KAIROSSWAP · SECURITY REVIEW

KairosSwap Protocol Review.

A security review of Protocol Review.

CLIENT	PROJECT	AUDITORS	AUDIT WINDOW	FINAL REPORT
KairosSwap	Protocol Review	Roberto Reigada, 0xCiphky .	March 9 – July 9, 2026	July 9, 2026

§ 00

Contents.

312 pages: overview, confidence ranking, scope, 191 findings, disclaimer, and about Guardian.

§ 01 **Overview** P. 03

§ 02 **Confidence Ranking** P. 04

§ 03 **Scope & Classifications** P. 05

§ 04 **Findings** P. 06

- Main Review 1 · 58 findings P. 14
- Remediation Review 1 · 15 findings P. 114
- Main Review 2 · 46 findings P. 131
- Remediation Review 2 · 6 findings P. 197
- Remediation Review 4 · 66 findings P. 208

§ 05 **Disclaimer** P. 311

§ 06 **About Guardian** P. 312

§ 01

Overview.

Guardian reviewed Protocol Review across 6 review rounds. The final report includes 12 High, 53 Medium, 84 Low, 42 Informational findings.

AUDIT FIRM
Guardian

CLIENT
KairosSwap

PROJECT
Protocol Review

PREPARED BY
Roberto Reigada, 0xCiphky .

AUDIT WINDOW
March 9 – July 9, 2026

FINAL REPORT
July 9, 2026

REPOSITORY
https://github.com/kairosswap/protocol

METHODOLOGY
Smart contract · Manual · Main Review 1 · Remediation Review 1 · Main Review 2 · Remediation Review 2 · Remediation Review 3 · Remediation Review 4

MAIN REVIEW 1 COMMIT
e81ff86a39a1

REMEDIATION REVIEW 1 COMMIT
42c18db57f99

MAIN REVIEW 2 COMMIT
235c635fd1ba

1.1 Confidence Ranking

2 / 5

Low Confidence

Guardian assigns a Confidence Ranking of 2 out of 5, combining finding severity, remediation outcome, and the reviewer's assessment of latent risk. The full scale is on the next page.

1	2	3	4	5
VERY LOW CONFIDENCE	LOW CONFIDENCE	MODERATE CONFIDENCE	HIGH CONFIDENCE	VERY HIGH CONFIDENCE

1.2 Findings at a Glance

SEVERITY	TOTAL	PENDING	ACK	PARTIAL	RESOLVED	FIX REVIEW
Critical	**0**	0	0	0	0	0
High	**12**	0	0	0	6	6
Medium	**53**	0	15	3	26	9
Low	**84**	1	10	1	65	7
Informational	**42**	3	14	2	20	3
All severities	**191**	**4**	**39**	**6**	**117**	**25**

§ 02

Confidence Ranking.

Guardian assigns a project-level confidence score using the scale below. The ranking combines finding severity, remediation outcome, and the reviewer's assessment of latent risk.

CONFIDENCE RANKING	DEFINITION AND RECOMMENDATION	RISK PROFILE
5 Very High Confidence	Codebase is mature, clean, and secure. No High or Critical vulnerabilities were found. Follows modern best practices with high test coverage and thoughtful design. Recommendation: code is highly secure at time of audit.	0 High/Critical findings and few Low/Medium severity findings.
4 High Confidence	Code is clean, well-structured, and adheres to best practices. Design patterns are sound and test coverage is strong. Recommendation: suitable for deployment after remediations; consider periodic review with changes.	0-1 High/Critical findings per engagement week and little to no Medium severity findings.
3 Moderate Confidence	Code is functional but shows gaps in defensive programming or test coverage. Recommendation: remediate the findings and re-review the changes before deployment.	Some High/Critical findings, with various Medium severity findings.
2 Low Confidence ASSIGNED	Recurring issues suggest systemic gaps rather than isolated mistakes. Recommendation: remediate, then commission a follow-up review at a frozen commit.	Over 6 High and Critical findings across the engagement.
1 Very Low Confidence	The volume and severity of findings indicate the codebase is not ready for production. Recommendation: substantial rework, then a full re-review.	Over 12 High and Critical findings across the engagement.

§ 03

Scope & Classifications.

Guardian reviewed Protocol Review in `https://github.com/kairosswap/protocol`.

IN SCOPE

https://github.com/kairosswap/protocol/blob/dev/packages/hardhat/contracts/oracles/FixedRatePricingOracl

3.1 Severity Matrix

	IMP · HIGH	IMP · MED	IMP · LOW
LIK · HIGH	Critical	High	Medium
LIK · MED	High	Medium	Low
LIK · LOW	Medium	Low	Low

IMPACT · HIGH	Significant loss of assets, harm to a group of users, or core protocol functionality is disrupted.
IMPACT · MEDIUM	Small amount of funds can be lost, or ancillary functionality is affected. Users may experience reduced or delayed receipt.
IMPACT · LOW	Unexpected behavior in some functionality that is notable but does not meet the criteria for higher severity.
LIKELIHOOD · HIGH	Possible with reasonable assumptions that mimic on-chain conditions; cost is low compared to amount gained.
LIKELIHOOD · MEDIUM	Possible only in uncommon cases or requires a large amount of capital relative to disruption caused.
LIKELIHOOD · LOW	Unlikely to ever occur in production.

§ 04

Findings.

Grouped by round. Visible IDs reset by severity within each round, so a finding is named the same way in the report as it was in the channel it was written in. Status is where each finding ended up; its own page says how.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
MAIN REVIEW 1					
H-01	Buyer adapter `realAssets` overstates NAV	Logical Error	HIGH	RESOLVED	p. 14
H-02	LP share price ignores settlement caps	Logical Error	HIGH	RESOLVED	p. 16
H-03	Buyer adapter allows forced early exits	Access Control	HIGH	RESOLVED	p. 19
H-04	Bucket Overlap Corrupts Share Pricing	Validation	HIGH	RESOLVED	p. 21
H-05	Buyer Deallocate Can Consume Unrelated Swap	DoS	HIGH	RESOLVED	p. 23
M-01	Expired swaps keep repricing after maturity	Unexpected Behavior	MEDIUM	RESOLVED	p. 24
M-02	Morpho adapters return raw cashflow change	Logical Error	MEDIUM	RESOLVED	p. 28
M-03	Buyer adapter omits escrowed assets from NAV	Unexpected Behavior	MEDIUM	RESOLVED	p. 31
M-04	Latest rate rewrites stale floating accrual	Oracle	MEDIUM	PARTIALLY RESOLVED	p. 34
M-05	Redeem Can Strand Wrapped Escrow	DoS	MEDIUM	RESOLVED	p. 38
M-06	BaseSwapRate getRate() Ignores Live Inputs	Oracle	MEDIUM	ACKNOWLEDGED	p. 39
M-07	Escrow claim cannot bypass blocked owners	Unexpected Behavior	MEDIUM	ACKNOWLEDGED	p. 40
M-08	Rate index compounds with update cadence	Logical Error	MEDIUM	RESOLVED	p. 42
M-09	LP full withdraw skips forcedeallocation penalty	Logical Error	MEDIUM	RESOLVED	p. 45
M-10	AbstractReferenceRate fabricates freshness	Oracle	MEDIUM	ACKNOWLEDGED	p. 47
M-11	TWAP history corrupts on source rotation	Unexpected Behavior	MEDIUM	ACKNOWLEDGED	p. 49
M-12	US10YR Pyth trades outside feed hours	Unexpected Behavior	MEDIUM	ACKNOWLEDGED	p. 51
M-13	L2 Sequencer Check Missing	Oracle	MEDIUM	RESOLVED	p. 52
M-14	Shared LP vesting blocks unrelated withdrawals	DoS	MEDIUM	RESOLVED	p. 53
M-15	Utilization Fees Can Be Reduced by Splitting	Logical Error	MEDIUM	RESOLVED	p. 55
M-16	Zero Share Supply Can Trigger False Pool Reset	Logical Error	MEDIUM	RESOLVED	p. 56
M-17	Uninitialized buckets can return valid rates	Validation	MEDIUM	RESOLVED	p. 57

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
M-18	Stale Buckets Can Pass Validation	Oracle	MEDIUM	RESOLVED	p. 58
L-01	Bucket saturation inflates LP pricing gas	Gas Griefing	LOW	RESOLVED	p. 59
L-02	Adapters stale-price realAssets on oracle fail	Oracle	LOW	RESOLVED	p. 61
L-03	Vault V2 cannot recover adapter escrow	Unexpected Behavior	LOW	RESOLVED	p. 63
L-04	Buyer adapter realAssets can DoS valuation	DoS	LOW	RESOLVED	p. 66
L-05	Factory deploys unscoped market adapter	Unexpected Behavior	LOW	RESOLVED	p. 68
L-06	Collateralization ratio runs backwards	Unexpected Behavior	LOW	RESOLVED	p. 70
L-07	Full withdraw mode breaks vault pull flow	Validation	LOW	RESOLVED	p. 72
L-08	isSwapActive is true for unknown ids	Warning	LOW	RESOLVED	p. 73
L-09	Unset multisig disables market creation fees	Unexpected Behavior	LOW	RESOLVED	p. 74
L-10	Protocol fee timelock can be bypassed	Unexpected Behavior	LOW	RESOLVED	p. 75
L-11	minTradeSize comment contradicts validation	Validation	LOW	RESOLVED	p. 77
L-12	SwapClosed emits zero collateralBalance	Events	LOW	RESOLVED	p. 78
L-13	Market creation ignores oracle validity	Validation	LOW	RESOLVED	p. 80
L-14	Deploy script wires TWAP as Base-Rate oracle	Warning	LOW	ACKNOWLEDGED	p. 81
L-15	Bounded early exit breaks on escrow fallback	Logical Error	LOW	RESOLVED	p. 83
L-16	Shared liquidityData breaks small withdrawals	Unexpected Behavior	LOW	RESOLVED	p. 85
L-17	PythAdapter Confidence Interval Not Validated	Oracle	LOW	RESOLVED	p. 87
L-18	BUY_FLOATING slippage guard ignores markup	Unexpected Behavior	LOW	RESOLVED	p. 88
L-19	Swap transfer depends on oracle freshness	Unexpected Behavior	LOW	RESOLVED	p. 89
L-20	Duplicate tenors can bypass maxRateChangeWad	Validation	LOW	RESOLVED	p. 91
L-21	Interpolation can mix different forecast epochs	Unexpected Behavior	LOW	RESOLVED	p. 92
L-22	Last LP full withdraw leaves residual collateral	Warning	LOW	RESOLVED	p. 93
L-23	calculateMaxLeverage under-scales WAD output	Documentation	LOW	RESOLVED	p. 95
L-24	Liquidity Routing Can Block Deposits	DoS	LOW	RESOLVED	p. 97
I-01	Buyer factory accepts counterfeit vaults	Trust Assumptions	INFORMATIONAL	RESOLVED	p. 98

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
I-02	Unused totalNotionalReduction in settleSwap	Informational	INFORMATIONAL	RESOLVED	p. 100
I-03	ImpliedBaseRate lets users manipulate pricing	Logical Error	INFORMATIONAL	ACKNOWLEDGED	p. 101
I-04	US10YR cranker pays for unchanged Pyth updates	Oracle	INFORMATIONAL	RESOLVED	p. 104
I-05	Use minimal proxies for buyer adapters	Gas Optimization	INFORMATIONAL	ACKNOWLEDGED	p. 106
I-06	Avoid copying swapId arrays in buyer views	Gas Optimization	INFORMATIONAL	RESOLVED	p. 107
I-07	Add known-settled credit helper in wrapper	Gas Optimization	INFORMATIONAL	RESOLVED	p. 108
I-08	Reuse updated index in LP share price path	Gas Optimization	INFORMATIONAL	ACKNOWLEDGED	p. 109
I-09	Cache swapPositions in settleBatch	Gas Optimization	INFORMATIONAL	RESOLVED	p. 110
I-10	Cache fresh rate index in buySwap	Gas Optimization	INFORMATIONAL	RESOLVED	p. 111
I-11	PR #544 fix leaves post-expiry repricing open	Warning	INFORMATIONAL	RESOLVED	p. 112
REMEDIATION REVIEW 1					
M-01	Cumulative markets still call getRate()	Logical Error	MEDIUM	RESOLVED	p. 114
M-02	Aave borrow index grows with zero debt	Unexpected Behavior	MEDIUM	ACKNOWLEDGED	p. 115
M-03	Morpho borrow share price is not cumulative	Unexpected Behavior	MEDIUM	ACKNOWLEDGED	p. 116
M-04	Expiry pricing includes post-expiry moves	Unexpected Behavior	MEDIUM	ACKNOWLEDGED	p. 117
M-05	LP Gate Triggers Before Expiry	Logical Error	MEDIUM	RESOLVED	p. 118
M-06	LP adapter reverts on last-share withdraw	Unexpected Behavior	MEDIUM	ACKNOWLEDGED	p. 119
M-07	Bucket PnL nets before swap caps	Logical Error	MEDIUM	ACKNOWLEDGED	p. 120
M-08	Mixed-expiry bucket misprices LP MtM	Logical Error	MEDIUM	RESOLVED	p. 122
M-09	Liquidatable swaps ignore liquidator haircut	Logical Error	MEDIUM	PARTIALLY RESOLVED	p. 124
L-01	Epoch pointer can clear too early	Unexpected Behavior	LOW	RESOLVED	p. 125
L-02	Donated funds cannot be swept at zero allocation	Warning	LOW	RESOLVED	p. 126
L-03	Buyer cleanup can fail on unrelated bad oracle	DoS	LOW	RESOLVED	p. 127
L-04	Market creation fee can change before create	Trust Assumptions	LOW	RESOLVED	p. 128
I-01	Spot-compound projection can revert	Math	INFORMATIONAL	PARTIALLY RESOLVED	p. 129
I-02	Expired swap can freeze LP mobility	Unexpected Behavior	INFORMATIONAL	ACKNOWLEDGED	p. 130

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
MAIN REVIEW 2					
H-01	Direct settlement stale-prices vault shares	Validation	HIGH	RESOLVED	p. 131
M-01	Fresh LP deposits underprice swap fees	Unexpected Behavior	MEDIUM	ACKNOWLEDGED	p. 134
M-02	Morpho adapters accept stale indexes	Validation	MEDIUM	RESOLVED	p. 136
M-03	Buyer adapter cleanup erases live allocation	Logical Error	MEDIUM	RESOLVED	p. 138
M-04	Morpho share ratios can reset cumulative indexes	Configuration	MEDIUM	ACKNOWLEDGED	p. 140
M-05	Buyer entry fees can be socialized	Unexpected Behavior	MEDIUM	ACKNOWLEDGED	p. 142
M-06	Poke spam blocks TWAR ring advancement	Unexpected Behavior	MEDIUM	RESOLVED	p. 144
M-07	Permissionless updates starve rate snapshots	Logical Error	MEDIUM	RESOLVED	p. 145
L-01	Buyer adapter counts liquidator bounty in NAV	Logical Error	LOW	RESOLVED	p. 147
L-02	Expired settlement still needs live oracle	DoS	LOW	RESOLVED	p. 148
L-03	SpotCompound floor blocks settlement	DoS	LOW	ACKNOWLEDGED	p. 150
L-04	Same-block cumulative index is stale	Logical Error	LOW	RESOLVED	p. 152
L-05	Compound adapter accepts stale index	Unexpected Behavior	LOW	RESOLVED	p. 154
L-06	Idle LP withdrawals need live oracle	Unexpected Behavior	LOW	RESOLVED	p. 156
L-07	Market creation skips base batch API check	Validation	LOW	RESOLVED	p. 157
L-08	New bounds leave stored rates valid	Validation	LOW	RESOLVED	p. 158
L-09	Buyer adapter drops early-exit slippage	Validation	LOW	RESOLVED	p. 159
L-10	Market adapter leaves withdrawal surplus	Unexpected Behavior	LOW	RESOLVED	p. 160
L-11	Stale signed oracle updates appear fresh	Validation	LOW	RESOLVED	p. 161
L-12	Inherited renounce bypasses wrapper timelock	Unexpected Behavior	LOW	RESOLVED	p. 163
L-13	Long min poke interval forces stale rates	Validation	LOW	ACKNOWLEDGED	p. 164
L-14	Tenor wrapper accepts short rate batches	Validation	LOW	RESOLVED	p. 165
L-15	Late recovery pokes smear Aave index growth	Logical Error	LOW	PENDING	p. 166
L-16	Long update interval forces invalid windows	Validation	LOW	RESOLVED	p. 167
L-17	Wrappers reject non-ERC165 oracle adapters	Compatibility	LOW	ACKNOWLEDGED	p. 168

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
L-18	Pyth adapter reverts on invalid prices	Unexpected Behavior	LOW	RESOLVED	p. 169
L-19	Short min spacing can erase TWAR history	Validation	LOW	RESOLVED	p. 170
L-20	Tenor bucket activation blocks reseeding	Validation	LOW	RESOLVED	p. 171
L-21	MIN_SHARE_PRICE floor overpays withdrawing LPs	Unexpected Behavior	LOW	ACKNOWLEDGED	p. 173
L-22	IndexBaseRateV1 ignores post-target anchors	Unexpected Behavior	LOW	RESOLVED	p. 174
L-23	Market Can Use Wrong Risk Tenor	Validation	LOW	ACKNOWLEDGED	p. 176
I-01	Virtual share price scales with expired queue	Warning	INFORMATIONAL	ACKNOWLEDGED	p. 177
I-02	Compound borrow index accrues with zero debt	Unexpected Behavior	INFORMATIONAL	ACKNOWLEDGED	p. 178
I-03	Early exit bypasses liquidation rewards	Warning	INFORMATIONAL	ACKNOWLEDGED	p. 180
I-04	Kairos oracles freeze negative-rate markets	Logical Error	INFORMATIONAL	PENDING	p. 181
I-05	TWAPOracle backfills current spot rate	Unexpected Behavior	INFORMATIONAL	PENDING	p. 182
I-06	Exact expiry can be early exited	Warning	INFORMATIONAL	RESOLVED	p. 183
I-07	Spot bucket aggregation biases MTM	Warning	INFORMATIONAL	ACKNOWLEDGED	p. 184
I-08	Referral IDs have 48-bit collision space	Warning	INFORMATIONAL	PENDING	p. 185
I-09	Self-downstream permanently bricks wrapper	Warning	INFORMATIONAL	RESOLVED	p. 186
I-10	Cutoff cannot mask downstream reverts	Warning	INFORMATIONAL	RESOLVED	p. 187
I-11	Pyth confidence config can overflow	Warning	INFORMATIONAL	RESOLVED	p. 188
I-12	Risk premiums can cross between sides	Warning	INFORMATIONAL	ACKNOWLEDGED	p. 189
I-13	Missing source-market caps for rate oracles	Warning	INFORMATIONAL	ACKNOWLEDGED	p. 190
I-14	Flat curve extrapolates tenors	Warning	INFORMATIONAL	PARTIALLY RESOLVED	p. 194
I-15	Nonexistent markets report default share price	Warning	INFORMATIONAL	RESOLVED	p. 196

REMEDIATION REVIEW 2

M-01	Morpho sources can brick base-rate TWAR pokes	DoS	MEDIUM	RESOLVED	p. 197
M-02	Oracle updates can sandwich LP shares	MEV	MEDIUM	PARTIALLY RESOLVED	p. 199
L-01	Max live gap can revive stale base-rate quotes	Unexpected Behavior	LOW	RESOLVED	p. 202
L-02	Expired adapter swaps keep liquidation haircuts	Unexpected Behavior	LOW	RESOLVED	p. 203

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
L-03	Buy swap lacks total input bound	Unexpected Behavior	LOW	RESOLVED	p. 205
L-04	Oversized indexes hide bucket floating exposure	Unexpected Behavior	LOW	RESOLVED	p. 206
REMEDIATION REVIEW 4					
H-01	Cumulative index basis drains LP backing	Unexpected Behavior	HIGH	REPORTED	p. 208
H-02	Interim liquidation drains LP backing	Math	HIGH	REPORTED	p. 214
H-03	Dead-oracle fallback drains LP backing	Unexpected Behavior	HIGH	REPORTED	p. 216
H-04	Early exit drains LP backing via linear accrual	Math	HIGH	REPORTED	p. 218
H-05	LP adapter NAV ignores liquidation floor	Unexpected Behavior	HIGH	REPORTED	p. 221
H-06	Force-dealloc forfeits buyer liquidation value	Validation	HIGH	REPORTED	p. 223
M-01	LP MTM Omits Cumulative Cross-Term	Logical Error	MEDIUM	RESOLVED	p. 225
M-02	Stale anchors let top-ups bypass vesting	Unexpected Behavior	MEDIUM	RESOLVED	p. 227
M-03	Force-deallocation can burn capped LP shares	Unexpected Behavior	MEDIUM	RESOLVED	p. 229
M-04	Morpho bad debt freezes supply-rate high-water	Oracle	MEDIUM	RESOLVED	p. 231
M-05	Spot-rate index floor erases negative accrual	Logical Error	MEDIUM	ACKNOWLEDGED	p. 233
M-06	Vest-Capped Adapter NAV Dilutes Vault LPs	Logical Error	MEDIUM	RESOLVED	p. 235
M-07	Source rollback quotes frozen index	Logical Error	MEDIUM	RESOLVED	p. 237
M-08	Morpho cadence skews spot accrual	Oracle	MEDIUM	ACKNOWLEDGED	p. 239
M-09	Dust force-dealloc burns valuable LP shares	Rounding	MEDIUM	REPORTED	p. 241
M-10	Base cumulative oracles still use stale adapters	Configuration	MEDIUM	REPORTED	p. 243
M-11	Aave index jumps inflate BUY_FLOATING profit	Unexpected Behavior	MEDIUM	REPORTED	p. 245
M-12	Base rate batch quotes can diverge	Validation	MEDIUM	REPORTED	p. 248
M-13	Partial forceDeallocate can fully exit LP	Unexpected Behavior	MEDIUM	REPORTED	p. 249
M-14	Factory misses non-exit adapter guard	Validation	MEDIUM	REPORTED	p. 250
M-15	Pre-poke reference indexes skew payouts	Unexpected Behavior	MEDIUM	REPORTED	p. 251
M-16	Negative tenor quote inflates LP NAV	Validation	MEDIUM	REPORTED	p. 253
M-17	RawIndex underprices BUY_FIXED LP backing	Configuration	MEDIUM	REPORTED	p. 255

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
L-01	LP withdrawals lack max-share slippage	Warning	LOW	RESOLVED	p. 257
L-02	Short-tenor liquidity floors to zero	Rounding	LOW	RESOLVED	p. 258
L-03	Dust collateral blocks full adapter exits	DoS	LOW	ACKNOWLEDGED	p. 260
L-04	PythAdapter rejects valid negative rates	Oracle	LOW	RESOLVED	p. 261
L-05	Settled swap ID reverts settlement batch	Warning	LOW	RESOLVED	p. 262
L-06	LP adapter bypasses market whitelist	Validation	LOW	ACKNOWLEDGED	p. 263
L-07	Idle LP value query needs live oracle	Warning	LOW	RESOLVED	p. 264
L-08	First market can lock spot oracle convention	DoS	LOW	ACKNOWLEDGED	p. 265
L-09	Long-tenor base-rate compounding can revert	Unexpected Behavior	LOW	RESOLVED	p. 266
L-10	Pool-zero event misses exact drains	Events	LOW	RESOLVED	p. 267
L-11	SwapClosed reports accrued unsigned legs	Events	LOW	RESOLVED	p. 268
L-12	Sweep leaves stale tracked swap	Unexpected Behavior	LOW	RESOLVED	p. 270
L-13	Delayed relays bypass max-rate-change breaker	Oracle	LOW	RESOLVED	p. 272
L-14	Duplicate tenor probe accepts bad batches	Validation	LOW	RESOLVED	p. 274
L-15	Default tenor probe misses live tenors	Validation	LOW	RESOLVED	p. 275
L-16	Tenor removal reuses old premium anchors	Oracle	LOW	RESOLVED	p. 276
L-17	No-op wrapper poke carries cooldown	Oracle	LOW	RESOLVED	p. 277
L-18	Negative premiums brick swap entry	Oracle	LOW	RESOLVED	p. 278
L-19	Virtual NAV skips settlement pokes	Unexpected Behavior	LOW	ACKNOWLEDGED	p. 279
L-20	Forced Bounds Reset Keeps Old Posting Cooldown	Oracle	LOW	RESOLVED	p. 280
L-21	Over-aged seed starts oracle invalid	Validation	LOW	RESOLVED	p. 281
L-22	Tenor shortening can brick live markets	Unexpected Behavior	LOW	PARTIALLY RESOLVED	p. 282
L-23	Ghost swap escrow cannot be claimed	Unexpected Behavior	LOW	REPORTED	p. 283
L-24	Long min spacing creates oracle outages	Validation	LOW	REPORTED	p. 285
L-25	BaseSwapRate average accepts bad rates	Unexpected Behavior	LOW	REPORTED	p. 287
L-26	ProtocolFeeForegone omits fee context	Events	LOW	REPORTED	p. 288
L-27	Permissionless markets bloat required tenors	Validation	LOW	REPORTED	p. 289

Findings continued.

