing no funds with which to settle it.

The unresolved positive balance prevents the cross-margin account from entering the bankruptcy state required by `resolve_perp_bankruptcy`. As shown in the resolver, it evaluates the applicable bankruptcy predicate and returns `UserNotBankrupt` unless that predicate permits the account to enter bankruptcy. This check occurs before the resolver calculates its insurance payment or socializes any remaining loss.

programs/velocity/src/controller/liquidation.rs

```
pub fn resolve_perp_bankruptcy([...]) -> VelocityResult<u64> {
    let liquidation_mode = get_perp_liquidation_mode(user, market_index)?;
    if !liquidation_mode.is_user_bankrupt(user)?
        && liquidation_mode.should_user_enter_bankruptcy(user)?
    { liquidation_mode.enter_bankruptcy(user)?; }
    validate!(
        liquidation_mode.is_user_bankrupt(user)?,
        ErrorCode::UserNotBankrupt, "user not bankrupt",
    )?;
    [...]
}
```

Perpetual-bankruptcy eligibility check

The spot-bankruptcy path is gated in the same way. `resolve_spot_bankruptcy` calls `is_cross_margin_bankrupt` before entering cross-margin bankruptcy and then requires the user's bankruptcy flag to be set. Thus, the unfunded positive PnL may prevent both cross-margin resolution paths from reaching their respective insurance and loss-allocation logic.

programs/velocity/src/controller/liquidation.rs

```rust
pub fn resolve_spot_bankruptcy([...]) -> VelocityResult<u64> {
    if !user.is_cross_margin_bankrupt() && is_cross_margin_bankrupt(user) {
        user.enter_cross_margin_bankruptcy();
    }
    validate!(
        user.is_cross_margin_bankrupt(),
        ErrorCode::UserNotBankrupt, "user not bankrupt",
    )?;
    [...]
}
```

Spot-bankruptcy eligibility check

The available liquidation paths do not provide a general cross-market setoff for this state. Positive-PnL-for-borrow liquidation requires the user to hold a nonzero spot borrow, while direct perpetual liquidation does not transfer a zero-base position. The `liquidate_perp_pnl_for_deposit` path addresses the inverse exchange—negative perpetual PnL for an existing spot deposit—and validates the relevant spot position, so it cannot consume a zero-base positive claim for an account without spot assets or liabilities.

A zero-base account may therefore retain a small positive claim in an empty-pool market while carrying a much larger negative balance elsewhere. Until the positive claim becomes payable or another eligible position is introduced, the loss market cannot draw its configured insurance coverage or socialize the uncovered deficit through these resolvers. This leaves the deficit unresolved and may defer resolution until a later insurance-fund state, giving the user the timing option described in the finding.

Remediation

Add a public, atomic cross-market estate-setoff operation for bankrupt accounts. The operation should reduce a zero-base positive perpetual quote claim and negative perpetual PnL elsewhere by the same value while explicitly recording the corresponding obligation between the two market ledgers. If the claim market's PnL pool lacks funds, record a receivable owed by that market to the loss market instead of treating the claim as funded or discarding it. Apply the bankruptcy predicate after this conserved setoff so an unpayable positive claim cannot veto resolution of the remaining debt. The issue cannot be fixed by ignoring positive PnL, because the protocol must preserve both the user's claim and the identity of the market that owes it.

Patch

Resolved in PR#422.

Expiry Solver Spends Earned Revenue-Share Backing On Perp Winners

ID	Severity	Status
OS-VEP-ADV-147	Medium	Resolved

Description

When `settle_expired_market` finalizes a perpetual market, it first transfers the available AMM fee budget into the PnL pool and then passes the pool's entire token balance to `calculate_expiry_price`. As shown in `settle_expired_market`, this balance is not reduced by `pending_revenue_share`, even though that amount is already owed to builders and referrers, or by the standing bankruptcy insurance tranche.

programs/velocity/src/vlp/amm/refresh.rs

```rust
pub fn settle_expired_market([...]) -> VelocityResult {
    [...]
    let total_excess_balance: i128 = get_token_amount(
        market.pnl_pool.scaled_balance,
        spot_market,
        market.pnl_pool.balance_type(),
    )?
    .cast()?;

    crate::dlog!(market.market_index);
    crate::dlog!(total_excess_balance);

    let expiry_price = amm::calculate_expiry_price(
        &market.amm,
        target_expiry_price,
        total_excess_balance,
        market.quote_asset_amount,
        // Folded into the cost basis: `settle_expired_position` settles funding
        // into each user's quote before paying them (OtterSec #125).
        market.net_unsettled_funding_pnl,
        market.order_step_size,
    )?;
    [...]
}
```

Gross PnL-pool balance passed into the expiry-price solver

The expiry-price solver treats its `total_excess_balance` argument as the value available to back aggregate user claims. In `calculate_expiry_price`, the balance is incorporated directly into the price calculation, so supplying the gross PnL-pool balance can produce a settlement price that authorizes positive trader PnL against encumbered tokens.

programs/velocity/src/vlp/amm/math/amm.rs

```rust
pub fn calculate_expiry_price(
    amm: &AMM,
    target_price: i64,
    total_excess_balance: i128,
    quote_asset_amount: i128,
    net_unsettled_funding_pnl: i64,
    order_step_size: u64,
) -> VelocityResult<i64> {
    [...]
    let net_user_cost_basis =
        calculate_net_user_cost_basis(quote_asset_amount, net_unsettled_funding_pnl)?;

    // net_user_unrealized_pnl negative = surplus in market
    // net_user_unrealized_pnl positive = expiry price needs to differ from oracle
    let best_expiry_price = -(net_user_cost_basis
        .safe_sub(total_excess_balance.cast::<i128>()?)?
        .safe_mul(PRICE_TIMES_AMM_TO_QUOTE_PRECISION_RATIO_I128)?
        .safe_div(amm.base_asset_amount_with_amm)?)
    .cast::<i64>()?;
    [...]
```

Expiry-price calculation treats the supplied balance as claim backing

This treatment is inconsistent with the market's canonical balance sheet. As shown in `calculate_perp_market_amm_summary_stats`, accrued revenue share is explicitly subtracted when calculating AMM equity because the corresponding tokens remain in the pools but belong to builders or referrers rather than the AMM.

programs/velocity/src/math/perp_market.rs

```rust
pub fn calculate_perp_market_amm_summary_stats(
    perp_market: &PerpMarket,
    spot_market: &SpotMarket,
    perp_market_oracle_price: i64,
) -> VelocityResult<i128> {
    [...]
    let net_user_pnl = calculate_net_user_pnl(
        &perp_market.amm,
        perp_market_oracle_price,
        perp_market.quote_asset_amount,
        perp_market.net_unsettled_funding_pnl,
    )?;

    pnl_tokens_available
        .safe_sub(net_user_pnl)?
```

```
            .safe_sub(perp_market.fee_ledger.pending_protocol_fee.cast()?)?
            .safe_sub(perp_market.fee_ledger.pending_if_fee.cast()?)?
            .safe_sub(perp_market.pending_revenue_share.cast()?)
    }
```

Canonical AMM accounting subtracts pending revenue share

The ordinary fee waterfall applies the same accounting rule before permitting assets to leave the PnL pool. `sweep_market_fees` reserves positive unsettled user PnL, the applicable bankruptcy insurance tranche, and `pending_revenue_share` before calculating the balance available for fee distributions. Revenue-share settlement confirms that the accrued amount is a direct claim on the PnL pool. In `sweep_completed_revenue_share_for_market`, completed builder and referrer rows are paid from that pool, and the sweep stops when the balance remaining above the other reserved claims cannot cover the accrued fee.