ID	TITLE	CATEGORY	SEVERITY	STATUS	PAGE
L-28	ReferenceRate ignores configured rates	Unexpected Behavior	LOW	REPORTED	p. 290
L-29	PythAdapter rejects fresh zero-percent rates	Validation	LOW	REPORTED	p. 292
I-01	Adapter NAV omits idle deallocated cash	Reentrancy	INFORMATIONAL	ACKNOWLEDGED	p. 293
I-02	Split exits round away swap fees	Rounding	INFORMATIONAL	ACKNOWLEDGED	p. 294
I-03	Callback credits wrapped payout early	Reentrancy	INFORMATIONAL	RESOLVED	p. 295
I-04	Split swaps bypass aggregate collateralization	Rounding	INFORMATIONAL	RESOLVED	p. 296
I-05	Fee receiver can block market actions	Unexpected Behavior	INFORMATIONAL	RESOLVED	p. 298
I-06	Unset Morpho ids report zero rates	Validation	INFORMATIONAL	RESOLVED	p. 300
I-07	BaseSwapRate cannot serve SwapCore	Warning	INFORMATIONAL	ACKNOWLEDGED	p. 301
I-08	Cached oracle health is skipped	Validation	INFORMATIONAL	RESOLVED	p. 302
I-09	Redeem callback can sell live NFT	Reentrancy	INFORMATIONAL	RESOLVED	p. 303
I-10	Tenor cardinality can exhaust gas	Gas Optimization	INFORMATIONAL	RESOLVED	p. 304
I-11	Pool-side self-liquidation captures reward	Warning	INFORMATIONAL	ACKNOWLEDGED	p. 306
I-12	Unbounded poke spacing disables refresh	Validation	INFORMATIONAL	REPORTED	p. 307
I-13	Split buys avoid fees via rounding	Rounding	INFORMATIONAL	REPORTED	p. 308
I-14	SpotCompound snapshots truncate extreme indexes	Warning	INFORMATIONAL	REPORTED	p. 309

HIGH **H-01** RESOLVED

Buyer adapter `realAssets` overstates NAV

CATEGORY	Logical Error
LOCATION	KairosBuyerAdapter.sol
REVIEW	Main Review 1

DESCRIPTION

The buyer adapter reports unsettled swaps as if the buyer can always collect the full theoretical net PnL. The root cause is in the valuation path below, which uses a theoretical number, together with the settlement path below, which pays only against reserved backing:

```solidity
(uint256 netAmount, uint8 netRecipient) = swapCore.getSwapNetAmount(swapIds[j]);
if (netRecipient == 1) {
assets += collateralBalance + netAmount;
} else {
assets += netAmount >= collateralBalance ? 0 : collateralBalance - netAmount;
}
```

That looks reasonable at first glance, but `getSwapNetAmount` is only a pricing view. It does not apply the hard settlement caps that the protocol enforces when money actually moves. On the buyer-positive path, `settleSwap` pays no more than the backing reserved for that swap:

```solidity
uint256 availableBacking = swap.poolCollateralBacking;
settlementAmount = availableBacking >= netObligation ? netObligation : availableBacking;
swap.collateralBalance += settlementAmount;
```

So the adapter can report value that simply is not there. If rates move sharply in the buyer's favor, `realAssets` can rise by the full theoretical `netAmount` even though the buyer can recover only the smaller of that number and `poolCollateralBacking`, with liquidation reducing it further in stressed cases.

Do note that the Morpho `VaultV2` accounting path consumes adapter `realAssets()` directly. In `VaultV2.accrueInterestView`, the vault starts from its idle token balance and then adds `IAdapter(adapters[i]).realAssets()` for each adapter to compute the vault's current asset base. That same path then feeds `totalAssets`, `previewDeposit`, `previewMint`, `previewWithdraw`, `previewRedeem`, `convertToShares`, and `convertToAssets`. In other words, an overstated buyer-adapter mark is not just a bad view function: it directly changes

H-01 Buyer adapter `realAssets` overstates NAV CONTINUED

ERC4626 share pricing and fee accrual in the vault that holds the adapter. Direct cap accounting is updated from adapter `change`, not from `realAssets`, but the vault NAV and share exchange rate are still driven by this inflated figure.
A realistic example is a vault where Alice is already deposited through the buyer adapter. One open swap is marked at 150000 tokens of profit by `realAssets`, but the position can actually collect only 20000 because that is all the LP side reserved. Bob deposits after the mark-up and gets too few vault shares because the vault believes it is richer than it really is. Alice can then redeem against the inflated NAV before the swap is settled, leaving Bob to absorb the gap once the position closes at the lower, real amount.

RECOMMENDATION

`realAssets` should use a claimable value, not a theoretical one. The best fix is to add a `SwapCore` view that returns the buyer's current conservative recovery after applying the same caps and liquidation logic used by settlement, and then build adapter NAV from that value. At a minimum, the buyer-positive branch must be capped by `poolCollateralBacking` instead of adding the full `netAmount`.

PROOF OF CONCEPT `https://gist.github.com/GuardianAudits/b682d6dab3655cf2bd64b3684a2f928e`

RESOLUTION

Status: Resolved.

HIGH **H-02** RESOLVED

LP share price ignores settlement caps

CATEGORY	Logical Error
LOCATION	`Utils.sol`
REVIEW	Main Review 1

DESCRIPTION

The LP share price is built from full mark-to-market PnL, but settlement is not. The root cause is the mismatch between the exact pricing and settlement paths below. The pricing path first computes bucket PnL from full accrued plus projected swap payments, without capping that value by the swap's remaining `collateralBalance` or `poolCollateralBacking`, and then adds the aggregate unrealized PnL straight into pool value:

```solidity
int256 totalFixedPayment = accruedFixedPayment + remainingFixedPayment;
int256 totalFloatingPayment = accruedFloatingPayment + remainingFloatingPayment;

if (market.rateType == Types.RateType.BUY_FIXED) {
lpPnL = totalFixedPayment - totalFloatingPayment;
} else {
lpPnL = totalFloatingPayment - totalFixedPayment;
}
```

```solidity
int256 unrealizedPnL = calculateTotalUnrealizedPnL(buckets, market, oracleIndex);

if (unrealizedPnL >= 0) {
mtmValue = SwapFormulas.safeCast(pool.totalCollateral) + unrealizedPnL;
} else {
uint256 absLoss = uint256(-unrealizedPnL);
if (absLoss >= pool.totalCollateral) {
return MIN_SHARE_PRICE;
}
mtmValue = SwapFormulas.safeCast(pool.totalCollateral - absLoss);
}

sharePrice = Math.mulDiv(uint256(mtmValue), WAD, pool.totalShares);
```

That number is then used immediately for LP entry and exit. New LPs mint shares against it

H-02 LP share price ignores settlement caps CONTINUED

unless the pool has already hit the special `MIN_SHARE_PRICE` floor, and exiting LPs burn shares against it:

```solidity
sharePrice = getPoolSharePrice(marketId);

if (sharePrice == Utils.MIN_SHARE_PRICE && pool.lockedCollateral > 0) {
revert Errors.E414();
}

sharesToMint = Math.mulDiv(collateralAmount, WAD, sharePrice);
```

```solidity
sharePrice = getPoolSharePrice(marketId);

if (fullAmount) {
sharesToRedeem = lpPos.shares;
withdrawalAmount = Math.mulDiv(sharesToRedeem, sharePrice, WAD);
} else {
sharesToRedeem = Math.mulDiv(amount, WAD, sharePrice, Math.Rounding.Ceil);
withdrawalAmount = amount;
}
```

The problem is that the protocol never guarantees those marks are collectible. When a swap is finally closed, the transfer is clipped to whatever collateral is still there:

```solidity
if (buyerOwes) {
uint256 availableCollateral = swap.collateralBalance;
settlementAmount = availableCollateral >= netObligation ? netObligation : availableCollateral;
swap.collateralBalance -= settlementAmount;
poolValueIncrease = settlementAmount;
} else {
uint256 availableBacking = swap.poolCollateralBacking;
settlementAmount = availableBacking >= netObligation ? netObligation : availableBacking;
swap.collateralBalance += settlementAmount;
poolValueDecrease = settlementAmount;
}
```

That gap is enough to move value between LPs. If the theoretical LP gain is larger than the buyer's remaining collateral, the share price is too high and an LP can withdraw against

H-02 LP share price ignores settlement caps CONTINUED

profit the pool will never collect. If the theoretical LP loss is larger than the backing reserved for that swap, the share price is too low and a new LP can mint too many shares before the loss is clipped on settlement. In both directions the pool is using a number that looks like PnL, but is not the amount the system can actually realize.

As an example, imagine that Alice and Bob are the only LPs in a market. A large swap moves sharply in the pool's favor, so the share price rises because the unrealized PnL says the pool is up another 200000 tokens. In reality the buyer has only 40000 tokens of collateral left. Alice withdraws while the share price still reflects the full 200000 gain, so she burns too few shares and walks away with more than her fair share of the pool. When the swap is finally settled and the pool collects only the 40000 that was actually posted, Bob is the one left holding the shortfall.

RECOMMENDATION

The share price needs to be based on realizable value, not raw swap economics. The clean fix is to value each live swap individually and cap its contribution by the amount the pool can actually win or lose under the current `collateralBalance`, `poolCollateralBacking` and liquidation rules. If the bucket model is kept, then LP minting and burning should be disabled whenever the bucket aggregate can no longer represent a conservative bound on settlement value.

PROOF OF CONCEPT https://gist.github.com/GuardianAudits/238b432b2c73af7dfc83b9a17f1eaca8

RESOLUTION

Status: Resolved.

HIGH **H-03** RESOLVED

Buyer adapter allows forced early exits

CATEGORY	Access Control
LOCATION	KairosBuyerAdapter.sol
REVIEW	Main Review 1

DESCRIPTION

`KairosBuyerAdapter` treats every `deallocate` call as if it came from a trusted strategy operator, even though Morpho Vault V2 uses the same adapter entrypoint for both allocator-controlled deallocation and permissionless force-deallocation flows. The adapter receives `selector`, `sender` and `assets` specifically so it can distinguish those contexts, but it discards all three values and only branches on the decoded action type.

On the `ACTION_EARLY_EXIT` branch the adapter simply builds a single `ExitRequest` with `minExitAmount = 0` and calls `swapCore.exitSwapEarly(requests, address(this))`. That call succeeds because the adapter itself is the owner of the swap positions. Nothing in the adapter checks whether the original vault caller was an authorized allocator, a sentinel or an arbitrary outsider entering through the vault's permissionless force-deallocation path.

```plain text
function deallocate(
bytes memory data,
uint256 assets,
bytes4 selector,
address sender
) external override onlyParentVault returns (bytes32[] memory ids, int256 change) {
selector;
sender;
assets;
...
requests[0] = ExitRequest(swapId, 0);
swapCore.exitSwapEarly(requests, address(this));
}
```

In a standard Vault V2 integration, an outsider can therefore ask the vault to force-deallocate this adapter with `ACTION_EARLY_EXIT` data and `assets = 0`. Because the adapter ignores the call context, it will unwind the live buyer position anyway. Because the amount argument is zero, the vault-side force-deallocation penalty is also zero. The result is any third party can force the strategy to realize early-exit fees, realize losses at an

H-03 Buyer adapter allows forced early exits CONTINUED

unfavorable moment or simply sabotage a vault's intended holding period whenever the underlying Kairos market has early exit enabled.

RECOMMENDATION

Do not allow `ACTION_EARLY_EXIT` to run on permissionless force-deallocation calls. The adapter should use `selector` and, if needed, `sender` to distinguish ordinary allocator-driven `deallocate` calls from vault-level `forceDeallocate` flows and should revert when early exit is requested from an untrusted context. If the strategy still needs a public force path, restrict that path to deterministic asset recovery actions such as ordinary settlement and keep discretionary strategy actions such as early exit behind explicitly authorized vault operators.

RESOLUTION

Status: Resolved.

HIGH **H-04** RESOLVED

Bucket Overlap Corrupts Share Pricing

CATEGORY	Validation
LOCATION	https://github.com/GuardianOrg/protocol-team1-1772465746057/blob/1e1150dea39d3b8c72eb06d1176e4b647fa4cfa6/packages/hardhat/contracts/SwapCore.sol#L128
REVIEW	Main Review 1

DESCRIPTION

During market creation, SwapCore.createMarket() accepts parameter sets where numBuckets * bucketInterval \< swapTerm because Utils.validateMarketParams() only enforces bucketInterval \<= swapTerm (and bucket-count bounds) but does not enforce that total bucket coverage spans the full swap term. Once such a market exists, Utils.getBucketId() wraps bucket IDs using (entryTimestamp / bucketInterval) % numBuckets, so a newly opened swap is assigned to a bucket that is still aggregating an older active swap from an earlier cycle. Utils.addSwapToBucket() then merges both positions into the same bucket aggregates even though they were opened in different time windows and should not be priced together.

Because those mixed aggregates are consumed by Utils.calculateBucketUnrealizedPnL(), Utils.calculateTotalUnrealizedPnL(), and Utils.calculateLpSharePrice(), LP mark-to-market and share price can become distorted once a bucket ID is reused while prior swaps in that bucket are still active. Depending on rate movement, this either depresses share price below fair value, allowing new depositors to acquire shares cheaply before gains settle, or inflates it above fair value, allowing existing LPs to exit at the expense of later depositors.

Since market creation is permissionless, this can arise from accidental misconfiguration due to a non-obvious parameter relationship, and can also be configured intentionally by a malicious deployer to extract mispricing.

RECOMMENDATION

Enforce numBuckets * bucketInterval \>= swapTerm during market validation so bucket IDs cannot wrap while swaps previously assigned to the same bucket are still active.

RESOLUTION

Status: Resolved.

Guardian: The current market validation still allows a bucket ID to be reused before an earlier swap assigned to that same bucket has expired. During market creation, validateMarketParams() enforces numBuckets * bucketInterval \>= swapTerm. That fixes the case where the bucket ring is shorter than the swap term, but it does not guarantee that two live swaps

H-04 Bucket Overlap Corrupts Share Pricing CONTINUED

cannot overlap in the same modulo bucket.
For example, if swapTerm = 1 day, bucketInterval = 1 hour, and numBuckets = 24, the current validation passes because 24h \>= 24h. But a swap opened at 00:59:59 in bucket 0 expires at 24:59:59, while another swap can already be opened in bucket 0 at 24:00:00. For the next 59 minutes and 59 seconds, both swaps are live and aggregated into the same bucket even though the configuration was accepted as valid.
Bucket IDs are assigned as bucketId = (entryTimestamp / bucketInterval) % numBuckets, so the bucket ring repeats every numBuckets * bucketInterval seconds. However, a swap can be opened at the very end of a bucket, while the next swap reusing that same bucket ID can be opened at the very start of the next cycle. This means the earliest same-bucket reuse can occur almost one full bucketInterval before the earlier swap expires.
As a result, the current validation does not fully prevent overlapping live swaps from being merged into the same bucket, so the protocol can reuse a bucket earlier than intended and continue pricing from mixed bucket state.

Recommendation
Update market validation so a bucket cannot be reused while a prior swap assigned to that bucket may still be active. A simple fix is to require numBuckets * bucketInterval \>= swapTerm + bucketInterval, which conservatively ensures the next generation of a bucket cannot arrive before the previous one has expired.

HIGH **H-05** RESOLVED

Buyer Deallocate Can Consume Unrelated Swap

CATEGORY	DoS
LOCATION	https://github.com/GuardianOrg/protocol-team1-1772465746057/blob/1e1150dea39d3b8c72eb06d1176e4b647fa4cfa6/packages/hardhat/contracts/adapters/KairosBuyerAdapter.sol#L287
REVIEW	Main Review 1

DESCRIPTION

KairosBuyerAdapter.deallocate() assumes buyer settlement paths use assets = 0, but does not enforce that invariant. In Kairos, settlement and liquidation are permissionless, so third parties can cause funds from a settled or liquidated buyer swap to land on the adapter before that swap is deallocated through the normal adapter flow. In Morpho Vault V2, forceDeallocate is also permissionless and accepts caller-supplied assets. Because KairosBuyerAdapter.deallocate() does not require assets == 0 for buyer settlement actions, a third party can call forceDeallocate on a different buyer position with nonzero assets, causing vault-side deallocation logic to consume unrelated idle balance already sitting on the adapter.

As a result, tokens that belong to one already-settled swap can be used during deallocation of another swap. When the original settled swap is later deallocated through the normal settled path, the adapter no longer holds enough tokens to forward its expected payout to the vault, causing that deallocation to revert and blocking recovery through the intended flow. Partial consumption of the settled swap is especially problematic because it can cause that swap's later deallocate() call to revert, blocking recovery of the remaining funds, or allow the shortfall to roll forward across later settled swaps and create a cascading deallocation failure.

Because the adapter also reports change based on the targeted swap path while VaultV2 can also pull extra tokens from shared adapter balance, vault-side cap/accounting can also become misaligned with the actual per-swap state.

RECOMMENDATION

Enforce assets == 0 for buyer settlement deallocations, including ACTION_MAKE_PAYMENT and ACTION_EARLY_EXIT, so unrelated adapter balances cannot be consumed by vault-side deallocation logic.

RESOLUTION

Status: Resolved.

MEDIUM M-01 RESOLVED

Expired swaps keep repricing after maturity

CATEGORY	Unexpected Behavior
LOCATION	SwapCore.sol, Utils.sol
REVIEW	Main Review 1

DESCRIPTION

An expired swap should be economically fixed at maturity. Here it is not. The protocol caps the payment period at the swap term, but it still derives the floating leg from the latest index using the full time from entry to the settlement transaction. As a result, a swap can continue to pick up rate movement after it has already matured.