The liability is also maintained independently from trader settlement state. As shown by `accrue_pending_revenue_share` and `settle_pending_revenue_share`, the market increments the aggregate when revenue share is earned and reduces it only when the corresponding obligation is discharged.

programs/velocity/src/state/perp_market.rs

```
/// Record builder/referrer revenue share accrued on a fill into the
/// per-market aggregate (mirrors the per-order `fees_accrued` write so the
/// fee sweep can reserve the tokens backing it).
pub fn accrue_pending_revenue_share(&mut self, amount: u64) -> VelocityResult {
    self.pending_revenue_share = self.pending_revenue_share.safe_add(amount)?;
    Ok(())
}

/// Discharge revenue share from the per-market aggregate as
/// `sweep_completed_revenue_share_for_market` pays it out of the pnl pool.
pub fn settle_pending_revenue_share(&mut self, amount: u64) -> VelocityResult {
    self.pending_revenue_share = self.pending_revenue_share.saturating_sub(amount);
    Ok(())
}
```

Per-market revenue-share liability accrual and discharge

Expired-position settlement does not enforce this reservation. In `settle_expired_position`, the committed expiry price determines the position closeout and the resulting PnL is paid through the PnL-pool transfer path. Neither this path nor its pool-availability check reduces `pending_revenue_share` or protects its token backing.

Consequently, positive-PnL traders can consume tokens already committed to builders and referrers while the revenue-share counter remains outstanding. For example, with 400 quote in the PnL pool and a 100-quote revenue-share liability, only 300 quote is unencumbered, yet the

gross-balance solve can commit 399.999900 quote to trader claims. Paying those claims leaves 0.000100 quote against the unchanged 100-quote liability, preventing the affected completed revenue-share row from being swept. Because the expiry price is committed during market wind-down, later repricing cannot restore the reserved backing.

Remediation

Define a single authoritative calculation for unencumbered expiry backing and use it both when committing the expiry price and when validating expired-position payouts. At minimum, subtract `pending_revenue_share` and the standing bankruptcy insurance tranche from the PnL pool before passing backing into `calculate_expiry_price`, using checked arithmetic with a fail-closed zero floor.

Patch

Resolved in PR#345.

Per-User Exception Enables Sybil Breaker Bypass

ID	Severity	Status
OS-VEP-ADV-148	Medium	Resolved

Description

`check_withdraw_limits` derives market-wide minimum deposit and maximum borrow amounts from the market's token balances, TWAPs, utilization, and configured guard threshold. However, when the resulting market state violates these limits, the function does not reject unconditionally. It delegates the decision to `check_user_exception_to_withdraw_limits` for the withdrawing `User`.

`check_user_exception_to_withdraw_limits` grants the bypass using only account-local state. A withdrawal qualifies when that `User` has nonnegative net and cumulative deposits, retains a deposit position, and its reconstructed pre-withdraw position is less than one tenth of `withdraw_guard_threshold`. The function neither records market-wide exception usage nor aggregates balances across other `User` accounts or authorities, making the exemption independently reusable by every qualifying account.

`update_spot_balances_and_cumulative_deposits_with_limits` first applies the requested debit to one user's spot position and the aggregate market balances, then calls `check_withdraw_limits` with that specific `User` and withdrawal amount. Consequently, the exception is evaluated separately for each account after its individual balance update, and an otherwise invalid market-wide withdrawal is accepted whenever that account satisfies the local principal conditions.

`handle_withdraw` increments the selected account's withdrawal totals and invokes the per-position update and limit check before performing the account's margin validations and releasing tokens from the spot-market vault. These other validations remain applicable, but they do not aggregate use of the small-depositor exception. A withdrawal that passes through the per-user exception can therefore proceed to the vault transfer despite breaching the market-wide withdrawal floor.

The ordinary internal-transfer path also operates on individual accounts. As shown in `transfer_spot_deposit`, the source account is debited through the same per-user limit wrapper. The surrounding control flow then credits the destination account and records the transfer without moving tokens into the vault. Same-authority accounts can therefore be used to distribute deposit claims into multiple qualifying positions before submitting their external withdrawals, including claims originating from a margin-backed source position when the applicable margin and breaker checks permit the transfer.