`getSwapNetAmount()` shows the pattern clearly. Once the swap passes expiry, `timeElapsed` is capped at `market.swapTerm`, but `deriveRateFromIndex()` still uses `timeElapsedSinceSwapOpened` and the freshest oracle index:

```solidity
uint256 timeElapsedSinceSwapOpened = block.timestamp - swap.entryTimestamp;
if (timeElapsedSinceSwapOpened == 0) {
return (0, 1);
}

uint256 timeElapsed;
uint256 swapExpiry = swap.entryTimestamp + market.swapTerm;
if (block.timestamp >= swapExpiry) {
timeElapsed = market.swapTerm;
} else {
timeElapsed = block.timestamp - swap.entryTimestamp;
}

int256 floatingPaymentRate = SwapFormulas.deriveRateFromIndex(
swap.entryFloatingIndex,
_getFreshIndex(market.referenceRateOracle),
timeElapsedSinceSwapOpened
);
```

`Settlement uses the same approach.` settleSwap() `fixes` swapDuration `at the contractual term for an ordinary expiry, but it still annualizes the floating rate over the full elapsed time up to the moment someone calls settlement:`

```solidity
```

M-01 Expired swaps keep repricing after maturity CONTINUED

```
uint256 swapDuration = market.swapTerm;
if (swap.isEarlyExit) {
swapDuration = block.timestamp - swap.entryTimestamp;
}

uint256 timeElapsedSinceSwapOpened = block.timestamp - swap.entryTimestamp;
int256 floatingPaymentRate = SwapFormulas.deriveRateFromIndex(
swap.entryFloatingIndex,
oracleIndex,
timeElapsedSinceSwapOpened
);
```

`LP pricing follows the same logic. The bucket path saves` totalElapsed `before applying the maturity cap and then feeds that uncapped value into` deriveRateFromIndex() `:`

```solidity
uint256 timeElapsed = currentTime > avgEntryTime ? currentTime - avgEntryTime : 0;
uint256 totalElapsed = timeElapsed;

if (timeElapsed > market.swapTerm) {
timeElapsed = market.swapTerm;
}
```

M-01 Expired swaps keep repricing after maturity CONTINUED

```
int256 accruedFloatingRate = 0;
if (totalElapsed > 0 && avgEntryIndex > 0) {
accruedFloatingRate = SwapFormulas.deriveRateFromIndex(avgEntryIndex, oracleIndex, totalElapsed);
}
```

The effect comes from the math helpers. deriveRateFromIndex() divides by the elapsed time it receives, and calculateSwapPayment() later multiplies that annualized rate by the capped settlement duration:

```solidity
function deriveRateFromIndex(
uint256 entryFloatingIndex,
uint256 closingFloatingIndex,
uint256 timeElapsed
) public pure returns (int256) {
int256 indexRatio = safeCast(Math.mulDiv(closingFloatingIndex, WAD, entryFloatingIndex));
int256 growthFactor = indexRatio - WAD_INT;
return (growthFactor * int256(SECONDS_IN_YEAR)) / int256(timeElapsed);
}

function calculateSwapPayment(
uint256 notional,
int256 rateWad,
uint256 timeElapsed,
uint256 utilFee,
uint256 riskPremium
) public pure returns (int256) {
uint256 basePayment = Math.mulDiv(notional, abs(rateWad) * timeElapsed, SECONDS_IN_YEAR * WAD);
...
}
```

That leaves the floating payment tied to the latest closingFloatingIndex`, not the index at expiry. If the oracle index moves after maturity, the payout moves with it. Whoever chooses when to settle inherits a timing option on post-expiry rate movement, and the same drift bleeds into LP mark-to-market.
For example, imagine Alice and Bob opening identical swaps in the same market and block. Both swaps mature on Friday. Alice settles immediately, so her payout reflects the index level at expiry. Bob waits until Sunday because the floating-rate oracle has moved in his

M-01 Expired swaps keep repricing after maturity CONTINUED

favor. Even though both contracts ended on Friday, Bob gets a different settlement result simply because he delayed settlement until after the post-expiry index move had already been incorporated.
That is an avoidable pricing error. Two identical expired positions should not settle differently just because one sat around longer after maturity.

RECOMMENDATION

Once a swap expires, every pricing path should use an expiry snapshot rather than the latest index. The cleanest fix is to freeze the floating leg at `swap.entryTimestamp + market.swapTerm` and reuse that value everywhere settlement or mark-to-market is computed. If that snapshot is not stored directly, the protocol should reconstruct the index at expiry and use the same capped timestamp in both rate derivation and payment calculation. `getSwapNetAmount()`, `settleSwap()` and LP pricing all need to follow the same rule so previews, settlement, and share pricing stay aligned.

RESOLUTION

Status: Resolved.

MEDIUM M-02 RESOLVED

Morpho adapters return raw cashflow change

CATEGORY	Logical Error
LOCATION	KairosMarketAdapter.sol, KairosBuyerAdapter.sol
REVIEW	Main Review 1

DESCRIPTION

Both Kairos adapters return `change` as the amount moved in the current call, not as the change in the adapter's allocation after the action. That does not match the accounting contract expected by Morpho Vault V2, where the vault updates its stored allocation for each returned id by adding the adapter's `change`, and expects the running sum of those changes to equal the adapter's current estimated position. The LP adapter is the clearest example. On allocate it returns the deposited assets:

```solidity
ids = new bytes32[](1);
ids[0] = adapterId;
change = int256(assets);
```

On deallocate it returns the collateral withdrawn in that call:

```solidity
ids = new bytes32[](1);
ids[0] = adapterId;
change = -int256(withdrawn);
```

That would only be correct if the adapter's allocation stayed equal to cumulative deposits minus cumulative withdrawals. It does not. The LP position is marked to market through Kairos share price:

```solidity
uint256 shares = swapCore.getLpShares(marketId, address(this));
if (shares > 0) {
uint256 sharePrice = swapCore.getPoolSharePrice(marketId);
assets += Math.mulDiv(shares, sharePrice, WAD);
}
```

The buyer adapter has the same problem. On allocate it now returns the swap's on-position collateral balance, which is better than returning the raw token spend, but it is still not the same thing as `newAllocation - oldAllocation`:

M-02 Morpho adapters return raw cashflow change CONTINUED

```solidity
ids = new bytes32[](1);
ids[0] = adapterId;
change = int256(collateralBalance);
```

Its deallocate paths return the payout released in that call:

```solidity
ids = new bytes32[](1);
ids[0] = adapterId;
change = -int256(payout);
```

```solidity
ids = new bytes32[](1);
ids[0] = adapterId;
change = -int256(received);
```

But the buyer adapter's own valuation is mark-to-market and can drift materially between interactions:

```solidity
(uint256 netAmount, uint8 netRecipient) = swapCore.getSwapNetAmount(swapIds[j]);
if (netRecipient == 1) {
assets += collateralBalance + netAmount;
} else {
assets += netAmount >= collateralBalance ? 0 : collateralBalance - netAmount;
}
```

Once value moves between calls, cash moved in the latest allocate or deallocate call is no longer the same thing as the change in the adapter's actual allocation. The vault's stored allocation for `adapterId` then drifts away from the adapter's real position value. That can break cap enforcement and leave the vault making allocation decisions from stale numbers.

For example, imagine a vault with a Kairos LP position that was originally funded with 1000000 tokens. Time passes, the share price rises, and the real position value reaches 1300000, but the vault still records 1000000 for that adapter id because no interaction has happened since the original deposit. Bob allocates another 100000 tokens. The adapter returns `+100000`, so the vault updates its stored allocation to 1100000 even though the real exposure is already 1400000. From there, cap checks and reallocation decisions are being made against the wrong base number.

M-02 Morpho adapters return raw cashflow change CONTINUED

RECOMMENDATION

Both adapters should return the signed difference between the adapter's new allocation value and the vault's previously recorded allocation for that id. In practice, each call should compute the adapter's allocation before the action, compute it again after the action, and return `newAllocation - oldAllocation`. For the buyer adapter, that calculation should use a conservative realizable value rather than an uncapped theoretical PnL.

RESOLUTION

Status: Resolved.

MEDIUM M-03

RESOLVED

Buyer adapter omits escrowed assets from NAV

CATEGORY	Unexpected Behavior
LOCATION	`KairosBuyerAdapter.sol`
REVIEW	Main Review 1

DESCRIPTION

M-03 Buyer adapter omits escrowed assets from NAV CONTINUED

`KairosBuyerAdapter` can permanently underreport its value after a payout is escrowed in `SwapCore`. When settlement or liquidation owes funds to the buyer side, `Utils._transferOrEscrow` attempts the token transfer and writes the amount into `escrowedCollateral` if the transfer fails. In that case the settlement path explicitly zeroes `buyerCollateralReleased`, so `settlementPayouts` no longer reflects the amount still owed to the adapter.

```plain text
bool transferred = _transferOrEscrow(IERC20(swapToken), swap.userAddress, collateralToRelease, swapId, escrowedCollateral);
if (!transferred) {
buyerCollateralReleased = 0;
}
```

`The adapter does not account for that receivable anywhere. Its already-settled deallocation path reads only settlementPayouts, forwards that amount to the vault, and removes the swap from internal tracking even when the actual payout is sitting in escrowedCollateral. Its realAssets view only sums the adapter's token balance and unsettled swap marks, so once the swap has been removed there is no remaining path that includes the escrowed claim in NAV.`

```plain text
if (settled) {
uint256 payout = swapCore.settlementPayouts(sid);
if (payout > 0) asset.safeTransfer(parentVault, payout);
_removeSwapId(mid, sid);
change = -int256(payout);
return (ids, change);
}
```

This means a Morpho vault integrating the adapter can price shares below the value it is actually entitled to recover. A user depositing while the adapter is understated can mint too many vault shares and dilute existing depositors, with the gap realized later if the escrow is eventually claimed.

RECOMMENDATION

M-03 Buyer adapter omits escrowed assets from NAV CONTINUED

Treat escrowed amounts as part of adapter value until they are claimed. The simplest fix is to include `swapCore.escrowedCollateral(swapId)` in both the already-settled deallocation path and in `realAssets`, and to avoid pruning a settled swap from tracking until both `settlementPayouts` and `escrowedCollateral` are zero. If the intended behavior is instead to exclude temporarily unreachable payouts from NAV, that policy should be explicit and the adapter should expose a separate flag so parent vaults can halt rather than silently underpricing shares.

RESOLUTION

Status: Resolved.

MEDIUM **M-04** PARTIALLY RESOLVED

Latest rate rewrites stale floating accrual

CATEGORY	Oracle
LOCATION	`RateIndexLib.sol`
REVIEW	Main Review 1

DESCRIPTION

`RateIndexLib` accrues the floating index by taking the latest oracle rate at the moment of update and applying that one rate across the entire time gap since the last update. That is not how floating accrual should work when the oracle interface only gives the protocol a spot rate. If the rate changed during the gap, the next update does not capture the realized path. It rewrites the whole interval using the last sampled value.

The core logic is in `RateIndexLib.update()`:

```solidity
function update(RateIndex storage index, address oracle) internal returns (uint256) {
uint256 currentTimestamp = block.timestamp;
uint256 elapsed = currentTimestamp - index.lastUpdates[oracle];

if (elapsed == 0) return index.indices[oracle];

(int256 currentRate, bool isValid) = IRateOracle(oracle).getRate();
require(isValid, "Oracle rate invalid");
uint8 decimals = IRateOracle(oracle).decimals();
int256 currentFloatingRate = Utils.normalizeToWadSigned(currentRate, decimals);

uint256 newIndex = SwapFormulas.calculateNewRateIndex(index.indices[oracle], currentFloatingRate, elapsed);

index.indices[oracle] = newIndex;
index.lastUpdates[oracle] = currentTimestamp;

return newIndex;
}
```

`The helper it calls makes the problem even clearer. The update is simply` current rate x seconds elapsed `:`

```solidity
function calculateNewRateIndex(
uint256 currentFloatingIndex,
```

M-04 Latest rate rewrites stale floating accrual CONTINUED

int256 rateWad,
uint256 secondsElapsed
) public pure returns (uint256) {
int256 rateAdjustment = (rateWad * int256(secondsElapsed)) / int256(SECONDS_IN_YEAR);
int256 multiplier = WAD_INT + rateAdjustment;
if (multiplier <= 0) return MIN_INDEX;
uint256 result = Math.mulDiv(currentFloatingIndex, uint256(multiplier), WAD);
return result > MIN_INDEX ? result : MIN_INDEX;
}
```
That would only be safe if `IRateOracle.getRate()` returned the realized average rate for the entire interval since the previous update. It does not. The protocol's own interface describes it as a single spot rate:
`solidity
/// @notice For reference rate and risk premium oracles — provides a single spot rate
interface IRateOracle is IKairosOracle {
function getRate() external view returns (int256 rate, bool isValid);
}
`
The in-tree oracle implementations follow that model. For example, the Aave reference-rate adapter reads the current `variableBorrowRate` from the protocol data provider:
`solidity
function _getRate(address pool) internal view override returns (uint256) {
IAaveProtocolDataProvider aaveDataProvider = IAaveProtocolDataProvider(pool);
(, , , , , , uint256 variableBorrowRate, , , , , ) =
aaveDataProvider.getReserveData(borrowToken);
uint256 rateInWad = (variableBorrowRate * 1e18) / 1e27;
return rateInWad;
}
`
The Chainlink adapter does the same kind of thing with the latest answer:
``solidity
function getRate() external view returns (int256 price, bool isValid) {
(price, , isValid) = getPriceWithMetadata();
}

function getPriceWithMetadata() public view returns (int256 price, uint256 updatedAt, bool isValid) {
try feed.latestRoundData() returns (
uint80 roundId,
```

M-04 Latest rate rewrites stale floating accrual CONTINUED

int256 answer,
uint256,
uint256 _updatedAt,
uint80 answeredInRound
) {
...
int256 scaledPrice = int256(_scaleToWad(uint256(answer), feed.decimals()));
return (scaledPrice, _updatedAt, true);
} catch {
return (0, 0, false);
}
}
```
So the index is built from a spot observation, but the math treats that observation as if it had been true for the whole stale interval. That creates a path-dependent pricing error whenever updates are sparse and rates move in the meantime.
This is not a theoretical maintenance edge case. Anyone can trigger an update directly:
`solidity
function updateMarketRateIndex(bytes32 marketId) external {
Types.Market storage market = markets[marketId];
if (!market.exists) revert Errors.E300();
rateIndex.update(market.referenceRateOracle);
}
`
Normal user flows also call `rateIndex.update()` before they proceed. `buySwap()` is one example:
`solidity
// Update rate indices
rateIndex.update(referenceRateOracle);
`
The same update happens in LP deposits, LP withdrawals, settlement, liquidation, and transfer of swap ownership. In practice that means the first user who touches a stale market is the one who decides which latest spot rate gets smeared across the whole missing interval.
A realistic example is a market using the Aave reference-rate oracle. Alice opened a `BUY_FIXED` swap a month ago, so she benefits when the floating leg rises. For most of that month Aave's borrow rate sat around 5 percent. Then utilization spikes late on the last day and the live borrow rate jumps to 20 percent. No one has touched the Kairos market during the month, so the rate index is still stale. Bob sees the spike and calls settlement while
```

M-04 Latest rate rewrites stale floating accrual CONTINUED

the rate is elevated. `RateIndexLib.update()` takes that 20 percent spot rate and applies it to the entire month-long gap. Alice's floating-leg payout is now computed as if the market had spent the whole month near 20 percent, even though the rate only moved there at the end. If the move had gone the other way, the opposite side would have the same opportunity.

This contaminates more than one surface. The same stale-gap repricing feeds swap settlement, `getSwapNetAmount()`, LP share pricing, adapter NAV, and wrapped-position payouts. It is a protocol-level accounting error, not just a bad preview.

RECOMMENDATION

The floating index needs to be built from realized accrual, not from the latest spot rate stretched across an arbitrary stale interval. There are two reasonable ways to fix that. One is to require a cumulative or checkpointed oracle that can produce the realized average over any interval and derive index growth from that historical data. The other is to checkpoint the oracle rate often enough inside the protocol that each update only covers a period for which the sampled rate is actually intended to apply. What should not remain in place is the current model where a single late sample can rewrite an entire gap.

If the protocol keeps the existing spot-rate oracle interface, then every pricing path that depends on `rateIndex` should be treated as unsafe whenever `lastUpdates[oracle]` is meaningfully stale. At minimum, the system should bound allowable staleness tightly and refuse to settle or mark to market across large unchecked gaps. That would not make the accrual correct, but it would reduce the size of the mispricing window until a proper historical-rate design is in place.

RESOLUTION

Status: Partially resolved.

MEDIUM **M-05** RESOLVED

Redeem Can Strand Wrapped Escrow

CATEGORY	DoS
LOCATION	https://github.com/GuardianOrg/protocol-team1-1772465746057/blob/1e1150dea39d3b8c72eb06d1176e4b647fa4cfa6/packages/hardhat/contracts/SwapPositionWrapper.sol#L230
REVIEW	Main Review 1

DESCRIPTION

For wrapped swaps, swap.userAddress in SwapCore is the SwapPositionWrapper contract. If settlement transfer fails, Utils._transferOrEscrow places funds into escrowedCollateral\[swapId\], but the swap is still marked as settled.
SwapPositionWrapper.redeem() only checks that the swap is settled, and _creditIfNeeded() only reads settlementPayouts\[swapId\] rather than escrowedCollateral\[swapId\]. On the escrow path, settlementPayouts is zero, so redeemable\[tokenId\] remains zero. The holder can still call redeem(), which burns the NFT and deletes tokenIdToSwapId\[tokenId\]. After that, SwapPositionWrapper.claimEscrow(tokenId) can no longer resolve the swapId, and the holder cannot call SwapCore.claimEscrow(swapId) directly because only swap.userAddress is authorized. This can permanently strand escrowed payout.
Because SwapCore.makePayment() is permissionless, settlement may be executed by a third party; a wrapped holder may only observe a settled position and call redeem() without realizing payout is currently parked in escrow, at which point the mapping deletion and burn make it permanently unclaimable.

RECOMMENDATION

In SwapPositionWrapper.redeem, reconcile escrow before state deletion and burn: either automatically call swapCore.claimEscrow(swapId) and add the result to redeemable\[tokenId\], or revert if swapCore.escrowedCollateral(swapId) \> 0 to force the holder to call claimEscrow first.

RESOLUTION

Status: Resolved.

MEDIUM M-06

ACKNOWLEDGED

BaseSwapRate getRate() Ignores Live Inputs

CATEGORY	Oracle
LOCATION	https://github.com/GuardianOrg/protocol-team1-1772465746057/blob/1e1150dea39d3b8c72eb06d1176e4b647fa4cfa6/packages/hardhat/contracts/oracles/BaseSwapRate.sol#L122
REVIEW	Main Review 1

DESCRIPTION

BaseSwapRate exposes setBaseSwapRate(), setReferenceRate(), and baseSwapRateWeightedAverage() as live or configurable rate inputs, but its runtime read path does not use them. getRate() calls latestRoundData(), and latestRoundData() returns baseSwapRateWadMock(). In baseSwapRateWadMock(), min == max == 0.05e18, so the returned rate is effectively fixed at 5%.
As a result, any integration that consumes BaseSwapRate through getRate()/latestRoundData() receives the mock constant, even after owner updates or weighted-average changes.

RECOMMENDATION

Wire latestRoundData()/getRate() to the intended live source (_baseSwapRate and/or baseSwapRateWeightedAverage()).

RESOLUTION

Status: Acknowledged.

MEDIUM **M-07** ACKNOWLEDGED

Escrow claim cannot bypass blocked owners

CATEGORY	Unexpected Behavior
LOCATION	SwapCore.sol
REVIEW	Main Review 1

DESCRIPTION

`SwapCore.claimEscrow` assumes an escrowed payout becomes recoverable as soon as an authorized caller retries the claim. That is not always true. The function always sends the escrowed tokens back to `swap.userAddress`, which is the original swap owner recorded in `SwapCore` and it has no way to redirect the payout to a different receiver when that owner is itself the reason the original transfer failed.

```plain text
function claimEscrow(bytes32 swapId) external nonReentrant returns (uint256 amount) {
Types.SwapPosition storage swap = swapPositions[swapId];
address recipient = swap.userAddress;
if (!_isSenderAuthorized(recipient)) revert Errors.E206();
amount = escrowedCollateral[swapId];
if (amount == 0) revert Errors.E400();
escrowedCollateral[swapId] = 0;
IERC20(swapToken).safeTransfer(recipient, amount);
}
```

That behavior is harmless when swap.userAddress is a normal EOA that was only temporarily unable to receive tokens. It breaks down when the recorded owner is an intermediary contract such as KairosBuyerAdapter or SwapPositionWrapper. In those integrations, settlement already routes payout to the intermediary because it owns the swap. If that transfer fails and funds are moved into escrow, the later recovery attempt goes through the same recipient again. The adapter and wrapper helper functions simply call back into swapCore.claimEscrow`, so they do not change the recipient or bypass the blocked address. If the intermediary remains unable to receive the token, escrow recovery keeps reverting and the payout remains stuck even though the beneficiary is known and an authorized caller is available.

RECOMMENDATION

M-07 Escrow claim cannot bypass blocked owners CONTINUED

Do not force escrow claims to repay only `swap.userAddress` when that address is a custody intermediary. The safest fix is to store enough beneficiary information to let escrow be redirected to the current economic owner or to an explicitly authorized alternate recipient during claim. If the protocol wants to preserve a strict no-redirection model for direct users, add a separate escrow-claim path for wrapper and adapter-owned swaps that forwards the recovered payout to the beneficiary expected by the integration instead of retrying the same blocked recipient.

RESOLUTION

Status: Acknowledged.

MEDIUM **M-08** RESOLVED

Rate index compounds with update cadence

CATEGORY	Logical Error
LOCATION	RateIndexLib.sol, SwapFormulas.sol, SwapCore.sol
REVIEW	Main Review 1

DESCRIPTION

The variable-rate side of the swap is supposed to accrue like simple APR interest, but that is not what happens once the index is updated more than once.
Each call to `RateIndexLib.update()` takes the current stored index and applies the return for the elapsed period on top of it:

```solidity
function update(RateIndex storage index, address oracle) internal returns (uint256) {
uint256 currentTimestamp = block.timestamp;
uint256 elapsed = currentTimestamp - index.lastUpdates[oracle];
...
uint256 newIndex = SwapFormulas.calculateNewRateIndex(index.indices[oracle],
currentFloatingRate, elapsed);
index.indices[oracle] = newIndex;
index.lastUpdates[oracle] = currentTimestamp;
return newIndex;
}
```

The helper itself looks like a simple-interest step:

```solidity
function calculateNewRateIndex(
uint256 currentFloatingIndex,
int256 rateWad,
uint256 secondsElapsed
) public pure returns (uint256) {
int256 rateAdjustment = (rateWad * int256(secondsElapsed)) / int256(SECONDS_IN_YEAR);
int256 multiplier = WAD_INT + rateAdjustment;
if (multiplier <= 0) return MIN_INDEX;
uint256 result = Math.mulDiv(currentFloatingIndex, uint256(multiplier), WAD);
return result > MIN_INDEX ? result : MIN_INDEX;
}
```

M-08 Rate index compounds with update cadence CONTINUED

That only stays simple if it is used once for the whole period. If the same formula is used multiple times, each later update is applied on top of an index that already grew on the earlier update. In other words, the result compounds.
A simple example shows the difference. If the reference rate stays at 10 percent for a full year, one update at the end gives 10 percent accrual. But if the index is updated halfway through the year and then again at the end, the result is 10.25 percent. Nothing about the rate changed. The only thing that changed was how often the index was refreshed.
That means two swaps with the same start time, end time, and oracle rate can settle to different values just because one path had more checkpoints. This is separate from the stale-rate issue. There, the problem is that a late spot rate gets applied to an old interval. Here, even a flat rate still gives different results when the update cadence changes.
This directly affects real positions because `SwapCore` stores the rate index when the swap is opened:

```solidity
Types.SwapPosition memory buyerSwapPosition = Types.SwapPosition({
...
entryFloatingIndex: rateIndex.getIndex(referenceRateOracle),
...
});
```

Later, the protocol uses the gap between that entry index and the current one to work out the variable-rate payment:

```solidity
int256 floatingPaymentRate = SwapFormulas.deriveRateFromIndex(
swap.entryFloatingIndex,
_getFreshIndex(market.referenceRateOracle),
timeElapsedSinceSwapOpened
);
```

So this affects `getSwapNetAmount`, settlement, liquidation checks, and LP pricing.
The impact is wider because the index is shared per oracle, not per swap:

```solidity
/**
* @dev Rate indices are shared across all markets using the same reference rate oracle,
* ensuring more frequent updates and consistent accrual across markets
*/
```

Any update to that oracle changes the outcome for every open swap that uses it, including the public refresh path:

M-08 Rate index compounds with update cadence CONTINUED

```solidity
function updateMarketRateIndex(bytes32 marketId) external {
Types.Market storage market = markets[marketId];
if (!market.exists) revert Errors.E300();
rateIndex.update(market.referenceRateOracle);
}
```

So value does not depend only on the rate path. It also depends on how often the protocol was touched. A user on the side that benefits when the variable rate goes their way can improve their result by forcing more checkpoints, and activity in another market can also change an open position if both markets share the same oracle.
That is the core issue: the protocol presents this side of the swap like simple APR accrual, but in practice more updates can increase the variable-rate payout even when the rate itself did not move.