programs/velocity/src/instructions/user.rs

```
fn transfer_spot_deposit([...]) -> anchor_lang::Result<()> {
    [...]
    {
        [...]
        from_user.increment_total_withdraws(
```

```
            amount,
            oracle_price,
            spot_market.get_precision().cast()?,
        )?;
        // prevents withdraw when limits hit
        controller::spot_position::update_spot_balances_and_cumulative_deposits_with_limi
            amount as u128,
            &SpotBalanceType::Borrow,
            spot_market,
            from_user,
        )?;
    }
    from_user.meets_withdraw_margin_requirement(
        perp_market_map,
        spot_market_map,
        oracle_map,
        MarginRequirementType::Initial,
    )?;
    validate_spot_margin_trading(from_user, perp_market_map, spot_market_map,
    oracle_map)?;
    [...]
}
```

Internal transfers debit the source through the per-user check

`handle_initialize_user` permits successive subaccounts by requiring the next sequential identifier, incrementing the account counts, and enforcing only the configured protocol-wide maximum, which may be disabled when set to zero. This creation control does not aggregate small-withdrawal eligibility across an authority's subaccounts, and using additional authorities likewise does not affect the market-local breaker state.

programs/velocity/src/instructions/user.rs

```
pub fn handle_initialize_user<'c: 'info, 'info>([...]) -> Result<()> {
    [...]
    validate!(
        sub_account_id == user_stats.number_of_sub_accounts_created,
        ErrorCode::InvalidUserSubAccountId,
        "Invalid sub account id {}, must be {}",
        sub_account_id,
        user_stats.number_of_sub_accounts_created
    )?;
    user_stats.number_of_sub_accounts_created =
        user_stats.number_of_sub_accounts_created.safe_add(1)?;

    let mut state = ctx.accounts.state.load_mut()?;
    safe_increment!(state.number_of_sub_accounts, 1);
```

```
    let max_number_of_sub_accounts = state.max_number_of_sub_accounts();
    validate!(
        max_number_of_sub_accounts == 0
            || state.number_of_sub_accounts <= max_number_of_sub_accounts,
        ErrorCode::MaxNumberOfUsers
    )?;
    [...]
}
```

User initialization does not aggregate exception eligibility

The relaunch configuration assigns nonzero `withdraw_guard_threshold` values to the illustrated USDT and SOL markets. Because the exception derives its per-user cutoff from one tenth of this configured market value but maintains no corresponding aggregate allowance, increasing the number of qualifying accounts increases the total amount capable of bypassing the floor.

An actor whose unsplit position exceeds the exception threshold can divide it among enough accounts that each reconstructed position remains below the threshold. Once the market-wide TWAP or utilization floor would reject the combined withdrawal, the shards can still qualify independently and collectively remove more immediately available vault liquidity than the breaker permits for one account. This favors a first mover capable of sharding and may leave later depositors unable to withdraw while they remain exposed to borrower repayment, liquidation, depeg, and default risk.

Remediation

Replace the independently reusable per-`User` exception with a market-scoped allowance for the same rolling window as the withdrawal breaker. When an external vault withdrawal fails the normal market-wide check, atomically consume its amount from a counter stored for that `SpotMarket` and reject the withdrawal if the remaining allowance is insufficient, regardless of the `User` or authority submitting it.

Patch

Resolved in PR#379.

Zero-Token Socialized Deposit Residues Veto Follow-On Bankruptcy

ID	Severity	Status
OS-VEP-ADV-149	Medium	Resolved

Description

When a spot-market bankruptcy socializes a loss that meets or exceeds the market's token-valued deposits, `calculate_cumulative_deposit_interest_delta_to_resolve_bankruptcy` caps the reduction at the current cumulative deposit interest minus one. As a result, the cumulative deposit interest remains at one even when depositors have effectively been wiped out.

programs/velocity/src/math/liquidation.rs

```rust
pub fn calculate_cumulative_deposit_interest_delta_to_resolve_bankruptcy(
    borrow: u128,
    spot_market: &SpotMarket,
) -> VelocityResult<u128> {
    let total_deposits = get_token_amount(
        spot_market.deposit_balance,
        spot_market,
        &SpotBalanceType::Deposit,
    )?;
    // No depositors to haircut: nothing to socialize against.
    if total_deposits == 0 {
        return Ok(0);
    }
    let delta = spot_market
        .cumulative_deposit_interest
        .safe_mul(borrow)?
        .safe_div_ceil(total_deposits)?;
    Ok(delta.min(spot_market.cumulative_deposit_interest.saturating_sub(1)))
}
```