RECOMMENDATION

The protocol should choose one model and implement it consistently.
If this side of the swap is supposed to use simple APR, then the protocol should stop rolling sub-period returns into a growing index. A safer design is to store cumulative `rate * time` and settle from that directly.
If compounding is actually intended, then the docs, pricing assumptions, collateral logic, and oracle requirements should all say that clearly.
Either way, checkpoint frequency should not change value when the underlying rate path is unchanged.

RESOLUTION

Status: Resolved.

MEDIUM M-09 RESOLVED

LP full withdraw skips forcedeallocation penalty

CATEGORY Logical Error
LOCATION KairosMarketAdapter.sol
REVIEW Main Review 1

DESCRIPTION

KairosMarketAdapter.deallocate() exposes a custom fullAmount mode that is not compatible with Morpho Vault V2's force-deallocation penalty model.
In normal Morpho flow, forceDeallocate() is permissionless and computes the penalty from the caller-supplied assets amount:
``plain text
bytes32[] memory ids = deallocateInternal(adapter, data, assets);
uint256 penaltyAssets = assets.mulDivUp(forceDeallocatePenalty[adapter], WAD);
uint256 penaltyShares = withdraw(penaltyAssets, address(this), onBehalf);
`
The LP adapter adds a second deallocation mode where the caller can encode fullAmount = true and redeem all LP shares while passing assets = 0 :
``plain text
if (!fullAmount && assets == 0) revert ZeroAmount();

uint256 withdrawn = swapCore.withdrawCollateral(marketId, assets, fullAmount, address(this), minCollateralOut);

if (fullAmount) {
if (withdrawn > 0) {
asset.safeTransfer(parentVault, withdrawn);
}
}

change = -int256(withdrawn);
``
That means any external user can call the vault's permissionless forceDeallocate() with assets = 0 and adapter data that sets fullAmount = true . The adapter will still unwind the entire unlocked LP position and push the realized collateral back to the vault, but the vault computes the penalty on zero assets, so penaltyAssets and penaltyShares` are also zero.

M-09 LP full withdraw skips forcedeallocation penalty CONTINUED

As a concrete example, imagine Alice deposited into a Morpho vault that uses `KairosMarketAdapter` as its liquidity adapter and governance configured a 10 percent force-deallocation penalty specifically to stop opportunistic exits from disrupting the LP strategy. Bob does not need allocator privileges. He can call `forceDeallocate(adapter, abi.encode(marketId, 0, true), 0, Alice)` as soon as the market has any unlocked liquidity. The adapter exits the LP position, the collateral returns to the vault and Alice loses no shares because the penalty is computed from zero. Bob has forced the vault out of the strategy for free, even though governance explicitly configured a penalty to make that behavior costly.

RECOMMENDATION

Do not expose this custom full-withdraw path through the standard Morpho `deallocate()` hook. The safest fix is to remove `fullAmount` support from `KairosMarketAdapter.deallocate()` entirely and require all deallocations to use the vault's explicit `assets` amount so Morpho's penalty logic remains meaningful. If the team still wants an all-shares unwind path, it should be implemented as a separate privileged operation that is not reachable through permissionless `forceDeallocate()`. At minimum, the adapter should reject `fullAmount` when the call context corresponds to force deallocation, or it should require a nonzero penalty basis that matches the actual amount being unwound.

RESOLUTION

Status: Resolved.

MEDIUM **M-10** ACKNOWLEDGED

AbstractReferenceRate fabricates freshness

CATEGORY	Oracle
LOCATION	`AbstractReferenceRate.sol`
REVIEW	Main Review 1

DESCRIPTION

`AbstractReferenceRate` makes every child oracle look fresh and valid even when it has no basis for doing so. It always stamps the answer with the current block time and always returns `isValid = true` as long as `_getRate(pool)` does not revert:

```plain text
function latestRoundData()
public
view
returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80 answeredInRound)
{
uint256 rate = _getRate(pool);
return (0, int256(rate), block.timestamp, block.timestamp, 0);
}

function getRate() external view virtual returns (int256 rate, bool isValid) {
(, int256 answer, , , ) = latestRoundData();
return (answer, true);
}
```

`The rest of the protocol trusts that metadata.` Utils.getOracleRate() `,` RateIndexLib.update() `and` TWAPOracle `all use` getRate() `as a health check.`
`This matters for` ReferenceRateCompound `directly. It derives its rate from Compound's` getUtilization() `and` getBorrowRate(utilization) `, but Compound notes that` getUtilization() `does not accrue interest first. Even so, the wrapper still reports` updatedAt = block.timestamp `and` isValid = true `, so downstream code cannot tell a fresh reading from a stale one.`
`The same problem applies to any child that returns a bad value without reverting. A zero or degraded rate is upgraded into a healthy-looking` (rate, true) `response with a fresh timestamp. That weakens market creation checks, TWAP wrapping and any runtime path that assumes` isValid` actually means the oracle is healthy.

M-10 AbstractReferenceRate fabricates freshness CONTINUED

RECOMMENDATION

Do not fabricate oracle health in the base class. Child adapters should return source-level metadata such as `(rate, updatedAt, isValid)`, or the base should revert when freshness and validity cannot be established. At minimum, `AbstractReferenceRate` should stop using `block.timestamp` as the source update time and stop returning `true` by default for every non-reverting child call.

RESOLUTION

Status: Acknowledged.

MEDIUM M-11

ACKNOWLEDGED

TWAP history corrupts on source rotation

CATEGORY	Unexpected Behavior
LOCATION	`TWARateOracle.sol`, `ReferenceRateMorpho.sol`
REVIEW	Main Review 1

DESCRIPTION

`TWAPOracle` exposes `setReferenceSpotRateAddress()` so the owner can replace the underlying spot-rate source, but the contract does not reset any of the historical state that gives the wrapper its time-averaged meaning. The call only updates `referenceSpotRateAddress` and emits an event. It leaves `currentRateIndex`, `lastUpdate`, `currentCheckpointIndex`, `checkpointCount` and all stored checkpoints untouched.

That breaks the accounting model as soon as the source changes. Both `updateTWAPRate()` and `getRateWindow()` assume that the full interval since `lastUpdate` belongs to the current source:

After a rotation, that assumption is false. The stored index and checkpoints still encode the old source, but the newly selected source is used to extend the index and to answer future window queries. The next `updateTWAPRate()` can therefore apply the new source's current rate across time that actually belonged to the old source and `getRateWindow()` can report a blended average made from two unrelated histories while still returning `isValid = true`.

This is not just a privileged-actor threat. A normal maintenance flow can trigger it. If operators deploy a corrected Morpho spot-rate adapter and migrate the wrapper to it, the wrapper does not start a clean new average. It immediately begins serving economically nonsensical data derived from both the old and new sources. Any market or downstream oracle that trusts this wrapper can price, settle, or mark positions off a corrupted rate for up to the full averaging window.

This is relevant to the checked-in deployment topology. The deployment flow already creates Morpho-backed `TWAPOracle` instances whose constructor source is a deployed `ReferenceRateMorpho`. The deployment scripts do not currently call `setReferenceSpotRateAddress()` themselves, so the bug is not triggered during initial deployment. The risk appears later, when operators use the exposed admin function to rotate a live wrapper to a replacement Morpho source during maintenance, migration, or recovery.

RECOMMENDATION

M-11 TWAP history corrupts on source rotation CONTINUED

Treat a source rotation as a new oracle history. `setReferenceSpotRateAddress()` should either revert unless the wrapper is freshly deployed, or it should fully reset `currentRateIndex`, `lastUpdate` and the checkpoint ring buffer before initializing the replacement source. If continuity across rotations is required, add explicit migration logic that proves the old and new sources are economically identical and safely transforms the historical state instead of silently blending unrelated histories.

RESOLUTION

Status: Acknowledged.

MEDIUM M-12

ACKNOWLEDGED

US10YR Pyth trades outside feed hours

CATEGORY	Unexpected Behavior
LOCATION	`PythAdapter.sol`
REVIEW	Main Review 1

DESCRIPTION

The deployed `US10YR` `Pyth` adapter only enforces an age check. That would be fine for a truly continuous feed, but `Pyth` documents its `Rates` feeds as session-based and notes that they are available only on weekdays from `8:00 AM` to `5:00 PM ET`. The Base deployment still gives `PythAdapter_US10YR` a four day `maxAge`.

Do note that the deployment script includes a market deployment path that picks up `PythAdapter_US10YR` and uses it as the `referenceRateOracle` when the adapter is present.

That means a quote from the previous Treasury session can remain acceptable far beyond the feed's active market hours. If a `Kairos` market uses that adapter, the protocol can keep allowing entries, settlements, liquidations and LP repricing overnight or across weekend and holiday gaps using the last in-session `US10YR` value.

The oracle can still be fresh enough to satisfy `getPriceNoOlderThan`, but the quote is not current enough to serve as a fair executable market price because the underlying feed is outside its documented session. Once a market is wired to this adapter, the problem reaches trading, settlement, liquidation and LP repricing directly, which is why it is more than a low-severity ops caveat.

RECOMMENDATION

Do not treat `maxAge` as a complete availability check for session-based `Pyth` `Rates` feeds. Add a market-hours guard for these feeds and block executable actions outside the supported session. At minimum, tighten the `US10YR` adapter's `maxAge` so a previous-session quote cannot remain tradable through overnight or weekend closure windows.

RESOLUTION

Status: Acknowledged.

MEDIUM M-13

RESOLVED

L2 Sequencer Check Missing

CATEGORY	Oracle
LOCATION	`ChainlinkAdapter.sol`
REVIEW	Main Review 1

DESCRIPTION

ChainlinkAdapter validates latestRoundData() responses and enforces a maxAge staleness check but does not check the L2 sequencer uptime feed or enforce a post-recovery grace period. If this adapter is used on Base, the protocol can accept stale or unsafe prices during sequencer downtime or immediately after recovery.

RECOMMENDATION

Add an L2 sequencer uptime and grace period check to ChainlinkAdapter.

RESOLUTION

Status: Resolved.

MEDIUM **M-14** RESOLVED

Shared LP vesting blocks unrelated withdrawals

CATEGORY	DoS
LOCATION	`KairosMarketAdapter.sol`
REVIEW	Main Review 1

DESCRIPTION

Kairos prevents an LP address from depositing and withdrawing in the same block by recording `lastDepositBlock` on the LP position. On its own, that is a reasonable anti-flash-loan guard. The problem starts once `KairosMarketAdapter` is used as a shared Morpho liquidity adapter.

When `VaultV2` auto-allocates deposits, the adapter supplies collateral to Kairos from its own address. That means every vault user shares the same LP identity inside Kairos. If a withdrawal later needs funds back from the adapter, the adapter exits through that same address. Because the lock is keyed by LP address rather than by vault user, a deposit by one account updates `lastDepositBlock` for the entire pooled position and can make another account's withdrawal revert with `E413` in the same block.

That turns a local anti-flash-loan rule into a cross-user griefing vector. An attacker only needs to watch for withdrawals that depend on adapter liquidity and front-run them with a dust deposit into the vault. The deposit refreshes the adapter's LP lock, the withdrawal tries to deallocate in the same block and the exit fails. Repeating that pattern can keep affected withdrawals failing block after block for very little cost. No funds are stolen directly, but users can be denied timely exits from an otherwise liquid position.

RECOMMENDATION

Avoid wiring this adapter into a flow that auto-allocates every user deposit directly into Kairos. Keeping an idle buffer in the vault, or moving allocation to an asynchronous keeper or allocator step, avoids refreshing a shared LP lock immediately before user withdrawals.

If synchronous allocation must remain, the vesting lock in Kairos needs to be scoped to something other than the adapter's single LP address, or the integration needs a withdrawal path that does not rely on unwinding a position that may have been touched earlier in the same block.

M-14 Shared LP vesting blocks unrelated withdrawals CONTINUED

RESOLUTION

Status: Resolved.

MEDIUM M-15

RESOLVED

Utilization Fees Can Be Reduced by Splitting

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/protocol-team1-1772465746057/blob/1e1150dea39d3b8c72eb06d1176e4b647fa4cfa6/packages/hardhat/contracts/lib/SwapFormulas.sol#L249
REVIEW	Main Review 1

DESCRIPTION

SwapFormulas.calculateUtilizationFee() prices utilization using the current trade's size relative to the currently available liquidity, i.e. tradeSize / availableLiquidity, rather than the full utilization transition caused by the order. SwapCore.buySwap() uses this value directly when determining utilizationFee, which is then incorporated into swap pricing and buyer collateral.

Because the fee is assessed independently for each swap against the liquidity available at that moment, a user can split one large order into smaller sequential swaps and pay materially less total utilization fee than if the same total notional were executed as a single trade. This weakens the intended economic effect of the kinked utilization curve, especially at higher utilization, where larger liquidity consumption is supposed to incur disproportionately higher fees.

For example, with available liquidity of 1000 and a linear utilization-fee slope of 10%, a single swap of 100 pays a utilization-fee contribution of 1.0. Splitting the same order into two sequential swaps of 50 pays about 0.25 and 0.2632, for a total of 0.5132. The same final exposure therefore reduces utilization fees by roughly 48.7% simply by splitting the order.

RECOMMENDATION

Price utilization based on the full utilization movement caused by the order, rather than only the local tradeSize / availableLiquidity ratio of each individual swap.

RESOLUTION

Status: Resolved.

MEDIUM M-16

RESOLVED

Zero Share Supply Can Trigger False Pool Reset

CATEGORY	Logical Error
LOCATION	https://github.com/GuardianOrg/protocol-team1-1772465746057/blob/1e1150dea39d3b8c72eb06d1176e4b647fa4cfa6/packages/hardhat/contracts/SwapCore.sol#L409
REVIEW	Main Review 1

DESCRIPTION

SwapCore.withdrawCollateral() can reduce a market's LP share supply to zero even though the pool is not actually empty. That alone is not a problem, since remaining collateral may still be required to support existing swap state. The issue is that supplyCollateral() later uses pool.totalShares == 0 as the trigger for 1:1 bootstrap minting, treating the market as if it were a fresh pool.

As a result, a new depositor can enter after the share supply has been reduced to zero and be minted shares as though they are the first LP, even though residual pool collateral or unresolved swap state may still remain. If that remaining state leaves value in the pool, the new depositor can capture value they did not properly pay for because zero share supply was treated as equivalent to an empty pool.

RECOMMENDATION

Do not use pool.totalShares == 0 alone as the condition for bootstrap minting. The protocol should ensure that 1:1 bootstrap minting is only possible when the pool is actually empty.

RESOLUTION

Status: Resolved.

MEDIUM **M-17** RESOLVED

Uninitialized buckets can return valid rates

CATEGORY	Validation
LOCATION	`FixedRatePricingOracle.sol`
REVIEW	Main Review 1

DESCRIPTION

`getBaseRate` does not verify that the buckets it reads were ever posted through `postForecasts`. If the global freshness check passes, the function can return an exact bucket that is not part of the active tenor set anymore, flat-extrapolate from an uninitialized edge bucket or interpolate against a zero-valued bucket with `updatedAt == 0`.

This creates silent bad pricing instead of a clean failure. In local reproduction, a curve with only the 365 day bucket posted returned `0` for a 90 day query with `isValid = true`, and a curve with only the 60 day bucket posted returned a non-zero interpolated rate against an uninitialized upper endpoint. The same exact-match shortcut also keeps serving forecasts for buckets that governance removed with `setTenorBuckets`.

RECOMMENDATION

Reject any quote unless every bucket used to produce it is both configured and initialized. For exact matches, require that the tenor is part of `tenorBuckets` and has `updatedAt != 0`. For interpolation and extrapolation, require initialized endpoints and return `false` if either side is missing.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/610 https://github.com/kairosswap/protocol/pull/616

MEDIUM **M-18** RESOLVED

Stale Buckets Can Pass Validation

CATEGORY	Oracle
LOCATION	https://github.com/GuardianOrg/protocol-team1-1772465746057/blob/13595591dbf64c6b24d55613bc1d6de5b900e0c9/packages/hardhat/contracts/oracles/FixedRatePricingOracle.sol#L109
REVIEW	Main Review 1

DESCRIPTION

FixedRatePricingOracle.getBaseRate() checks freshness using only the most recently updated tenor bucket in the curve, rather than the specific tenor bucket or bucket pair used for the returned quote. Forecasts are likely intended to be updated as a full tenor curve in a single batch, but the contract does not enforce that. As a result, some tenor buckets can remain stale while another bucket is refreshed, and queries for the stale tenor can still return isValid = true.

This is important because SwapCore trusts the oracle's validity flag and uses the returned base rate directly for swap pricing, collateral calculations, LP mark-to-market pricing, and early-exit settlement. For example, if 7d and 30d are configured and only 30d gets refreshed, a later query for 7d can still return the stale 7d rate as valid. The protocol therefore continues operating with outdated pricing inputs instead of cleanly rejecting them.

RECOMMENDATION

Validate freshness for the actual bucket or bucket pair used to produce the quote. Exact-match queries should require the exact tenor to be fresh, interpolated quotes should require both surrounding buckets to be fresh, and extrapolated quotes should require the endpoint bucket used to be fresh.

Alternatively, add checks in postForecasts() to enforce full-curve updates against the configured tenorBuckets, so partial refreshes cannot leave stale or uninitialized tenor points implicitly valid.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/610 https://github.com/kairosswap/protocol/pull/61 (https://github.com/kairosswap/protocol/pull/610) 6

LOW **L-01** RESOLVED

Bucket saturation inflates LP pricing gas

CATEGORY	Gas Griefing
LOCATION	Utils.sol
REVIEW	Main Review 1

DESCRIPTION

LP entry, LP exit and LP-adapter valuation all depend on `getPoolSharePrice`, but that function scales with the configured bucket count rather than with a bounded constant-time accounting model. The market validator allows up to 1000 buckets, and the pricing path iterates over every bucket on each call. For each populated bucket it may also perform an external `getBaseRate` call for a bucket-specific remaining tenor.

```plain text
if (params.numBuckets == 0 || params.numBuckets > 1000) revert Errors.E703();
```

```plain text
for (uint256 i = 0; i < market.numBuckets; i++) {
Types.Bucket storage bucket = buckets[i];
if (bucket.lpNotional == 0) {
continue;
}
uint256 remainingTenor = market.swapTerm > timeElapsed ? market.swapTerm - timeElapsed : 0;
if (remainingTenor > 0) {
projectedRate = getBaseRateOracle(market.baseSwapRateOracle, remainingTenor);
}
totalLpPnL += calculateBucketUnrealizedPnL(bucket, market, projectedRate, oracleIndex);
}
```

Because bucket assignment is deterministic and time-based, an attacker can keep one live swap in each bucket by opening small positions across successive intervals. Once enough buckets are populated, every call to `supplyCollateral`, `withdrawCollateral`, and `KairosMarketAdapter.realAssets` inherits the full loop and the full set of oracle calls.

A standalone Forge POC was used to compare two otherwise identical 1000-bucket markets. In the sparse case with one populated bucket, `getPoolSharePrice` consumed 2,415,994 gas. In the dense case with one live swap in every bucket, the same call consumed 14,031,500 gas.

L-01 Bucket saturation inflates LP pricing gas CONTINUED

Using a shared gas cap of 8,223,747, the sparse market still priced successfully while the saturated market failed. That is a material attacker-controllable gas spike on a critical path.
This does not yet prove a hard block-limit denial of service on current EVM block gas limits, so the impact is better characterized as gas griefing and degraded composability rather than outright uncallability. The risk is still real because these pricing calls sit on the main transaction path for LP actions and downstream adapter valuation, and the gas burden can become large enough to make integrations brittle or uneconomical.

RECOMMENDATION

If these gas costs are not acceptable, consider not putting an unbounded bucket walk with external oracle calls on a critical onchain pricing path. The cleanest fix is to replace the current share-pricing model with incrementally maintained accounting that does not need to scan every bucket during user actions. If the bucket model must stay, reduce the maximum bucket count to a conservative gas-safe value and cache or precompute the required tenor rates so LP-facing operations do not suffer attacker-controlled gas inflation.

RESOLUTION

Status: Resolved.

LOW **L-02** RESOLVED

Adapters stale-price realAssets on oracle fail

CATEGORY	Oracle
LOCATION	KairosBuyerAdapter.sol, KairosMarketAdapter.sol
REVIEW	Main Review 1

DESCRIPTION

Both Kairos Morpho adapters can continue returning stale marks exactly when the underlying Kairos market has stopped accepting fresh pricing updates. The root cause is that `SwapCore._getFreshIndex` catches oracle failures and silently falls back to the stored rate index instead of propagating the error.

```plain text
function _getFreshIndex(address oracle) private view returns (uint256 idx) {
idx = rateIndex.getIndex(oracle);
uint256 elapsed = block.timestamp - rateIndex.getLastUpdate(oracle);
if (elapsed > 0) {
try IRateOracle(oracle).getRate() returns (int256 r, bool ok) {
if (ok) {
int256 rateWad = Utils.normalizeToWadSigned(r, IRateOracle(oracle).decimals());
idx = SwapFormulas.calculateNewRateIndex(idx, rateWad, elapsed);
}
} catch {}
}
}
```

KairosBuyerAdapter.realAssets `relies on` getSwapNetAmount`, and` KairosMarketAdapter.realAssets `relies on` getPoolSharePrice`. Both of those views ultimately use` _getFreshIndex`, so they keep returning values based on the last stored index when the oracle is stale, invalid, or reverting. That behavior diverges from the actual state-changing Kairos paths, which use` rateIndex.update `and revert on invalid oracle data.`

```plain text
(int256 currentRate, bool isValid) = IRateOracle(oracle).getRate();
require(isValid, "Oracle rate invalid");
```

L-02 Adapters stale-price realAssets on oracle fail CONTINUED

As a result, an external vault can continue using adapter `realAssets` for deposits, withdrawals, or share conversion while the underlying market itself has already halted on oracle failure. This creates a window where vault users trade against stale NAV precisely during oracle incidents, which is when pricing error is most dangerous.

RECOMMENDATION

Do not let adapter NAV continue updating from stale state once the oracle has failed. The simplest fix is to make the valuation path used by `realAssets` revert under the same conditions as the state-changing Kairos path, instead of silently reusing the last stored index. If a stale fallback is intentionally kept for monitoring or read-only UX, surface that condition explicitly and make sure parent vault integrations can detect it and stop deposits, withdrawals, and share pricing until the oracle is healthy again.

RESOLUTION

Status: Resolved.

LOW L-03 RESOLVED

Vault V2 cannot recover adapter escrow

CATEGORY	Unexpected Behavior
LOCATION	KairosBuyerAdapter.sol
REVIEW	Main Review 1

DESCRIPTION

When a buyer-side settlement or liquidation transfer fails, `SwapCore` does not keep the payout in `settlementPayouts`. Instead it escrows the amount under `escrowedCollateral[swapId]` and zeroes the released payout value. That fallback is correct at the core protocol level, but `KairosBuyerAdapter` is not wired to recover it through the normal Vault V2 adapter interface.

The adapter's settled deallocation path only checks `settlementPayouts`, forwards that amount to the vault, and then removes the swap from internal tracking. If the settlement transfer was escrowed, `settlementPayouts` is zero, so the adapter reports no recovered funds, returns `change = 0`, and can still prune the swap even though the real proceeds are still sitting inside `SwapCore`.

```solidity
if (settled) {
uint256 payout = swapCore.settlementPayouts(sid);
if (payout > 0) asset.safeTransfer(parentVault, payout);
_removeSwapId(mid, sid);
change = -int256(payout);
return (ids, change);
}
```

The adapter does expose a dedicated `claimEscrow` function, but that function is `onlyParentVault`. The problem is that standard Morpho Vault V2 integrations do not route arbitrary adapter-specific helper calls. The normal adapter surface is `allocate`, `deallocate`, and `realAssets`. Since `SwapCore.claimEscrow` pays only the recorded swap owner, external users cannot bypass the adapter and recover the escrow directly either. For adapter-owned swaps, the adapter remains the only effective owner, yet the standard vault interface does not provide a path to make that recovery happen.

```solidity
function claimEscrow(bytes32 swapId) external onlyParentVault returns (uint256 claimed) {
claimed = swapCore.claimEscrow(swapId);
if (claimed > 0) {
```

L-03 Vault V2 cannot recover adapter escrow CONTINUED

asset.safeTransfer(parentVault, claimed);
}
}
```

Consider a realistic Morpho Vault V2 deployment where Alice deposits USDT into the vault and Bob, acting as allocator, uses KairosBuyerAdapter to buy a Kairos swap on the vault's behalf. The important point is not that USDT omits the standard boolean return value on transfer, because the escrow helper already treats empty return data as success. The relevant risk is that USDT retains administrative controls such as pause and blacklist behavior. When the swap later settles in Alice's favor, SwapCore tries to return the buyer proceeds to the adapter because the adapter is the recorded swap owner. If the USDT transfer reverts because the token is paused or because SwapCore has been blacklisted as a sender, the proceeds are moved into escrowedCollateral[swapId] instead. Bob then performs the normal vault-side recovery step and calls deallocate for the settled swap. The adapter sees settlementPayouts[swapId] == 0, forwards nothing back to the vault, returns change = 0, and removes the swap from its tracking set. At that point Alice's funds still exist inside SwapCore, but neither Alice nor Bob can retrieve them through the standard Vault V2 workflow even though the vault was the economic owner of the position.

More broadly, the bug does not depend on USDT specifically or on any claim that escrow is a common path. The protocol intentionally implements escrow as a supported fallback for failed buyer transfers. Once that fallback exists, every integration that takes custody of swap ownership also needs a standard way to recover escrowed proceeds and reflect that recovery in vault accounting. Otherwise the system is explicitly acknowledging that funds may be redirected into escrow while leaving one of its intended integrations unable to complete the recovery flow.

The result is that escrowed buyer proceeds can become practically stranded in a standard Vault V2 deployment. Even if a custom vault or privileged wrapper later adds a bespoke path to call claimEscrow, the adapter still does not report the recovered amount through the ordinary deallocate` accounting contract, so vault allocation and cap state remain stale after recovery. This is therefore both a recoverability failure and an integration-accounting failure.

### — RECOMMENDATION
```

L-03 Vault V2 cannot recover adapter escrow CONTINUED

Escrow recovery needs to be exposed through the standard adapter interface. A straightforward fix is to add a dedicated `deallocate` action that claims escrow, forwards the recovered funds to the vault, and returns the appropriate negative `change`. The adapter should also avoid removing settled swaps from tracking while either `settlementPayouts` or `escrowedCollateral` remains nonzero. If the integration intends to support escrow at all, recovery cannot depend on an adapter-specific helper that standard Vault V2 does not know how to call.

RESOLUTION

Status: Resolved.

LOW **L-04** RESOLVED

Buyer adapter realAssets can DoS valuation

CATEGORY	DoS
LOCATION	BuyerAdapter.sol
REVIEW	Main Review 1

DESCRIPTION

`KairosBuyerAdapter.realAssets()` performs a full scan over every tracked buyer swap and recomputes mark-to-market value one position at a time. The function first loops over all active markets, then over all tracked swap ids in each market, then calls `swapCore.getSwapNetAmount()` for every unsettled swap. That means the gas cost of vault valuation grows linearly with the number of live buyer positions held through the adapter.

```solidity
function realAssets() external view override returns (uint256 assets) {
assets = asset.balanceOf(address(this));

uint256 length = marketIds.length;
for (uint256 i = 0; i < length; i++) {
bytes32[] memory swapIds = _marketSwapIds[marketIds[i]];
for (uint256 j = 0; j < swapIds.length; j++) {
(, , bool settled, , uint256 collateralBalance, , , , , , , ) =
swapCore.swapPositions(swapIds[j]);

if (!settled) {
(uint256 netAmount, uint8 netRecipient) = swapCore.getSwapNetAmount(swapIds[j]);
...
}
}
}
}
```

This is not a cheap inner call either. getSwapNetAmount() derives a fresh floating-leg mark using the current simulated oracle index, so each per-swap iteration performs additional pricing work instead of reading a cached balance. As the vault accumulates more fragmented buyer exposure, realAssets() becomes more and more expensive.

Morpho uses adapter realAssets()` on valuation-critical paths. That means deposits, withdrawals, redemptions, previews, accrual, and cap logic can all inherit this unbounded

L-04 Buyer adapter realAssets can DoS valuation CONTINUED

per-swap walk. An allocator can naturally create many live positions over time, and a malicious or careless strategy can intentionally fragment exposure into many small swaps. Once enough swaps exist, routine vault operations can become prohibitively expensive or revert under gas limits. This is a liveness issue on the vault integration path, not just an inefficient view.

Consider a realistic deployment where Alice deposits into a Morpho vault that uses `KairosBuyerAdapter` for rate exposure, and Bob is the vault's approved allocator. Bob does not need to do anything exotic to trigger the problem. He can simply implement a rolling strategy that opens many small swaps instead of a few large ones so that entries mature at different times or target slightly different rate views. After weeks or months of normal operation, the adapter may be tracking a large set of live buyer positions across one or several Kairos markets. At that point, when Alice calls `previewWithdraw`, `withdraw`, or even when a new depositor checks `previewDeposit`, the vault has to walk all of those swaps through `realAssets()`. The valuation work scales with historical position fragmentation rather than with a bounded adapter state, so ordinary vault actions can become too expensive or revert even though no single Kairos position is individually problematic.

This issue is distinct from the existing LP pricing gas finding. That one scales with configured bucket count in LP share pricing. This one scales with the raw number of live buyer swaps held by the adapter, so even a single-market deployment can still degrade badly if the strategy opens enough positions.

RECOMMENDATION

Do not make vault valuation scale with live swap count. The safest fix is to maintain aggregate buyer valuation state on write and have `realAssets()` read a bounded amount of cached data rather than scanning every position. If exact real-time marking is too expensive to maintain, use a conservative cached value and update it during allocate, deallocate, settlement, liquidation, and escrow-recovery events. At minimum, cap the number of simultaneously tracked swaps per adapter before using it in a vault strategy whose core user flows depend on `realAssets()`.

RESOLUTION

Status: Resolved.

LOW L-05 RESOLVED

Factory deploys unscoped market adapter

CATEGORY	Unexpected Behavior
LOCATION	KairosMarketAdapterFactory.sol
REVIEW	Main Review 1

DESCRIPTION

`KairosMarketAdapterFactory` presents itself as a safe one-adapter-per-vault deployment path, but the adapter it deploys is not scoped to any fixed Kairos market. The factory stores a single adapter per vault in `vaultToAdapter`, which gives the impression that the vault is being connected to a specific reviewed strategy surface. In reality, the deployed `KairosMarketAdapter` accepts an arbitrary `marketId` on every `allocate` call, tracks that market dynamically and returns only a single global `adapterId` back to Morpho for cap accounting.

```solidity
function createAdapter(address vault, address asset) external returns (address adapter) {
if (vaultToAdapter[vault] != address(0)) revert AdapterAlreadyExists();
...
adapter = address(new KairosMarketAdapter(vault, swapCore, asset));
...
vaultToAdapter[vault] = adapter;
}
```

```solidity
if (data.length == 32) {
marketId = abi.decode(data, (bytes32));
} else if (data.length == 64) {
(marketId, minSharesOut) = abi.decode(data, (bytes32, uint256));
}
...
swapCore.supplyCollateral(marketId, assets, address(this), 0);
...
ids = new bytes32[](1);
ids[0] = adapterId;
change = int256(assets);
```

That design collapses all Kairos market exposure for a vault into one Morpho accounting

L-05 Factory deploys unscoped market adapter CONTINUED

bucket even though the underlying Kairos markets can differ materially in oracle design, maturity profile, whitelist settings, collateralization, risk premium source and owner control. The problem is amplified by the fact that `SwapCore.createMarket` is permissionless. A new market with the same asset can be created at any time and immediately becomes reachable by the already-deployed adapter, without any new factory action and without any new Morpho cap domain.

A realistic example is a USDC vault where Alice deposits after Bob, the curator, adds the official Kairos LP adapter and sets what he believes is a conservative Morpho cap for a single reviewed 7 day market. Weeks later, Carol creates a second USDC Kairos market with a fragile or owner-controlled oracle. Bob's allocator key is then compromised, or Bob's strategy script simply selects the wrong `marketId`. Because the adapter is unscoped and always reports the same `adapterId`, the vault can move Alice's capital into Carol's market without violating the existing Morpho cap. From the vault's point of view nothing unusual happened at the cap layer, but economically Alice is now exposed to a completely different and potentially much riskier market than governance intended.

This is not just a monitoring inconvenience. It removes the main isolation boundary that Morpho caps are supposed to provide. A vault curator can no longer constrain exposure per Kairos market once this factory path is used, even though market-level risk varies in ways that are highly relevant to LP safety.

RECOMMENDATION

The adapter deployed by the factory should be market-scoped, not protocol-scoped. The cleanest fix is to deploy one adapter per Kairos market and make the market identifier immutable in the adapter constructor. If multi-market support is still required, the adapter should at minimum enforce an immutable allowlist of market ids and return a market-specific id, not just a single global `adapterId`, so Morpho caps and disable controls can operate at the same granularity as the actual Kairos risk surface.

RESOLUTION

Status: Resolved.

LOW L-06 RESOLVED

Collateralization ratio runs backwards

CATEGORY	Unexpected Behavior
LOCATION	`SwapFormulas.sol, SwapCore.sol`
REVIEW	Main Review 1

DESCRIPTION

The protocol exposes `collateralizationRatio` as if a larger value means a market is safer and more heavily collateralized, but the implementation does the opposite. Both buyer collateral and LP collateral are divided by `collateralizationRatio`, while available liquidity and maximum leverage increase with it. In other words, raising the ratio makes the market less collateralized and more levered even though the name and surrounding comments strongly suggest the reverse.

`SwapCore.createMarket()` describes the parameter as the ratio of LP collateral required per unit of notional exposure, and market validation allows values up to `10e18`. A market creator, frontend, or curator reading that interface would reasonably expect `2e18` to mean 200 percent collateralization and therefore stricter limits. In reality, `2e18` halves required collateral and doubles the liquidity the pool will advertise against the same capital base.

Consider a realistic case where Bob creates a permissionless USDC market and promotes it as conservative because it uses a `2e18` collateralization ratio. Alice sees the parameter, sees the usual wording around collateral, and assumes the market is safer than a `1e18` market. She supplies liquidity, and other users buy swaps against it. Under the hood, however, the pool is allowing roughly twice as much notional exposure per unit of collateral as Alice expects, and buyers are posting only half the collateral she thinks they are. A rate move that should have been comfortably absorbed by an overcollateralized market can now consume the pool far faster because the market was configured in the opposite direction from what the interface advertised.

RECOMMENDATION

The implementation and the public semantics need to be brought back into alignment. The best fix is to make the math match the name: if `collateralizationRatio` is supposed to represent more required collateral when it is larger, the collateral and liquidity formulas should be inverted accordingly and the accepted range should be reviewed around that meaning. If the current math is intentional, then the parameter should be renamed everywhere to reflect what it actually is, such as an inverse collateral factor or leverage factor, and

L-06 Collateralization ratio runs backwards CONTINUED

the creation path should reject values that are likely to be misread. The safer option is to fix the formulas and validation rather than preserving a misleading parameter name.

RESOLUTION

Status: Resolved.

LOW **L-07** RESOLVED

Full withdraw mode breaks vault pull flow

CATEGORY	Validation
LOCATION	`KairosMarketAdapter.sol`
REVIEW	Main Review 1

DESCRIPTION

`KairosMarketAdapter.deallocate()` has two different modes. In partial-withdraw mode, the vault passes a concrete `assets` amount and later pulls that same amount from the adapter. In full-withdraw mode, the adapter is supposed to send the realized `withdrawn` amount to the vault itself, which only works safely if `assets == 0`.

The problem is that the contract only enforces one side of that rule. It rejects `assets == 0` for partial withdrawals, but it does not reject `fullAmount == true` with `assets > 0`.

That breaks the Morpho pull flow. In full-withdraw mode the adapter may already have pushed `withdrawn` to the vault, but `VaultV2` still performs its normal `safeTransferFrom(adapter, vault, assets)` after `deallocate()` returns. If `assets` is nonzero, the vault tries to pull funds a second time even though the adapter was using the push-transfer path.

The outcome is a revert. The adapter sends the unlocked collateral to the vault, then the vault tries to pull `assets` from the adapter and fails because the adapter no longer has that balance. If the adapter happens to hold unrelated idle funds, the vault can pull those instead, which is even worse because the call succeeds with the wrong accounting.

RECOMMENDATION

The adapter should explicitly revert when `fullAmount == true` and `assets > 0`. That check should happen before any external call. This is the smallest fix and it keeps the two deallocation modes aligned with Morpho's expectations.

If the team wants to keep a custom all-assets unwind path, it would be cleaner to expose it separately instead of overloading the standard `deallocate()` flow with two incompatible transfer models.

RESOLUTION

Status: Resolved.

LOW L-08 RESOLVED

isSwapActive is true for unknown ids

CATEGORY	Warning
LOCATION	SwapCore.sol
REVIEW	Main Review 1

DESCRIPTION

`isSwapActive` does not check whether the swap exists. The root cause is the whole function:

```plain text
function isSwapActive(bytes32 swapId) external view returns (bool) {
return !swapPositions[swapId].settled;
}
```

For an unknown mapping key, Solidity returns the default struct. In that default struct settled is false, so the function returns true for a swap that was never created. Any offchain service that uses isSwapActive as an existence check will treat arbitrary ids as live swaps unless it adds its own second check.

RECOMMENDATION

Return false when the stored swap has `entryTimestamp == 0`, or revert for unknown ids. Either choice is better than conflating "missing" with "active."

RESOLUTION

Status: Resolved.

LOW L-09 RESOLVED

Unset multisig disables market creation fees

CATEGORY	Unexpected Behavior
LOCATION	SwapCore.sol, Admin.sol
REVIEW	Main Review 1

DESCRIPTION

`SwapCore.createMarket` treats the market-creation fee as configured as soon as `CREATE_MARKET_FEE_AMOUNT` is nonzero, but it only validates, emits and transfers that fee inside branches that also require `admin.getProtocolMultisig() != address(0)`. If the amount is set while `PROTOCOL_MULTISIG` is still unset, the function proceeds to create both markets and simply skips fee collection entirely.

This is inconsistent with the protocol-fee path. `Admin.setProtocolFeeEnabled` explicitly refuses to enable protocol fees until a nonzero multisig has been configured, but the market-creation fee setters have no equivalent guard. As a result, the protocol can appear to have market-creation fees enabled while market creation is actually free.

The deployment flow configures the market-creation fee token and amount, then logs that fees will be sent to `admin.getProtocolMultisig()`, but it does not configure that recipient in the same flow. In that state, anti-spam assumptions around paid market creation fail silently and any user can continue creating markets without paying the intended fee.

RECOMMENDATION

Require a nonzero fee recipient whenever market-creation fees are configured or charged. The simplest fix is to make `createMarket` revert if `CREATE_MARKET_FEE_AMOUNT > 0` and `admin.getProtocolMultisig() == address(0)`. It would also be safer to add the same invariant at configuration time so setting a nonzero market-creation fee amount is impossible until the recipient has already been set.

RESOLUTION

Status: Resolved.

LOW L-10 RESOLVED

Protocol fee timelock can be bypassed

CATEGORY	Unexpected Behavior
LOCATION	`Admin.sol, SwapCore.sol`
REVIEW	Main Review 1

DESCRIPTION

`Admin` only timelocks the numeric value of `PROTOCOL_FEE_RATE`. The owner can queue a nonzero rate through `queueProtocolFeeRate`, wait out `FEE_RATE_TIMELOCK`, activate that rate through `activateProtocolFeeRate` and still leave `protocolFeeEnabled` set to false. At that point the fee is armed but not yet charged. The actual switch that makes users start paying is `setProtocolFeeEnabled` and that function is immediate.

`SwapCore.buySwap` reads the live `Admin` state at execution time. It performs the buyer's slippage check on the swap rate first, then separately computes `protocolFee` from `admin.getProtocolFeeEnabled()` and `admin.getProtocolFeeRate()`, adds that fee into `totalBuyerRequired` and pulls the full amount from the buyer:

```plain text
if (rateBound != type(int256).max) {
if (market.rateType == Types.RateType.BUY_FIXED) {
if (swapRate > rateBound) revert Errors.E510();
} else {
if (baseRate < rateBound) revert Errors.E510();
}
}

uint256 protocolFee = 0;
if (admin.getProtocolFeeEnabled()) {
uint256 feeRate = admin.getProtocolFeeRate();
if (feeRate > 0) {
protocolFee = (notionalAmount * feeRate * swapTerm) / (SwapFormulas.SECONDS_IN_YEAR *
SwapFormulas.WAD);
}
}

uint256 totalBuyerRequired = requiredBuyerCollateral + protocolFee;
IERC20(swapToken).safeTransferFrom(msg.sender, address(this), totalBuyerRequired);
```

Because rateBound never constrains protocolFee or totalBuyerRequired`, a buyer has no

L-10 Protocol fee timelock can be bypassed CONTINUED

onchain way to say "execute only if fees are still off" or "execute only if the protocol fee is below X". An owner can therefore pre-arm a fee rate, watch pending `buySwap` transactions and frontrun them with `setProtocolFeeEnabled(true)`. If the buyer has enough balance and allowance, the swap still succeeds but pays an extra unexpected fee. If not, the transaction reverts. The issue is amplified by the wider protocol because market creation does not cap `swapTerm`, so the same annualized fee rate can become a large upfront charge on long-dated markets.

RECOMMENDATION

Timelock the whole protocol-fee configuration rather than only the numeric rate. At minimum, enabling fees should be queued and activated under the same delay as the rate itself so users get reliable notice before charging begins. It would also be safer to let buyers bound fee exposure directly, for example by adding a `maxProtocolFee` or `maxTotalBuyerRequired` parameter to `buySwap`, so governance cannot change fee charging between quote time and execution without giving users a way to reject the trade.

RESOLUTION

Status: Resolved.

LOW **L-11** RESOLVED

minTradeSize comment contradicts validation

CATEGORY	Validation
LOCATION	https://github.com/GuardianOrg/protocol-team1-1772465746057/blob/1e1150dea39d3b8c72eb06d1176e4b647fa4cfa6/packages/hardhat/contracts/SwapCore.sol#L124
REVIEW	Main Review 1

DESCRIPTION

SwapCore.createMarket documents minTradeSize as "0 = no minimum," but the actual validation path in Utils.validateMarketParams requires minTradeSize \> 0. Since createMarket always calls _initializeMarket, which calls Utils.validateMarketParams, any attempt to create a market with minTradeSize = 0 will revert.