Deposit-interest reduction is capped so the index remains at least one

Spot deposits are represented by scaled balances rather than raw token amounts. As shown by `get_token_amount`, a deposit's token value is calculated by multiplying its scaled balance by the cumulative deposit interest and rounding down by the market precision. After the index is reduced to one, a strictly positive scaled balance can therefore represent zero tokens.

programs/velocity/src/math/spot_balance.rs

```rust
pub fn get_token_amount(
    balance: u128,
    spot_market: &SpotMarket,
    balance_type: &SpotBalanceType,
) -> VelocityResult<u128> {
```

```
    let precision_decrease = 10_u128.pow(19_u32.safe_sub(spot_market.decimals)?);
    let cumulative_interest = match balance_type {
        SpotBalanceType::Deposit => spot_market.cumulative_deposit_interest,
        SpotBalanceType::Borrow => spot_market.cumulative_borrow_interest,
    };
    let token_amount = match balance_type {
        SpotBalanceType::Deposit => balance
            .safe_mul(cumulative_interest)?
            .safe_div(precision_decrease)?,
        SpotBalanceType::Borrow => balance
            .safe_mul(cumulative_interest)?
            .safe_div_ceil(precision_decrease)?,
    };
    Ok(token_amount)
```

Deposit token values round down after applying cumulative interest

`resolve_spot_bankruptcy` clears the bankrupt borrow and then applies the calculated reduction to the market-wide cumulative deposit interest. Because this operation changes the global index without iterating over other users' spot positions, affected depositors retain their existing positive scaled balances even when those balances subsequently convert to zero tokens. The cross-margin bankruptcy predicate does not evaluate deposit positions using their current market and token value. Instead, `is_cross_margin_bankrupt` immediately treats any deposit position with a positive `scaled_balance` as an asset and returns false. A zero-token deposit residue can consequently prevent an account with an unrelated spot liability or negative cross-margin perpetual PnL from being admitted into bankruptcy.

programs/velocity/src/math/bankruptcy.rs

```
pub fn is_cross_margin_bankrupt(user: &User) -> bool {
    // user is bankrupt iff they have spot liabilities, no spot assets, and no perp
    exposure
    let mut has_liability = false;
    for spot_position in user.spot_positions.iter() {
        if spot_position.scaled_balance > 0 {
            match spot_position.balance_type {
                SpotBalanceType::Deposit => return false,
                SpotBalanceType::Borrow => has_liability = true,
            }
        }
    }
    [...]
}
```

Bankruptcy admission treats every positive scaled deposit as an asset

`resolve_perp_bankruptcy` relies on the selected liquidation mode's bankruptcy predicate before repairing negative perpetual PnL. If the zero-token deposit residue causes the predicate to return

false, the function does not enter bankruptcy and subsequently rejects the resolution because the user is not marked bankrupt.

programs/velocity/src/controller/liquidation.rs

```rust
pub fn resolve_perp_bankruptcy([...]) -> VelocityResult<u64> {
    let liquidation_mode = get_perp_liquidation_mode(user, market_index)?;
    if !liquidation_mode.is_user_bankrupt(user)?
        && liquidation_mode.should_user_enter_bankruptcy(user)?
    {
        liquidation_mode.enter_bankruptcy(user)?;
    }
    validate!(
        liquidation_mode.is_user_bankrupt(user)?,
        ErrorCode::UserNotBankrupt,
        "user not bankrupt",
    )?;
    [...]
}
```

Perpetual bankruptcy resolution requires the affected predicate to admit the user

The PnL-for-deposit liquidation path cannot consume the residue either. Its cross-margin `get_spot_token_amount` implementation confirms that the row is deposit-typed, converts it to tokens, and rejects the position when the resulting token amount is zero. Thus, the same row that appears to be an asset during bankruptcy admission provides no transferable asset during liquidation.