RECOMMENDATION

Align code and docs. Either allow minTradeSize = 0 by removing the \> 0 require, or update the code comment (and any external docs) to explicitly state minTradeSize must be greater then zero.

RESOLUTION

Status: Resolved.

LOW L-12 RESOLVED

SwapClosed emits zero collateralBalance

CATEGORY	Events
LOCATION	SwapCore.sol, Utils.sol
REVIEW	Main Review 1

DESCRIPTION

The protocol exposes `SwapClosed` as the unified close event for expiry, early exit and liquidation and that event includes a field named `collateralBalance`. The problem is that this field is emitted after the settlement and liquidation helpers have already zeroed the underlying storage slot.

The normal settlement path makes this explicit. `Utils.settleSwap()` settles the net payment, prepares any buyer collateral release and then clears the buyer-side collateral before control returns to `SwapCore`:

```plain text
if (swap.collateralBalance > 0) {
collateralToRelease = swap.collateralBalance;
buyerCollateralReleased = collateralToRelease;
swap.collateralBalance = 0;
}
```

After that, SwapCore._settleSwaps() emits SwapClosed and passes swap.collateralBalance directly into the event:

```plain text
emit SwapClosed(
marketId,
swapId,
swap.userAddress,
msg.sender,
swap.isEarlyExit ? uint8(1) : uint8(0),
buyerCollateralReleased,
lpCollateralReleased,
amountSettled,
totalPaymentsMade,
netRecipient,
floatingPayment,
fixedPayment,
```

L-12 SwapClosed emits zero collateralBalance CONTINUED

swap.collateralBalance,
earlyExitFee
);
```
The liquidation path does the same thing. Utils.liquidateSwap() zeroes swap.collateralBalance before SwapCore.liquidateSwap() emits the unified close event, so liquidated positions also publish a zero collateralBalance even when buyer collateral originally existed.
This is not dead metadata in the repo as the indexer stores SwapClosed.collateralBalance on the unified swaps entity the frontend then trusts that stored value for settled positions:
`typescript
collateralBalance: isSettled
? BigInt(swapEntity.collateralBalance || 0)
: BigInt(swapEntity.buyerCollateral || 0),
`
The portfolio page then aggregates that settled collateralBalance into user metrics:
`typescript
const collateralBalance = BigInt(swap.collateralBalance || 0);
settledCollateral += collateralBalance;
```

As a result, closed positions are systematically reported with zero collateral in offchain views, even though users may care about the original deployed margin or about residual value that was released or escrowed at close. This can mislead portfolio views, buyer-vault accounting displays and any operator dashboard that relies on the unified event stream instead of re-reading full onchain state.

RECOMMENDATION

Do not emit `swap.collateralBalance` after the settlement helpers have zeroed it. If the protocol wants to expose pre-close buyer collateral, capture that amount in a local variable before clearing storage and emit that value instead. If the intended meaning is something else, rename the field so consumers do not interpret it as meaningful remaining collateral. Offchain consumers should also stop treating the current `SwapClosed.collateralBalance` value as settled collateral until the event payload is corrected.

RESOLUTION

Status: Resolved.

LOW L-13

RESOLVED

Market creation ignores oracle validity

CATEGORY	Validation
LOCATION	SwapCore.sol
REVIEW	Main Review 1

DESCRIPTION

`SwapCore._initializeMarket()` calls `getBaseRate()` and `getRate()` to validate the supplied oracles, but it only checks that the calls do not revert. It ignores the returned `isValid` booleans.

That means a market can be created with an oracle that is already stale or invalid, as long as the adapter reports that state with `false` instead of reverting. `ChainlinkAdapter` is one example: a stale feed returns `(0, false)`, but still passes market creation.

In the `ChainlinkAdapter` flow, the stale oracle passed `createMarket()`, the first LP deposit in the creation block succeeded and then later actions started failing once the protocol actually enforced oracle validity. So the protocol can create a market, accept funds and only afterwards reveal that the oracle was never usable.

RECOMMENDATION

Require the returned boolean to be `true` during market creation. A non-reverting call should not be treated as proof that the oracle is healthy.

```plain text
try IRateOracle(params.referenceRateOracle).getRate() returns (int256 rate, bool ok) {
require(ok, "referenceRateOracle invalid");
} catch {
revert("referenceRateOracle must implement IRateOracle");
}
```

Apply the same check to the base rate and optional risk premium oracles.

RESOLUTION

Status: Resolved.

LOW **L-14** ACKNOWLEDGED

Deploy script wires TWAP as Base-Rate oracle

CATEGORY	Warning
LOCATION	`17_deploy_base_markets.ts, SwapCore.sol`
REVIEW	Main Review 1

DESCRIPTION

The Base market deployment script builds AAVE and COMPOUND market configs by pairing the live reference-rate adapters with `TWAPOracle_*` contracts as `baseRateOracle`.

```typescript
if (AAVE_ORACLE && AAVE_TWAR) {
markets.push({
referenceRateOracle: AAVE_ORACLE,
baseRateOracle: AAVE_TWAR,
name: "AAVE",
});
}

if (COMPOUND_ORACLE && COMPOUND_TWAR) {
markets.push({
referenceRateOracle: COMPOUND_ORACLE,
baseRateOracle: COMPOUND_TWAR,
name: "COMPOUND",
});
}
```

`That wiring is incompatible with SwapCore. Market creation explicitly requires baseSwapRateOracle to implement IBaseRateOracle.getBaseRate(...).`

```solidity
try IBaseRateOracle(params.baseSwapRateOracle).getBaseRate(params.swapTerm) returns (int256, bool) {}
catch { revert("baseSwapRateOracle must implement IBaseRateOracle"); }
```

`But TWAPOracle only implements IRateOracle.`

```solidity
contract TWAPOracle is Ownable2Step, IRateOracle {
```

L-14 Deploy script wires TWAP as Base-Rate oracle CONTINUED

So the AAVE and COMPOUND market-creation calls in this script should revert during `_initializeMarket()`. The script then catches the error, logs it and continues. The result is not a direct security exploit, but it is still a meaningful deployment defect: operators can run the script expecting those markets to be created, while the intended markets are silently skipped from the final deployment state.

RECOMMENDATION

Make the deployment wiring match the oracle interfaces that `SwapCore` enforces. If the second oracle is meant to be a tenor-dependent base-rate source, pass a contract that implements `IBaseRateOracle`, such as `ImpliedBaseRate` or another adapter with a real `getBaseRate(...)` entrypoint. If `TWAPOracle` is intended to fill that role, it needs a compatible `getBaseRate(...)` interface before it is used in market creation.
The script should also fail when a configured market cannot be created, instead of catching the error and moving on as though the deployment was largely successful.

RESOLUTION

Status: Acknowledged.

LOW L-15 RESOLVED

Bounded early exit breaks on escrow fallback

CATEGORY	Logical Error
LOCATION	SwapCore.sol, Utils.sol
REVIEW	Main Review 1

DESCRIPTION

`SwapCore.exitSwapEarly()` advertises `minExitAmount` as the buyer's slippage guard for early exit. The problem is that the guard is enforced after settlement against `settlementPayouts`, while the protocol's failed-transfer fallback writes buyer proceeds into escrow and zeroes the exact value that `settlementPayouts` mirrors.

The flow in `exitSwapEarly()` is:

```plain text
results = _settleSwaps(swapIds);

for (uint256 i = 0; i < requests.length; i++) {
if (requests[i].minExitAmount == 0) continue;
uint256 buyerPayout = settlementPayouts[swapIds[i]];
if (buyerPayout < requests[i].minExitAmount) revert Errors.E552();
}
```

Inside `_settleSwaps()`, the recorded payout is taken directly from `buyerCollateralReleased`:

```plain text
settlementPayouts[swapId] = buyerCollateralReleased;
```

But `Utils.settleSwap()` clears that value whenever the payout had to be escrowed instead of transferred immediately:

```plain text
bool transferred = _transferOrEscrow(IERC20(swapToken), swap.userAddress, collateralToRelease, swapId, escrowedCollateral);
if (!transferred) {
buyerCollateralReleased = 0;
}
```

That means the slippage guard is really checking "how much was synchronously delivered to the recipient address" rather than "what economic exit value was realized for the buyer."

L-15 Bounded early exit breaks on escrow fallback CONTINUED

If the recipient cannot currently receive the token and the protocol would otherwise fall back to escrow, `settlementPayouts` becomes zero, the `minExitAmount` check fails and the whole early-exit transaction reverts. The escrow fallback is therefore unavailable precisely when the caller uses a nonzero slippage bound.

RECOMMENDATION

Do not enforce `minExitAmount` against only `settlementPayouts` after the transfer attempt. The bound should be checked against the economic buyer value realized by settlement before payout delivery is attempted, or against a value that includes escrowed proceeds when the transfer fails. In other words, slippage protection should measure exit economics, not synchronous token delivery. As a practical fix, compare `minExitAmount` to the computed buyer-side payout before `_transferOrEscrow`, or preserve a separate `buyerPayoutDue` value that remains nonzero even when the payout is redirected into escrow.

RESOLUTION

Status: Resolved.

LOW L-16 RESOLVED

Shared liquidityData breaks small withdrawals

CATEGORY	Unexpected Behavior
LOCATION	KairosMarketAdapter.sol
REVIEW	Main Review 1

DESCRIPTION

`VaultV2` stores a single `liquidityData` payload for its liquidity adapter and reuses that exact same bytes value for both automatic allocation during deposits and automatic deallocation during withdrawals. That design is incompatible with `KairosMarketAdapter`, because the adapter gives the second encoded word different meanings on entry and exit.

On allocation, a 64-byte payload is decoded as `(marketId, minSharesOut)`, so the second word acts as a deposit slippage guard. On deallocation, the same 64-byte payload is decoded as `(marketId, minCollateralOut)`, so that same number becomes a minimum withdrawal amount instead.

```plain text
if (data.length == 64) {
(marketId, minSharesOut) = abi.decode(data, (bytes32, uint256));
}
```

```plain text
if (data.length == 64) {
(marketId, minCollateralOut) = abi.decode(data, (bytes32, uint256));
}
```

This means a vault configured with abi.encode(marketId, minSharesOut) can accept deposits normally and still become unable to process healthy small withdrawals. Whenever the vault needs to pull liquidity from the adapter, it forwards the same payload into deallocate and the former deposit slippage value is enforced as minCollateralOut. If the requested withdrawal from Kairos is smaller than that number, SwapCore.withdrawCollateral reverts even though the market is liquid and the share price is healthy.

In practice, this makes the adapter's 64-byte deposit slippage mode unsafe when the adapter is used as Morpho's liquidity adapter. A more conservative minSharesOut` improves deposit protection but also increases the chance that ordinary user exits fail until governance or an allocator manually changes the shared configuration.

L-16 Shared liquidityData breaks small withdrawals CONTINUED

RECOMMENDATION

Do not reuse one shared `liquidityData` blob for two different parameter meanings. The cleanest fix is to make `KairosMarketAdapter` interpret the payload consistently on allocation and deallocation, or to disable the 64-byte slippage mode when the adapter is wired into `VaultV2` as the liquidity adapter. If separate slippage controls are required for deposits and withdrawals, the integration should expose separate configuration for allocate and deallocate data instead of relying on one shared bytes payload.

RESOLUTION

Status: Resolved.

LOW L-17

RESOLVED

PythAdapter Confidence Interval Not Validated

CATEGORY	Oracle
LOCATION	PythAdapter.sol
REVIEW	Main Review 1

DESCRIPTION

PythAdapter.getRate retrieves the Pyth price struct but never reads pythPrice.conf, the confidence interval field. Pyth publishes a confidence interval alongside every price to quantify uncertainty. During volatile market conditions, this interval can widen significantly, meaning the reported price may deviate from the true market rate. Pyth's documentation recommends rejecting prices where the confidence interval is too wide relative to the price itself. Without this check, the adapter can pass high-uncertainty prices into any downstream logic that consumes the oracle rate.

RECOMMENDATION

Reject or bound prices when pythPrice.conf is too large relative to pythPrice.price (e.g., enforce a max confidence ratio or use conservative bounds), in line with Pyth's confidence-interval guidance.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/606

LOW L-18

RESOLVED

BUY_FLOATING slippage guard ignores markup

CATEGORY	Unexpected Behavior
LOCATION	`SwapCore.sol`
REVIEW	Main Review 1

DESCRIPTION

`SwapCore.buySwap()` exposes `rateBound` as slippage protection, but on the `BUY_FLOATING` path it only checks `baseRate`. The deterministic markup that is actually charged at entry, `utilizationFee + riskPremium`, is computed separately and is not covered by the bound.

That creates a quote-to-execution gap. A prior trade can reduce available liquidity and push `utilizationFee` up, or the risk-premium oracle can move higher, while `baseRate` stays above the user's bound. The transaction still succeeds, but the buyer gets a more expensive `BUY_FLOATING` position than expected. `KairosBuyerAdapter.allocate()` forwards the same `rateBound`, so vault integrations inherit the same problem.

RECOMMENDATION

For `BUY_FLOATING`, the slippage check should cover the full entry cost, not just `baseRate`. Let callers cap either the total markup (`utilizationFee + riskPremium`) or the final all-in rate charged at execution. Any adapter should pass through and enforce the same bound.

RESOLUTION

Status: Resolved.

LOW L-19 RESOLVED

Swap transfer depends on oracle freshness

CATEGORY	Unexpected Behavior
LOCATION	`SwapCore.sol`
REVIEW	Main Review 1

DESCRIPTION

`transferSwapPosition()` only changes who owns an active swap, but it still forces a live `rateIndex.update()` on the market's reference oracle before the transfer can complete.

```plain text
swap.userAddress = newOwner;

rateIndex.update(markets[swap.marketId].referenceRateOracle);
```

`That pricing dependency is unnecessary for a pure custody move. No swap terms change, no collateral moves, and no settlement value is realized. If the reference oracle is stale, invalid, or temporarily reverting, the ownership transfer fails anyway. This turns an oracle outage into a custody freeze.`

`The impact reaches more than direct transfers because` SwapPositionWrapper `uses` transferSwapPosition() `inside both` wrap() `and` unwrap()`. For example imagine the reference oracle goes unhealthy for a few hours while a user is trying to move a live position into the wrapper for downstream use, or trying to unwrap it so they can send it to a rescue wallet. Nothing about that action needs a fresh rate, but the transfer still reverts because the function insists on updating the index first. The user is stuck with the current custody layout until the oracle recovers. Therefore, the protocol is blocking ownership mobility in a path that should remain available even when pricing functions are paused by an oracle problem.

RECOMMENDATION

Remove the oracle update from `transferSwapPosition()`. The custody transfer should not depend on live oracle availability when no economic state is being repriced. If the protocol still wants an explicit refresh hook, keep it as a separate call or perform it only in functions that actually consume pricing or settlement state. If some freshness bookkeeping must remain near transfers, make it non-blocking so an unhealthy oracle cannot trap users in the wrong custody path.

L-19 Swap transfer depends on oracle freshness CONTINUED

RESOLUTION

Status: Resolved.

LOW L-20

RESOLVED

Duplicate tenors can bypass maxRateChangeWad

CATEGORY	Validation
LOCATION	FixedRatePricingOracle.sol
REVIEW	Main Review 1

DESCRIPTION

`postForecasts` does not reject duplicate entries in `tenors`. The function writes each item immediately, so the same tenor can be updated several times in one transaction. Each step is compared only against the value written in the previous loop iteration, not against the rate that existed before the batch started. That makes the rate-change breaker weaker than it looks. A batch like `[6%, 7%, 8%]` for the same tenor will pass a `1%` breaker if the previous stored rate was `5%`, even though the final rate moved by `3%` in one call.

RECOMMENDATION

Reject duplicate tenors in `postForecasts`. The safest change is to require each active tenor to appear exactly once per batch. If partial updates remain allowed, at minimum keep a per-call seen set and revert on duplicates before any writes happen.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/609

LOW L-21 RESOLVED

Interpolation can mix different forecast epochs

CATEGORY	Unexpected Behavior
LOCATION	`FixedRatePricingOracle.sol`
REVIEW	Main Review 1

DESCRIPTION

`postForecasts` accepts any subset of tenor buckets and writes each bucket independently. The contract does not track a curve-level snapshot or epoch, so adjacent buckets can come from different posting runs while still looking fresh enough to use together.

This weakens the integrity of interpolated quotes. A 75 day rate can be built from one endpoint posted in an older model run and another endpoint posted in a newer run, even though those values were never meant to describe the same curve. That can distort pricing whenever the off-chain engine updates the curve incrementally.

RECOMMENDATION

Treat each forecast curve as a single snapshot. The smallest safe change is to store and validate a per-post epoch or batch timestamp, then require both interpolation endpoints to share that epoch before returning a quote. If partial updates are not needed, require every configured tenor bucket to be posted atomically in one call.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/610`

LOW L-22 RESOLVED

Last LP full withdraw leaves residual collateral

CATEGORY	Warning
LOCATION	SwapCore.sol
REVIEW	Main Review 1

DESCRIPTION

`SwapCore.withdrawCollateral()` rounds `fullAmount` withdrawals down from the current share price:

```solidity
sharePrice = getPoolSharePrice(marketId);

if (fullAmount) {
sharesToRedeem = lpPos.shares;
withdrawalAmount = Math.mulDiv(sharesToRedeem, sharePrice, WAD);
}
```

It then burns all redeemed shares and subtracts only that rounded amount from pool.totalCollateral:

```solidity
lpPos.shares -= sharesToRedeem;
pool.totalShares -= sharesToRedeem;
pool.totalCollateral -= withdrawalAmount;
```

When the withdrawing LP is the last shareholder and the pool share price is above 1e18, this double rounding can leave a small remainder in pool.totalCollateral even though pool.totalShares has been reduced to zero. The issue is not limited to direct calls into SwapCore.withdrawCollateral(). It is also reachable through the vault adapter flow because KairosMarketAdapter.deallocate() forwards fullAmount=true withdrawals into the same core function.

That breaks the protocol's own empty-pool assumptions. getPoolSharePrice() returns 1e18 whenever totalShares == 0, so the leftover collateral is masked rather than reconciled. The dust is small, but it is still incorrect accounting. It can remain stranded indefinitely if the market is never reinitialized, or be transferred to the next LP that becomes the first depositor after the pool is reopened. Because the same logic is used by adapter-mediated exits, the bug affects both direct LP withdrawals and Morpho vault deallocations.

L-22 Last LP full withdraw leaves residual collateral CONTINUED

RECOMMENDATION

When a `fullAmount` withdrawal burns the last outstanding LP shares, the withdrawal path should not leave residual collateral behind. The smallest safe fix is to detect the last-shareholder case and pay out the entire remaining unlocked collateral, or equivalently add any rounding remainder to `withdrawalAmount` before zeroing the pool. The implementation should preserve the invariant that `pool.totalShares == 0` implies `pool.totalCollateral == 0` and `pool.lockedCollateral == 0` after a successful unwind. Add regression coverage for both a direct `withdrawCollateral(fullAmount=true)` path and the `MorphoVault.deallocate -> KairosMarketAdapter.deallocate -> SwapCore.withdrawCollateral(fullAmount=true)` path.

RESOLUTION

Status: Resolved.

LOW L-23 RESOLVED

calculateMaxLeverage under-scales WAD output

CATEGORY	Documentation
LOCATION	SwapFormulas.sol
REVIEW	Main Review 1

DESCRIPTION

`SwapFormulas.calculateMaxLeverage()` is documented to return a leverage ratio "in WAD precision":

```plain text
* @return Maximum leverage ratio (unitless, in WAD precision)
```

But the implementation never multiplies the numerator by `WAD`:

```plain text
return (SECONDS_IN_YEAR * collateralizationRatioWad) / (effectiveRate(swapRateWad) * term);
```

That means any valid leverage below `1e18` `truncates to` `0` `instead of returning a nonzero WAD-scaled ratio. With` swapRateWad = 4.400359769748619533e18`,` term = 25261170`, and` collateralizationRatioWad = 1.622899438354856419e18`, the function returns` `0` `even though the correctly scaled result is about` 4.604e17`.`

This helper is only exercised by the test wrappers and the implemented fuzzing suite, so the immediate impact is limited to incorrect leverage reporting and broken assumptions in any offchain or future onchain consumer that treats the result as WAD-scaled. The bug is still real because the code and NatSpec disagree and the function currently returns a value in the wrong units.

RECOMMENDATION

Make the implementation match its declared units by applying `WAD` scaling in the numerator:

```plain text
return Math.mulDiv(SECONDS_IN_YEAR * collateralizationRatioWad, WAD, effectiveRate(swapRateWad) * term);
```

L-23 calculateMaxLeverage under-scales WAD output CONTINUED

RESOLUTION

Status: Resolved.

LOW L-24

RESOLVED

Liquidity Routing Can Block Deposits

CATEGORY	DoS
LOCATION	https://github.com/GuardianOrg/protocol-team1-1772465746057/blob/13595591dbf64c6b24d55613bc1d6de5b900e0c9/packages/hardhat/contracts/SwapCore.sol#L452
REVIEW	Main Review 1

DESCRIPTION

Vault deposits can be temporarily blocked when the configured liquidity adapter route points to a market that cannot currently accept LP deposits. For example, SwapCore.supplyCollateral() reverts when sharePrice == MIN_SHARE_PRICE and lockedCollateral \> 0, and that failure propagates through KairosMarketAdapter into the vault deposit path. This is notable because KairosMarketAdapter can manage multiple markets, so deposits may still be valid for other markets, yet deposits are still blocked as long as the vault remains configured to route through the unavailable one. The same issue can also arise if the configured market is terminated or the adapter path otherwise becomes unusable.

RECOMMENDATION

Be aware that this can temporarily block deposits and may require manual intervention to update or clear the liquidity adapter configuration.

RESOLUTION

Status: Resolved.

INFORMATIONAL **I-01** RESOLVED

Buyer factory accepts counterfeit vaults

CATEGORY	Trust Assumptions
LOCATION	`KairosBuyerAdapterFactory.sol`
REVIEW	Main Review 1

DESCRIPTION

`KairosBuyerAdapterFactory.createAdapter()` is presented as a factory for Morpho Vault V2 adapters, but it does not actually verify that the `vault` argument is a real Morpho vault. It only checks that the address has code, that `asset()` returns the expected token, and that the caller matches `owner()` or `curator()`. Any contract can be written to satisfy those checks. A fake vault only needs to expose the same small surface and return values that make the factory happy. Once that happens, the factory deploys a real `KairosBuyerAdapter` for it and emits the same `AdapterCreated` event it would emit for a genuine Morpho vault.

That is the whole problem. The factory looks like a source of truth for "official Kairos buyer adapters for Morpho vaults", but in practice it is only a deployer for any contract that imitates the expected interface. As soon as one counterfeit vault passes the check, factory provenance stops being a reliable legitimacy signal. For example, imagine that Bob deploys a fake vault contract that returns USDC from `asset()` and returns Bob from `owner()` and `curator()`. He then calls the official buyer-adapter factory and gets a real adapter plus a real factory event. Alice later sees that adapter on a dashboard or in an indexer that trusts the factory address and assumes it belongs to a genuine Morpho vault. The factory has now helped Bob produce something that looks official even though the underlying vault was never authenticated.

RECOMMENDATION

The factory should verify that `vault` comes from a canonical Morpho source of truth instead of trusting a loose interface check. The best fix is to require that the vault was deployed by the approved Morpho vault factory or is listed in an authoritative Morpho registry. If that is not available, a strict codehash or version check is still better than the current behavior.

If the team intentionally wants a permissionless deployment path for arbitrary vault-like contracts, that path should be separated and clearly named so the main factory and its events still mean what users and integrators expect them to mean.

I-01 Buyer factory accepts counterfeit vaults CONTINUED

RESOLUTION

Status: Resolved.

INFORMATIONAL I-02 RESOLVED

Unused totalNotionalReduction in settleSwap

CATEGORY	Informational
LOCATION	https://github.com/GuardianOrg/protocol-team1-1772465746057/blob/1e1150dea39d3b8c72eb06d1176e4b647fa4cfa6/packages/hardhat/contracts/lib/Utils.sol#L367
REVIEW	Main Review 1

DESCRIPTION

Utils.settleSwap declares uint256 totalNotionalReduction = 0; and conditionally assigns it to swap.notionalAmount when pool collateral backing is released, but the variable is never read, returned, or used to update state.

RECOMMENDATION

Remove the unused variable, or, if the notional reduction logic was intended, update the relevant state explicitly.

RESOLUTION

Status: Resolved.

INFORMATIONAL I-03 ACKNOWLEDGED

ImpliedBaseRate lets users manipulate pricing

CATEGORY	Logical Error
LOCATION	`ImpliedBaseRate.sol, SwapCore.sol, Utils.sol`
REVIEW	Main Review 1

DESCRIPTION

`ImpliedBaseRate` is not an external market rate. It is calculated from current protocol liquidity, so the same user who is about to trade can first move liquidity and then immediately consume the new rate. The protocol still treats that value like a neutral oracle input in LP pricing, swap entry, and early-exit settlement.

```solidity
uint256 correspondingAvailableLiquidity = SwapFormulas.calculateAvailableLiquidity(
swapCore.totalLPAvailableCollateral(crsMarketId),
referenceRate,
market.swapTerm,
market.collateralizationRatio
);

uint256 currentAvailableLiquidity = SwapFormulas.calculateAvailableLiquidity(
swapCore.totalLPAvailableCollateral(currentMarketId),
referenceRate,
currentMarket.swapTerm,
currentMarket.collateralizationRatio
);
```

`Once that rate is exposed through` Utils.getBaseRateOracle() `, it reaches every pricing path that matters.` Utils.calculateTotalUnrealizedPnL() `and` Utils.calculateLpSharePrice() `use it to mark LP positions, while` SwapCore.supplyCollateral() `and` SwapCore.withdrawCollateral() `mint and burn shares against the price observed before balances are updated.` SwapCore.buySwap() `uses the same rate to lock swap terms and size collateral, and the early-exit branch in` _settleSwaps() `reads it again to project the value of the remaining leg.`

`For example imagine a trader holds a position they want to exit early. Before calling the exit path, they first add or remove liquidity in the paired market used by` ImpliedBaseRate`. That shifts the reported base rate without any real change in the outside market. The

I-03 ImpliedBaseRate lets users manipulate pricing CONTINUED

trader then exits while `_settleSwaps()` reads the distorted rate. Because `Utils.calculateEarlyExitSettlement()` uses that projected rate to value the remaining payments, the trader can make the exit cheaper or more favorable and push the loss onto passive LPs. The same idea applies on entry: move liquidity first, then buy the swap at a better coupon or collateral requirement. It also applies to LP shares: mint or burn at the old share price, then let the same liquidity move reprice the pool immediately after.

This is worse in an early exit because the projected rate is not only used to size remaining payments. It can also decide whether those projected payments are counted at all.

```solidity
bool projectedFavorsBuyer = (market.rateType == Types.RateType.BUY_FIXED &&
remainingFloatingPayment > remainingFixedPayment) ||
(market.rateType == Types.RateType.BUY_FLOATING && remainingFixedPayment >
remainingFloatingPayment);

if (projectedFavorsBuyer) {
totalFixedPayment = accruedFixedPayment;
totalFloatingPayment = accruedFloatingPayment;
} else {
totalFixedPayment = accruedFixedPayment + remainingFixedPayment;
totalFloatingPayment = accruedFloatingPayment + remainingFloatingPayment;
}
```

`LP share pricing and swap pricing both become functions of mutable protocol liquidity instead of an external rate. If markets use ImpliedBaseRate here, users can create their own repricing window and extract value from passive LPs.`

`This is not just an unused code path. The current deployment set in this repository already includes deployed ImpliedBaseRate contracts on Base and the active protocol integration setup also creates a market that uses ImpliedBaseRate as its baseSwapRateOracle`.

RECOMMENDATION

Do not allow `ImpliedBaseRate` to be used anywhere the protocol needs a manipulation-resistant base rate. `buySwap()`, LP share pricing and early-exit settlement should read an exogenous or time-lagged rate that cannot be changed by the same user in the same pricing window. If market imbalance is meant to affect pricing, compute that adjustment inside `SwapCore` from a snapshot of pre-action state and apply it explicitly as part of the transaction, instead of routing it through an oracle that depends on live balances. As an immediate safeguard, reject `ImpliedBaseRate` as `baseSwapRateOracle` for markets that support LP mint/burn or early exit until the pricing flow is redesigned.

I-03 ImpliedBaseRate lets users manipulate pricing CONTINUED

RESOLUTION

Status: Acknowledged.

INFORMATIONAL **I-04** RESOLVED

US10YR cranker pays for unchanged Pyth updates

CATEGORY	Oracle
LOCATION	`crank_oracles.ts, constants.ts, deploy.sh, PythAdapter.sol`
REVIEW	Main Review 1

DESCRIPTION

The Base oracle cranker treats `US10YRTreasury` as stale whenever the onchain Pyth `publishTime` is more than ten minutes old and the scheduler runs every minute. That creates a bad interaction with a session-based feed. Outside Treasury hours, Hermes keeps returning the same last-session payload and the onchain Pyth contract keeps the same last-session `publishTime`. After ten minutes, the cranker starts deciding that every run needs an update even though there is nothing new to submit.

The write path then still fetches the Hermes binary, pays `getUpdateFee` and calls `updatePriceFeeds(...)`. Pyth explicitly allows `updatePriceFeeds` to succeed even when the payload is not newer. The function meant to avoid this, `updatePriceFeedsIfNecessary(...)`, is not used. The result is repeated paid no-op writes throughout nights, weekends and holidays.

This is live-reachable on Base. There are currently two unpaused markets using `PythAdapter_US10YR` as their `referenceRateOracle` and the latest US10YR payload currently costs `0.0000015 ETH` to submit. The immediate effect is wasted keeper spend. The more important effect is that the updater wallet gets drained faster, which increases the chance that a later real Pyth update is missed and the market falls into the separate stale-oracle failure mode already logged.

RECOMMENDATION

Do not trigger Pyth writes from age alone. Before sending an update, compare the latest Hermes `publish_time` against the current onchain `publishTime` and skip the transaction unless the offchain payload is newer. The best fix is to replace the current `updatePriceFeeds(...)` call with `updatePriceFeedsIfNecessary(...)`, passing the feed id and Hermes publish time.

For session-based feeds such as `US10YR`, the keeper should also stop treating off-hours as a reason to send updates at all.

I-04 US10YR cranker pays for unchanged Pyth updates CONTINUED

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/604

INFORMATIONAL I-05 ACKNOWLEDGED

Use minimal proxies for buyer adapters

CATEGORY	Gas Optimization
LOCATION	`KairosBuyerAdapterFactory.sol`
REVIEW	Main Review 1

DESCRIPTION

Every call to `KairosBuyerAdapterFactory.createAdapter` deploys a full new `KairosBuyerAdapter` instance with `new KairosBuyerAdapter(...)`, then runs constructor logic to set `parentVault`, `swapCore`, `asset`, compute `adapterId` and issue two max approvals. That full bytecode deployment is the main gas cost of the `KairosBuyerAdapterFactory` contract\`.
A minimal proxy or clone-with-immutable-args design would replace full adapter deployments with a much smaller proxy deployment plus initialization, cutting the cost of each new adapter materially.

RECOMMENDATION

Refactor `KairosBuyerAdapterFactory` to deploy a shared implementation through an EIP-1167 proxy or a clone-with-immutable-args pattern. Since `KairosBuyerAdapter` currently relies on constructor-set immutables and constructor-time approvals, the practical options are to move setup into a one-time `initialize` flow, or to adopt a clone pattern that keeps immutable args while still handling approvals during initialization.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-06 RESOLVED

Avoid copying swapId arrays in buyer views

CATEGORY	Gas Optimization
LOCATION	KairosBuyerAdapter.sol
REVIEW	Main Review 1

DESCRIPTION

`KairosBuyerAdapter.realAssets()` and `KairosBuyerAdapter.collateralInMarket()` both load `_marketSwapIds[marketId]` into a memory array before iterating it. That forces a full storage-to-memory copy of every tracked swap id array on each call, even though the functions only need sequential read access.

This is unnecessary overhead on the adapter's hottest read paths. `realAssets()` is especially sensitive because it already walks every tracked buyer position and is part of the vault-facing valuation surface. Replacing the memory copy with a storage reference and a cached length removes the array materialization cost while preserving behavior.

Gas savings: deterministic savings on every non-empty market iteration in both view functions, with the benefit scaling linearly with the number of tracked swaps per market.

RECOMMENDATION

Iterate the storage array directly instead of copying it to memory first. A minimal pattern is:

```plain text
bytes32[] storage swapIds = _marketSwapIds[marketId];
uint256 swapLen = swapIds.length;
for (uint256 j; j < swapLen; ) {
bytes32 swapId = swapIds[j];
...
unchecked { ++j; }
}
```

`Apply the same change to both realAssets() and collateralInMarket()`.

RESOLUTION

Status: Resolved.

INFORMATIONAL **I-07** RESOLVED

Add known-settled credit helper in wrapper

CATEGORY	Gas Optimization
LOCATION	`SwapPositionWrapper.sol`
REVIEW	Main Review 1

DESCRIPTION

Several single-token flows in `SwapPositionWrapper` already know the position is settled before they try to credit proceeds. `settle()` checks the position and may settle it itself, `exitEarly()` settles before crediting, and `redeem()` explicitly verifies settlement. Even so, all of them route through `_creditIfNeeded()`, which calls `swapCore.swapPositions()` again just to re-check a fact the caller already established.

A focused benchmark in a temporary Foundry harness measured this exact pattern on a settled single-token path. The current version averaged `49,346` gas, while a version that switched to a `_creditKnownSettled(...)` helper averaged `47,148` gas. That is a saving of `2,198` gas per call.

Gas savings: about `2,198` gas per call on known-settled paths such as `settle()`, `exitEarly()`, and `redeem()`.

RECOMMENDATION

Split the current helper into two paths: keep `_creditIfNeeded()` for callers that genuinely do not know settlement status and add a `_creditKnownSettled(tokenId, swapId)` variant for call sites that already verified or caused settlement. That removes one redundant external `swapPositions()` read from each of those flows.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/617`

INFORMATIONAL I-08 ACKNOWLEDGED

Reuse updated index in LP share price path

CATEGORY	Gas Optimization
LOCATION	`SwapCore.sol`
REVIEW	Main Review 1

DESCRIPTION

Both `supplyCollateral()` and `withdrawCollateral()` update the reference-rate index and then immediately go through the share-price path. At that point no time has elapsed, so `getPoolSharePrice()` just re-reads the same stored index through `_getFreshIndex()` before passing it into `Utils.calculateLpSharePrice(...)`.
A focused benchmark measured this pattern at `32,713` gas in its current form and `30,993` gas when the fresh index from `update()` is passed directly into `Utils.calculateLpSharePrice(...)`, for a saving of `1,720` gas per call.
Gas savings: about `1,720` gas per `supplyCollateral()` or `withdrawCollateral()` call on this path.

RECOMMENDATION

In `supplyCollateral()` and `withdrawCollateral()`, keep the value returned by `rateIndex.update(...)` and pass it straight into `Utils.calculateLpSharePrice(...)` instead of immediately routing back through `getPoolSharePrice()`.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-09 RESOLVED

Cache swapPositions in settleBatch

CATEGORY	Gas Optimization
LOCATION	SwapPositionWrapper.sol
REVIEW	Main Review 1

DESCRIPTION

`SwapPositionWrapper.settleBatch()` currently re-reads `swapCore.swapPositions(swapId)` more often than needed. For an uncredited token, the function can hit `swapPositions()` once in the counting loop, again in the array-fill loop and then a third time through `_creditIfNeeded()` during the final credit loop. A focused benchmark in a temporary Foundry harness measured a representative 10-token batch where every token was uncredited and unsettled. The current flow averaged `932,348` gas, while a one-scan version that cached `tokenId` and `swapId` in memory, batch-called `makePayment()` and credited directly without re-reading settlement status averaged `889,541` gas. That is a saving of `42,807` gas for the batch, or about `4,280` gas per token on this path.
Gas savings: about `42,807` gas for a 10-token all-unsettled batch, or about `4,280` gas per token. The saving scales with batch size.

RECOMMENDATION

Refactor `settleBatch()` into a single discovery pass. Cache the relevant `tokenId` and `swapId` pairs in memory, collect the unsettled `swapId`s for `makePayment()` and then credit the cached positions directly instead of routing back through `_creditIfNeeded()` and re-checking settlement.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/613

INFORMATIONAL **I-10** RESOLVED

Cache fresh rate index in buySwap

CATEGORY	Gas Optimization
LOCATION	`SwapCore.sol`
REVIEW	Main Review 1

DESCRIPTION

`SwapCore.buySwap()` calls `rateIndex.update(referenceRateOracle)`, then ignores the returned index and reads `rateIndex.getIndex(referenceRateOracle)` twice to populate `entryFloatingIndex` and the `SwapCreated` event. Those reads are redundant because `update()` already returns the fresh value.
A focused benchmark measured this exact pattern at `50,115` gas in its current form and `48,761` gas with the cached-index version, for a saving of `1,354` gas per call.
Gas savings: about `1,354` gas per `buySwap()` call.

RECOMMENDATION

Store the return value of `rateIndex.update(referenceRateOracle)` in a local variable and reuse it for both the struct field and the event instead of calling `rateIndex.getIndex(referenceRateOracle)` again.

RESOLUTION

Status: Resolved.

INFORMATIONAL I-11 RESOLVED

PR #544 fix leaves post-expiry repricing open

CATEGORY Warning
LOCATION https://github.com/kairosswap/protocol/pull/544
REVIEW Main Review 1

DESCRIPTION

[PR #544](https://github.com/kairosswap/protocol/pull/544) adds historical snapshots and uses them in `getSwapNetAmount()` and settlement, but it does not apply the same rule to LP mark-to-market. `calculateBucketUnrealizedPnL()` still derives the floating leg from the live `oracleIndex` with uncapped `totalElapsed`, so expired-but-unsettled LP exposure keeps moving when the oracle changes after maturity. `SwapCore.getPoolSharePrice()` still routes through this path, so the branch does not fully close [M-01](/31e8bda5828c8167893fc28604be8e71?v=3178bda5828c818f8c9c000c8a2b48ea&pvs=25). This is not just a stale view path. The same share price is used by `supplyCollateral()`, `withdrawCollateral()` and `KairosMarketAdapter.realAssets()`, so live LP deposits, withdrawals and downstream valuations can still be priced off post-expiry oracle movement.

The branch also makes expiry freezing best-effort instead of deterministic. `RateIndexLib.getIndexAt()` returns `0` when there is no post-target snapshot, `SwapCore.getSwapNetAmount()` then falls back to the legacy `_getFreshIndex()` path, and `Utils.settleSwap()` likewise falls back to current-index pricing when `expiryIndex == 0`. That means both preview and settlement are only fixed when historical reconstruction succeeds. The preview fallback still feeds `KairosBuyerAdapter.realAssets()`, so buyer-side mark-to-market can remain drift-prone until someone calls `updateMarketRateIndex()` after expiry. The new `OracleIndexDriftTest` file only exercises the fixed path after explicitly seeding that update, so this window is not covered by the branch tests.

One more sign that the fix stopped short is in the LP pricing code itself. The comment in `calculateBucketUnrealizedPnL()` still says the uncapped `totalElapsed` path "matches settlement." That was true before the refactor, but it is no longer true once settlement switches to expiry-index pricing. The stale comment is consistent with the code: the settlement path was updated, while the LP mark-to-market path was left behind.

There is also a second edge case in the new snapshot design. Bitmap scans are capped at roughly 35 days, but `swapTerm` is not similarly bounded. A dormant or long-dated market can therefore drop back to the legacy path instead of resolving an expiry index.

I-11 PR #544 fix leaves post-expiry repricing open CONTINUED

RECOMMENDATION

Apply the expiry-snapshot rule to every post-expiry pricing path, not just swap preview and settlement. `getPoolSharePrice()` and the bucket mark-to-market logic should resolve the floating leg from the expiry index and capped duration once a bucket is past maturity. Remove the silent `expiryIndex == 0` fallback from user-facing mark-to-market paths, or replace it with a deterministic rule that cannot reintroduce live-oracle repricing. If bounded bitmap scans are kept, enforce a matching maximum supported horizon or store enough checkpoint data to guarantee lookup for the full supported `swapTerm`.

RESOLUTION

Status: Resolved.

MEDIUM M-01

RESOLVED

Cumulative markets still call getRate()

CATEGORY	Logical Error
LOCATION	SwapCore.sol
REVIEW	Remediation Review 1

DESCRIPTION

`createMarket()` accepts `RateConvention.Cumulative` and `RateIndexLib.initialize()` validates that mode through `getCumulativeIndex()`, but `buySwap()` still unconditionally calls `Utils.getOracleRate(referenceRateOracle)` before any market-type branching. The shipped cumulative adapters under `contracts/oracles/cumulativeIndex/` implement `getCumulativeIndex()` and `decimals()`, not `getRate()`, so a market can be created successfully and then revert on the first buy.

The same assumption is repeated in the quote and implied-rate paths:

```plain text
int256 referenceRate = Utils.getOracleRate(market.referenceRateOracle);
```

That leaves cumulative markets in a half-initialized state where the advertised oracle mode is accepted at creation time but not supported by execution-time code paths.

RECOMMENDATION

Split the pricing oracle from the settlement index oracle. For cumulative markets, never call `getRate()` on the reference oracle unless a separate spot-pricing oracle is configured for that purpose. Any path that only needs the cumulative index should use `getCumulativeIndex()` exclusively.

RESOLUTION

Status: Resolved.

MEDIUM M-02

ACKNOWLEDGED

Aave borrow index grows with zero debt

CATEGORY	Unexpected Behavior
LOCATION	`CumulativeIndexBorrowAave.sol`
REVIEW	Remediation Review 1

DESCRIPTION

`CumulativeIndexBorrowAave` assumes `getReserveNormalizedVariableDebt(asset)` always represents borrow interest that has actually accrued on Aave. That is not true when the reserve has no variable debt. In that state, Aave can still return a higher projected debt index based on the current borrow rate and elapsed time, even though its stored borrow index does not move because there are no borrowers paying variable interest.

Kairos uses this adapter as a cumulative floating-rate source. If the linked Aave reserve spends time with zero variable debt, Kairos can keep accruing floating interest that never existed on the source market. That misprices the floating leg and can shift value between LPs and buyers at mark-to-market and settlement.

RECOMMENDATION

Do not use `getReserveNormalizedVariableDebt()` on its own. The adapter should also check whether the reserve currently has non-zero variable debt. If debt is zero, it should return the stored borrow index instead or mark the oracle invalid. If that cannot be implemented safely, this adapter should only be used for reserves where non-zero variable debt is guaranteed as an operating assumption.

RESOLUTION

Status: Acknowledged.

MEDIUM M-03 ACKNOWLEDGED

Morpho borrow share price is not cumulative

CATEGORY	Unexpected Behavior
LOCATION	CumulativeIndexBorrowMorpho.sol
REVIEW	Remediation Review 1

DESCRIPTION

CumulativeIndexBorrowMorpho reports Morpho's borrow share price as if it were a monotone cumulative index:

```plain text
uint256 index = (borrowAssets + VIRTUAL_ASSETS) * 1e27
/ (borrowShares + VIRTUAL_SHARES);
```

That assumption is wrong. Morpho interest accrual grows totalBorrowAssets while leaving totalBorrowShares unchanged, but normal borrow() and repay() flows also change totalBorrowShares with asymmetric rounding. Upstream Morpho rounds borrow-share minting up and repay-share burning down. Those rounding steps can make the reported share price move down across legitimate market activity even when the underlying borrow rate is non-negative. That breaks the contract's cumulative-oracle promise. Kairos accepts this value through RateIndexLib as realized floating accrual and later annualizes index changes with SwapFormulas.deriveRateFromIndex()`. Once the reported "cumulative" index can fall, ordinary Morpho borrower activity can make Kairos infer a negative floating rate from a market that never had a negative borrow rate in the first place.

RECOMMENDATION

Do not use Morpho borrow share price as a RateConvention.Cumulative oracle. If Morpho borrow exposure is needed, use a spot-rate integration and compound it in Kairos, or build an adapter around a truly monotone realized borrow index instead of raw borrow-share price.