programs/velocity/src/state/liquidation_mode.rs

```rust
fn get_spot_token_amount(&self, user: &User, spot_market: &SpotMarket) ->
VelocityResult<u128> {
    let spot_position = user.get_spot_position(spot_market.market_index)?;
    validate!(
        spot_position.balance_type == SpotBalanceType::Deposit,
        ErrorCode::WrongSpotBalanceType,
        "User did not have a deposit for the asset market"
    )?;
    let token_amount = spot_position.get_token_amount(spot_market)?;
    validate!(
        token_amount != 0,
        ErrorCode::InvalidSpotPosition,
        "asset token amount zero for market index = {}",
        spot_market.market_index
    )?;
    Ok(token_amount)
}
```

PnL-for-deposit liquidation rejects a deposit that converts to zero tokens

The generic spot-balance updater also does not normalize this state. In `update_spot_balances`, an update opposing the existing balance type reduces the scaled balance only when its current token amount is nonzero. A zero-token update therefore leaves both the user's scaled deposit and the corresponding market aggregate unchanged. Consequently, a depositor whose collateral was wiped by socialization may be unable to resolve later bad debt in another cross-margin market. Bankruptcy repair, insurance-fund use, or further loss socialization for that account can remain delayed until funding is added or sufficient interest accrues for the residue to represent at least one token.

Remediation

Add a market-aware normalization step for deposit positions where `scaled_balance` is positive but `get_token_amount` returns zero. Before bankruptcy admission and PnL-for-deposit liquidation validation, atomically subtract the position's exact scaled residue from `spot_market.deposit_balance`, clear the user's scaled position, and then re-evaluate the relevant bankruptcy or liquidation predicate against the normalized state. Ensure the adjustment updates both ledgers in the same transaction rather than merely ignoring the user row.

Patch

Resolved in PR#422.

Pyth Ownership Check Accepts Invalid Oracles

ID	Severity	Status
OS-VEP-ADV-150	Low	✓ Resolved

Description

`OracleMap::load_one` treats ownership by an allowlisted Pyth program as sufficient to register an account, without verifying that its data represents a valid Pyth price feed. A caller could therefore supply another Pyth-owned account type or a zeroed account reassigned to the Pyth program, and the account would enter the oracle map. Subsequent price decoding may fail unexpectedly, causing affected instructions to revert, or could accept malformed data if downstream parsing is insufficiently strict. Ownership determines which program may modify an account, but does not verify its internal type or initialization state.

programs/velocity/src/state/oracle_map.rs

```rust
pub fn load_one<'c>(
    account_info: &'c AccountInfo<'a>,
    slot: u64,
    oracle_guard_rails: Option<OracleGuardRails>,
) -> VelocityResult<OracleMap<'a>> {
    let mut oracles: BTreeMap<Pubkey, AccountInfo<'a>> = BTreeMap::new();
    if EXTERNAL_ORACLE_PROGRAM_IDS.contains(account_info.owner) {
        let pubkey = account_info.key();
        oracles.insert(pubkey, account_info.clone());
    } else if account_info.owner == &crate::id() {
        let data = account_info.try_borrow_data().map_err(|e| {
            msg!("Failed to borrow data while loading oracle map {:?}", e);
            UnableToLoadOracle
        })?;
        let account_discriminator = &data[..8];
        [...]

    }[...]
}
```

Remediation

Validate the Pyth magic value, version, account-type discriminator, expected data length, and initialization status before insertion. This ensures only structurally and semantically valid Pyth price accounts are accepted as oracle inputs.

Patch

Resolved in PR#440.

5. Suggestions

Here, we present a discussion of the 3 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-VEP-SUG-00	Informational	✗ To-do	Post-Expiry Cranks Influence The Finalization-Time Settlement TWAP
OS-VEP-SUG-01	Informational	✗ To-do	Third-Party Zero-Slot Referral Escrow Suppresses Rewards
OS-VEP-SUG-02	Informational	✓ Resolved	Variable Slot Times Alter Accounting Calculations

Post-Expiry Cranks Influence The Finalization-Time Settlement TWAP

ID	Severity	Status
OS-VEP-SUG-00	Informational	✗ To-do

Description

When `settle_expired_market` finalizes an expired perpetual market, it derives `target_expiry_price` from the market's current historical oracle data, passes that value into `calculate_expiry_price`, then stores the result in `market.expiry_price` and moves the market to `MarketStatus::Settlement`. For non-prelaunch markets, the target input is the current five-minute oracle TWAP rather than a value pinned to the market's `expiry_ts`, as shown in `settle_expired_market`.

The oracle TWAP state used by finalization is mutable through the market statistics refresh path. `update_oracle_twap` recalculates both the funding-period TWAP and the five-minute TWAP using the supplied `now` timestamp and latest normalized oracle price, then writes those values back to `historical_oracle_data`.

`handle_settle_expired_market` refreshes `update_oracle_derived_stats` immediately before calling `settle_expired_market`, and the public `update_amms` keeper-style instruction also reaches the same oracle-derived stats path through an arbitrary signer while the exchange is not paused. The settlement helper only checks that `expiry_ts` is nonzero and not in the future before consuming the current TWAP; it does not require that the TWAP was captured at the expiry timestamp.

As a result, an additional refresh after expiry but before finalization can change the committed `expiry_price` for the market if post-expiry oracle data moves the five-minute TWAP. Expired-position settlement later uses this stored `expiry_price` to value each user's base asset amount.

Remediation

If settlement is intended to price off the market state at expiry, snapshot the oracle TWAP at the expiry timestamp and have finalization consume that snapshot rather than live state. Otherwise document that post-expiry cranks are an accepted input to finalization.

Third-Party Zero-Slot Referral Escrow Suppresses Rewards

ID	Severity	Status
OS-VEP-SUG-01	Informational	✗ To-do

Description

Referral reward accounting depends on a per-market referral row in the taker's `RevenueShareEscrow`. In `find_or_create_referral_index`, the escrow first returns no referral when it has no referrer, then searches existing rows, and finally attempts to allocate a new referral row for the market. If that allocation fails because the escrow has no available order slot, the function logs the failure and returns `None` rather than surfacing an error.

The order controller uses this return value as the condition for whether a fill should apply referral economics. `can_reward_user_with_referral_reward` returns true only when `find_or_create_referral_index` returns an index, and fee calculation applies the referee discount and referrer reward only when that boolean is true. As a result, a full or zero-slot escrow follows the same fee path as an account without an applicable referral row, causing the referral discount and reward to be omitted without failing the fill.

The escrow can be initialized with a caller-supplied order capacity. `handle_initialize_revenue_share_escrow` assigns the escrow authority from the provided authority account, resizes the order vector to `num_orders`, copies the authority's current referrer from `UserStats`, and validates the escrow. The associated account context initializes the escrow PDA using the authority key while the payer is the signer, so the initializer is not required by this context to be the escrow authority.

programs/velocity/src/instructions/user.rs

```
411 pub fn handle_initialize_revenue_share_escrow<'c: 'info, 'info>(
412     ctx: Context<'info, InitializeRevenueShareEscrow<'info>>,
413     num_orders: u16,
414 ) -> Result<()> {
415     let escrow = &mut ctx.accounts.escrow;
416     escrow.authority = ctx.accounts.authority.key();
417     escrow
418         .orders
419         .resize_with(num_orders as usize, RevenueShareOrder::default);
420
421     let mut user_stats = ctx.accounts.user_stats.load_mut()?;
422     escrow.referrer = user_stats.referrer;
423     user_stats.update_builder_referral_status();
424
425     escrow.validate()?;
426     Ok(())
427 }
428
```

```rust
pub fn handle_resize_revenue_share_escrow_orders<'c: 'info, 'info>(
    ctx: Context<'info, ResizeRevenueShareEscrowOrders<'info>>,
    num_orders: u16,
) -> Result<()> {
    let escrow = &mut ctx.accounts.escrow;
    validate!(
        num_orders as usize >= escrow.orders.len(),
        ErrorCode::InvalidRevenueShareResize,
        "Invalid shrinking resize for revenue share escrow"
    )?;

    escrow
        .orders
        .resize_with(num_orders as usize, RevenueShareOrder::default);
    escrow.validate()?;
    Ok(())
```

When an escrow is initialized with zero order capacity, referral row creation for that authority cannot reserve a slot. Recovery is possible because the resize handler permits non-shrinking growth, but the affected parties may not learn from the transaction result that referral economics were skipped.

Remediation

Change `find_or_create_referral_index` to return a `Result<Option<u32>>` or equivalent error-bearing type, and propagate `RevenueShareEscrowOrdersAccountFull` when an escrow with a referrer cannot allocate the required referral row. Update the call sites that compute `reward_referrer` and builder/referral escrow info so a referral-bearing fill fails when the referral row cannot be created, while still returning the no-referral case only for escrows that genuinely have no referrer.

Variable Slot Times Alter Accounting Calculations

ID	Severity	Status
OS-VEP-SUG-02	Informational	✓ Resolved

Description

The new clock logic in `active_slot_duration_ms` selects a single duration at the queried slot and applies it to the entire historical slot delta, even when the interval crosses a slot-duration transition. Consequently, elapsed time may be under- or overestimated, causing discontinuities in auctions, expiries, cooldowns, and keeper scheduling.

programs/velocity/src/math/time.rs

```
267 pub const fn active_slot_duration_ms(
268     base_ms: u16,
269     pending_ms: u16,
270     effective_slot: u64,
271     now_slot: u64,
272 ) -> u16 {
273     if pending_ms != 0 && now_slot >= effective_slot {
274         pending_ms
275     } else {
276         base_ms
277     }
278 }
```

Moreover, `feature_gate_effective_slot` adds a fixed warmup without ensuring the result aligns with an actual epoch boundary. If activation occurs mid-epoch, Velocity may apply the reduced slot duration prematurely and accelerate time-based accounting for part of an epoch. The implementation should derive the start of the next epoch using Solana's `EpochSchedule` rather than hard-coding `432_000`, which may not hold across clusters or schedule configurations.

programs/velocity/src/instructions/admin.rs

```
1 fn feature_gate_effective_slot(account: &AccountInfo, expected: &Pubkey) ->
  Result<u64> {
2     [...]
3     let activated_at = u64::from_le_bytes(data[1..9].try_into().unwrap());
4     Ok(activated_at.saturating_add(FEATURE_WARMUP_SLOTS))
5 }
```

`get_auction_duration` converts the intended wall-clock duration into slots but caps the result at `u8::MAX`, while fixed-auction interpolation in `calculate_auction_price_for_fixed_auction` treats the stored `u8` as the authoritative duration. Any configuration requiring more than 255 slots is therefore shortened. This changes the intended price ramp and may give makers less execution time than configured.

Remediation

Use `EpochSchedule` to derive the correct next epoch boundary.

Patch

Resolved in PR#425 and PR#464.

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.

The design review asks whether the programs's architecture and operating assumptions support its intended behavior. We study the inputs it accepts, the decisions it makes, the information shared between components, and the services or participants it trusts. We then consider how it behaves when those inputs are unexpected or conflicting. The goal is to find conditions that can produce an incorrect decision, inconsistent state, or loss of service.

Design problems often arise from interactions rather than one defective function. Two components may each process a value correctly but interpret it differently. A failure may also leave one component ready to continue while another treats the same operation as incomplete. Local checks can look correct in both cases, so we follow the complete operation across component boundaries.

The implementation review asks whether the code matches the design. We trace untrusted data from its entry point to the point where it affects behavior, checking the assumptions made along the way. The review covers validation, error handling, state updates, concurrency, persistence, resource use, and cleanup. Its exact focus depends on the target's language, runtime, and architecture rather than a fixed list of bug classes.

The boundary between design and implementation is not exact. Missing validation may be a local coding error, while the same omission across several components may reflect an unclear rule. We classify a finding by its root cause, but judge its impact from the resulting behavior and the conditions needed to trigger it.

We begin by reading the target as a team and agreeing on its main components, trust boundaries, and expected behavior. Auditors then review different paths through the system and compare notes where those paths meet. We reproduce suspected issues with focused tests or traces when the setup allows. Before reporting a finding, another auditor reviews its trigger, impact, and proposed remediation.

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