RESOLUTION

Status: Acknowledged.

MEDIUM **M-04** ACKNOWLEDGED

Expiry pricing includes post-expiry moves

CATEGORY	Unexpected Behavior
LOCATION	`RateIndexLib.sol`
REVIEW	Remediation Review 1

DESCRIPTION

`RateIndexLib.getIndexAt()` reconstructs a historical index by interpolating between the nearest stored snapshots around `targetTime`. That is only exact if the index path between those checkpoints is the one the protocol wants to price. In settlement-critical flows that assumption does not hold.

`SwapCore._settleSwaps()` first calls `rateIndex.update(...)`, which appends a new snapshot at the current settlement time and only then resolves `expiryIndex = rateIndex.getIndexAt(..., actualExpiry)`. If the book was quiet and there was no checkpoint near expiry, the reconstructed expiry index is pulled toward that late post-expiry snapshot. `getSwapNetAmount()` and LP expiry pricing rely on the same historical lookup once such a late snapshot exists.

That means the current design does not fully freeze value at maturity. It removes the old unbounded "use the live index" bug, but a delayed first post-expiry checkpoint can still leak post-expiry rate movement back into the supposed expiry price. With an entry snapshot at 5%, a first post-expiry snapshot one term later after the oracle has moved to 50%, the reconstructed expiry rate lands near the midpoint instead of the true expiry level. Settlement and mark-to-market then price the floating leg off a contaminated expiry index.

RECOMMENDATION

Do not treat `getIndexAt()` as exact for settlement-critical expiry pricing unless the protocol guarantees a checkpoint close to expiry. The smallest safe fix is to store an explicit expiry snapshot when a swap or bucket crosses maturity, or otherwise resolve expiry only from pre-expiry data. If that cannot be guaranteed, settlement-critical paths should only proceed when the available snapshots pin down expiry closely enough. Otherwise, the protocol should wait for a reliable expiry checkpoint instead of estimating expiry from a much later post-expiry update.

RESOLUTION

Status: Acknowledged.

MEDIUM M-05

RESOLVED

LP Gate Triggers Before Expiry

CATEGORY	Logical Error
LOCATION	`SwapCore.sol`
REVIEW	Remediation Review 1

DESCRIPTION

The expired-unsettled LP gate is keyed to a coarse expiry epoch rather than the true swap expiry timestamp. When a swap is opened, buySwap() records expiryEpoch = floor((entryTimestamp + swapTerm) / bucketInterval). Later, _revertIfExpiredUnsettled() treats the market as blocked once expiryEpoch * bucketInterval \<= block.timestamp.

That means the gate activates at the start of the bucket containing expiry, not at the actual maturity time of the earliest unsettled swap. If a position is opened near the end of a bucket, LP deposits and withdrawals can be blocked for almost a full bucketInterval before any swap has actually expired.

For example, if swapTerm = 1 day and bucketInterval = 1 hour, a swap opened at 12:59:59 expires at 12:59:59 the next day, but the gate begins blocking at 12:00:00. For the next 59 minutes and 59 seconds, supplyCollateral() and withdrawCollateral() reverts even though no expired swaps that can be settled exist.

The impact is that protocol actions are blocked earlier than intended. LPs can be prevented from depositing or withdrawing even though no swap has actually matured yet, and any integration that depends on those flows inherits the same premature denial of service.

RECOMMENDATION

Modify the LP gate to use the earliest exact unsettled swap expiry, not the floored expiry epoch boundary. This ensures supplyCollateral() and withdrawCollateral() remain available until a swap has actually matured.

RESOLUTION

Status: Resolved.

MEDIUM **M-06** ACKNOWLEDGED

LP adapter reverts on last-share withdraw

CATEGORY	Unexpected Behavior
LOCATION	KairosMarketAdapter.sol
REVIEW	Remediation Review 1

DESCRIPTION

KairosMarketAdapter assumes a partial withdrawCollateral call returns exactly the requested assets, so it leaves VaultV2 in pull mode for that amount after deallocate returns. That assumption is false when the adapter is the last LP in a market with active swaps. In that case SwapCore.withdrawCollateral can keep one share alive and reduce the actual withdrawal amount even though fullAmount is false.

```plain text
if (pool.totalShares == sharesToRedeem) {
if (pool.lockedCollateral > 0) {
sharesToRedeem -= 1;
withdrawalAmount = Math.mulDiv(sharesToRedeem, sharePrice, WAD);
}
}
```

The adapter then holds less than VaultV2 tries to pull with safeTransferFrom, so deallocate reverts. This breaks partial allocator withdrawals in a stressed market. It also breaks forceDeallocate, because that function reuses the same pull-after-deallocate behavior while KairosMarketAdapter blocks fullAmount = true. An allocator can still fall back to fullAmount = true through the normal deallocate` call, but the permissionless emergency unwind is still lost in this state.

RECOMMENDATION

Handle the realized amount instead of assuming the requested amount is always available on partial withdrawal. The safest fix is to switch to a push-based transfer whenever withdrawn != assets, or add a dedicated deallocation mode that reports and transfers the actual withdrawn amount back to the vault.

RESOLUTION

Status: Acknowledged.

MEDIUM M-07

ACKNOWLEDGED

Bucket PnL nets before swap caps

CATEGORY	Logical Error
LOCATION	`Utils.sol`
REVIEW	Remediation Review 1

DESCRIPTION

Final settlement is capped per swap. LP gains on a given swap cannot exceed that swap's buyer collateral, and LP losses cannot exceed that swap's pool collateral backing.

The bucket MtM path does not preserve those per-swap caps. In calculateBucketUnrealizedPnL(), the protocol first computes one net bucket PnL from averaged bucket parameters and only then caps that net result by the bucket's aggregate collateral totals. That is not equivalent to summing individually capped swap outcomes.

If one swap in the bucket is already far enough in LP profit that its payoff should be saturated at its own buyer-collateral cap, while another swap in the same bucket is LP-lossmaking, the bucket model can still let the first swap's uncapped excess offset the second swap before any cap is applied. Real settlement cannot do that, because the profitable swap would have been capped first.

For example, suppose two swaps are in the same bucket. Swap A would give LP +200, but its buyer collateral cap is 100, so realizable PnL is +100. Swap B would give LP -50, fully realizable. The true total is therefore +50.

But the bucket model can instead net first and produce uncapped bucket PnL of +150. If the aggregate buyer collateral in the bucket is at least 150, that full +150 survives the bucket cap even though actual realizable settlement is only +50.

Since bucket result feeds directly into getPoolSharePrice(), LP pricing and the Morpho LP adapter's realAssets() can diverge from realizable per-swap settlement.

RECOMMENDATION

M-07 Bucket PnL nets before swap caps CONTINUED

The key is to keep the per-swap collateral cap in place before positions are netted for any LP valuation that affects accounting. Right now the bucket model blends swaps together too early, so a single cap at the end is no longer the same as what would happen when those swaps actually settle.
One way to fix that is to make the aggregation more selective, so only swaps with similar payoff and cap behavior are grouped together before netting and capping. If that still leaves too much approximation, the better option is to stop using this bucket mark-to-market path for values that need to match real settlement, such as LP share price and adapter asset reporting, and use a more conservative valuation there instead.

RESOLUTION

Status: Acknowledged.

MEDIUM **M-08** RESOLVED

Mixed-expiry bucket misprices LP MtM

CATEGORY	Logical Error
LOCATION	`SwapCore.sol`
REVIEW	Remediation Review 1

DESCRIPTION

The LP share-price path decides whether a bucket is active or expired from the bucket's average entry time instead of from each swap's true expiry. In _resolveBucketExpiryIndices(), the protocol computes avgEntryTime = bucket.weightedEntryTime / bucket.lpNotional, then sets expiryTime = avgEntryTime + swapTerm, and only treats the bucket as expired once the current timestamp passes that single average-expiry timestamp.

That assumption is false whenever swaps inside the same bucket were opened at different times. In that case, some swaps can already be expired while others are still active, but the share-price path still forces the whole bucket into a single lifecycle state.

For example, if swapTerm = 30 days and bucketInterval = 1 day, two swaps can enter the same bucket 23 hours apart. Their real expiries are also 23 hours apart. During the window after the earlier swap has expired but before the bucket's average expiry is reached, the whole bucket is still treated as active. The already-expired subset therefore continues to follow the live oracle and active-bucket logic even though those swaps should already be frozen at their own expiry. The opposite window also exists on the other side of average expiry, where the still-active subset can be frozen too early.

The same root issue persists even after all swaps in the bucket are expired. Once the bucket is treated as expired, _resolveBucketExpiryIndices() still resolves only one historical floating index at the bucket's average expiry time and applies it to the entire bucket. If swaps inside that bucket actually expired at different times and the floating index moved between those expiries, exact settlement would freeze each swap at its own expiry index, but the LP MtM path instead freezes all of them at one intermediate average-expiry index.

As a result, getPoolSharePrice() can diverge from exact per-swap pricing during mixed expired/active bucket states, and any path that relies on that share price inherits the same accounting error.

RECOMMENDATION

M-08 Mixed-expiry bucket misprices LP MtM CONTINUED

Do not determine bucket lifecycle from avgEntryTime + swapTerm. Buckets that can contain swaps with different expiries need to track expired and active subsets separately, or the protocol must avoid using bucket-level MtM for accounting-critical pricing during mixed-expiry states.

RESOLUTION

Status: Resolved.

MEDIUM M-09

PARTIALLY RESOLVED

Liquidatable swaps ignore liquidator haircut

CATEGORY	Logical Error
LOCATION	`Utils.sol`
REVIEW	Remediation Review 1

DESCRIPTION

`calculateBucketUnrealizedPnL()` caps LP mark-to-market against the full posted collateral in `bucket.totalBuyerCollateral` or `bucket.totalPoolBacking`. That is no longer the realizable value once a swap is already liquidatable. At that point `isSwapLiquidatable()` has moved the economic cutoff down to `collateral - liquidatorReward` or `poolBacking - liquidatorReward`, and `liquidateSwap()` pays that missing slice to the liquidator instead of the winning side.

Consequently, `getPoolSharePrice()` can still overstate pool NAV during the liquidation window. LP minting, burning and any downstream integration that trusts LP share price can trade against value that is no longer collectible by the pool.

RECOMMENDATION

Once a swap is liquidatable, share pricing should stop treating the full posted collateral as collectible. Cap its value at the amount the winning side could actually recover after the liquidation incentive is deducted.

RESOLUTION

Status: Partially resolved.

LOW L-01

RESOLVED

Epoch pointer can clear too early

CATEGORY	Unexpected Behavior
LOCATION	`SwapCore.sol`
REVIEW	Remediation Review 1

DESCRIPTION

`_advanceEpochPointer()` assumes the next non-empty expiry epoch must appear within `swapTerm / bucketInterval` steps. That assumption is false once swaps with later expiries have been opened long after the earliest expiry. If the earliest epoch is cleared after a long delay, the scan can hit `maxIter` before it reaches the next populated epoch and will reset `earliestUnsettledEpoch[marketId]` to zero.
Once the pointer is zeroed, `_revertIfExpiredUnsettled()` no longer blocks LP deposits and withdrawals even though expired unsettled swaps still exist. The contract comments say those operations must be blocked while expired swaps remain unsettled because share price may be unreliable, so clearing the pointer early defeats the safety gate exactly when it is supposed to be active.

RECOMMENDATION

Track non-zero expiry epochs with an ordered data structure instead of a bounded linear scan. If a scan is kept, it must continue until the next populated epoch is found rather than clearing the pointer after an arbitrary hop count.

RESOLUTION

Status: Resolved.

LOW L-02 RESOLVED

Donated funds cannot be swept at zero allocation

CATEGORY	Warning
LOCATION	`KairosBuyerAdapter.sol, KairosMarketAdapter.sol`
REVIEW	Remediation Review 1

DESCRIPTION

Both Morpho adapters expose a sweep action intended to recover donated or accidentally transferred tokens from the adapter. However, under the Morpho VaultV2, every deallocation goes through deallocateInternal(), which reverts if the adapter ID's current recorded allocation is zero.
As a result, if tokens are transferred directly to an adapter while Morpho records caps\[id\].allocation == 0, the intended sweep path cannot be used immediately. The vault reverts before the adapter's returned change is applied, so the donation cannot be recovered through ACTION_SWEEP unless the adapter later has non-zero recorded allocation.

RECOMMENDATION

If this behavior is acceptable, explicitly document that ACTION_SWEEP cannot recover donated funds while the adapter has zero recorded allocation, so operators are aware of the edge case. Otherwise, add a separate recovery path for accidental token transfers that does not depend on Morpho's deallocate flow.

RESOLUTION

Status: Resolved.

LOW L-03

RESOLVED

Buyer cleanup can fail on unrelated bad oracle

CATEGORY	DoS
LOCATION	KairosBuyerAdapter.sol
REVIEW	Remediation Review 1

DESCRIPTION

`KairosBuyerAdapter.deallocate()` always recomputes `change = _allocationDelta(_realAssets())` before it returns. That is too strict for `ACTION_CLAIM_ESCROW` and `ACTION_REMOVE_SETTLED`. Those actions only need to forward already-realized funds or prune bookkeeping for a settled swap. They do not need fresh pricing for the rest of the portfolio.

The final `_realAssets()` call still walks every remaining unsettled swap and calls `swapCore.getSwapNetAmount()` on each of them. That valuation depends on healthy oracle reads. If some other tracked live swap is tied to a stale or invalid oracle, the cleanup transaction reverts even though the swap being cleaned up is already settled. Consequently, one unhealthy live position can block escrow recovery or removal of an unrelated settled swap.

RECOMMENDATION

Do not require a full live portfolio valuation to complete `ACTION_CLAIM_ESCROW` or `ACTION_REMOVE_SETTLED`. Let those cleanup actions return using the last reported allocation when fresh MtM is unavailable, then resync allocation later. Another valid option is to separate bookkeeping cleanup from NAV synchronization so an unhealthy oracle on one live position cannot freeze unrelated recovery work.

RESOLUTION

Status: Resolved.

LOW L-04

RESOLVED

Market creation fee can change before create

CATEGORY	Trust Assumptions
LOCATION	`Admin.sol`
REVIEW	Remediation Review 1

DESCRIPTION

`setMarketCreationFeeToken` and `setMarketCreationFeeAmount` take effect immediately. `SwapCore.createMarket` is permissionless and reads those values from `Admin` at execution time before it charges the caller. The creator does not provide any expected fee token or maximum fee amount in calldata.

Consequently, the owner can watch pending `createMarket` transactions and front-run them with a fee increase or a token switch. If the caller has enough ETH or ERC20 allowance for the new configuration, the market is still created but pays an unexpected fee.

The live read happens in `SwapCore.createMarket`, which loads `CREATE_MARKET_FEE_AMOUNT` and `CREATE_MARKET_FEE_TOKEN` and then transfers the configured fee to the multisig in the same call.

RECOMMENDATION

Do not let market-creation fee changes apply immediately to pending permissionless creates. The safest fix is to timelock `setMarketCreationFeeToken` and `setMarketCreationFeeAmount` so creators get reliable notice before charging changes take effect.

It would also be safer to let callers bound fee exposure directly. For example, `createMarket` can accept an expected fee token and a `maxFeeAmount`, then revert if the live config is different at execution time.

RESOLUTION

Status: Resolved.

INFORMATIONAL **I-01** PARTIALLY RESOLVED

Spot-compound projection can revert

CATEGORY	Math
LOCATION	`SwapFormulas.sol`
REVIEW	Remediation Review 1

DESCRIPTION

`indexFromCumulativeRateTimeCompound()` derives the live index with `expWad(crt / SECONDS_IN_YEAR)`. `expWad()` reverts once the exponent reaches its documented upper bound.
That makes `SpotCompoundRate` markets depend on an unstated operating limit for the accumulated `rate * time` term. Ordinary APRs should stay well inside the safe range, but a sufficiently high configured rate, or a long enough stale period under an unusually large rate, can push the exponent over the limit and make fresh-index reads revert. The affected paths are settlement-critical because `RateIndexLib.update()` and `SwapCore._getFreshIndex()` both depend on this projection for `SpotCompoundRate`.

RECOMMENDATION

Treat the exponent ceiling as a protocol constraint instead of a hidden math limit. Add an explicit guard before calling `expWad()`, and either clamp to a safe upper bound or revert with a domain-specific custom error that points operators to an invalid oracle configuration. If very large compounded rates must be supported, use an index representation that does not depend on a single exponential call over an unbounded accumulator.

RESOLUTION

Status: Partially resolved.

INFORMATIONAL I-02

ACKNOWLEDGED

Expired swap can freeze LP mobility

CATEGORY	Unexpected Behavior
LOCATION	`SwapCore.sol`
REVIEW	Remediation Review 1

DESCRIPTION

`buySwap()` does not check whether the market already has an expired unsettled swap. At the same time, `supplyCollateral()` and `withdrawCollateral()` both revert once `_revertIfExpiredUnsettled()` sees an expired head-of-queue swap. Consequently, a small expired position can force LPs to clear queue maintenance before they can move capital, while new buyer trades remain available against the market's remaining liquidity.

This is the case since settlement is permissionless but unrewarded. If the expired swap is economically negligible, nobody has a strong reason to spend gas settling it. The result is an asymmetric liveness failure: LPs lose immediate mobility, but buyers can still keep opening new swaps because `buySwap()` has no equivalent guard.

This does not permanently trap funds because any caller can settle the expired swap and unblock LP actions. The issue is that the protocol lets a cheap expired position externalize cleanup cost onto LPs while keeping one side of the market open.

RECOMMENDATION

Make the market-wide expired-unsettled rule consistent. Consider gating `buySwap()` with the same `_revertIfExpiredUnsettled()` check used by LP deposits and withdrawals. If buyer trading should remain available, add a direct incentive or automatic cleanup mechanism so expired queue heads do not rely on altruistic settlement.

RESOLUTION

Status: Acknowledged.

HIGH **H-01** RESOLVED

Direct settlement stale-prices vault shares

CATEGORY	Validation
LOCATION	`SwapCore.sol`
REVIEW	Main Review 2

DESCRIPTION

`SwapCore.makePayment()` is public and settles any supplied swap id. It does not distinguish ordinary user-owned swaps from swaps whose buyer is a Vault V2 adapter. Therefore any account can directly settle an adapter-owned Kairos swap without going through the vault allocator or the adapter's `deallocate()` logic.

`KairosBuyerAdapter._realAssets()` values open buyer swaps by reading `SwapCore.getSwapNetAmount()` and adding or subtracting the current mark-to-market amount from posted buyer collateral. Once the swap is settled, the same position is no longer valued as an open swap. The adapter instead counts the tokens it actually received, escrowed collateral, or zero if the settlement consumed the buyer collateral. This means a direct settlement can immediately move adapter NAV from a pre-settlement mark to a lower or higher realized value.

Morpho Vault V2 caches the first total-assets value used in a transaction. After `accrueInterest()` records that first value, later share mints or burns in the same transaction can still be priced from the stale snapshot. A shareholder can call `accrueInterest()`, directly settle a loss-making adapter-owned swap through `SwapCore.makePayment()`, then redeem at the pre-loss share price. The mirror case also exists: a depositor can settle a gain-making adapter-owned swap and mint shares at the pre-gain share price.

The value that can be extracted scales with the stale NAV gap and the attacker's share ownership. In the Base fork PoC below, the attacker captures about 30% of a `1,572,406.998739 USDC` stale NAV gap because they own about 30% of the Morpho vault shares at redemption time.

POC

The current POC is:

``bash
forge test --match-path test-foundry/zz_POCs/DirectSettlementCurrentLocalPOC.t.sol -vvvv
`

`The test is a Base fork PoC. It uses real Base USDC at` 0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913 `and a real Morpho Vault V2, Gauntlet USDC Prime at` 0x050cE30b927Da55177A4914EC73480238BAD56f0 `. It also uses the standard Base deployment oracle/config addresses from` deployments/base `: Morpho reference rate oracle` 0x2775289dEb423980AD2D7344850089Fb7F23963F`, Morpho USDC/cbBTC fixed-rate forecast oracle

H-01 Direct settlement stale-prices vault shares CONTINUED

`0x3b964A9820E5781C57D9809b521A0134E9C886A4` and the 90-day risk-premium wrapper `0x3Fac8aF279C78Eb82C263EFcacd01D01740Bcfc9`.
The PoC deploys the current audited local `Admin`, `SwapCore`, and `KairosBuyerAdapter` contracts onto the fork. This is necessary because the live Base `SwapCore` deployment artifact exposes an older ABI, while the audited adapter targets the current `buySwap(bytes32,uint256,address,int256,uint256)` and batched `makePayment(bytes32[])` interface. The fixed-rate oracle shim in the test only adapts the old live oracle read surface to the current local `ITenorRateOracle` interface; every rate read is forwarded to the real Base oracle.
No mock contracts are used. There are two fork setup details to make the current audited contracts usable with the selected real vault. First, the selected Gauntlet vault has abdicated adapter-registry changes, so the test uses one scoped `vm.mockCall` for the live registry's `isInRegistry(address)` read to model the production condition where the Kairos adapter has been accepted by the registry. Second, the live vault auto-allocates ordinary deposits into its liquidity adapter, leaving zero idle USDC. The test seeds an idle Base USDC balance on the real vault to model the realistic state after a keeper or allocator deallocation leaves liquidity available before a manual Kairos allocation.
The exploit transaction is executed by an unprivileged shareholder contract. It calls `vault.accrueInterest()` while the Kairos swap is still open, directly calls `SwapCore.makePayment([swapId])`, observes that the adapter's realized NAV is lower than the open-swap mark, then redeems the attacker's shares before Vault V2 refreshes the transaction's first total-assets snapshot.
The passing Base fork run produced:

```plain text
fork chain id = 8453
real vault = Gauntlet USDC Prime (gtusdcp)
real vault address = 0x050cE30b927Da55177A4914EC73480238BAD56f0
real asset = Base USDC
swap notional = 200,000,000.000000 USDC
assets pushed to adapter = 10,000,000.000000 USDC
pre-settlement vault assets = 247,763,986.344113 USDC
pre-settlement adapter NAV = 4,890,144.352831 USDC
adapter NAV after settlement = 3,317,737.354092 USDC
actual post-settlement NAV = 246,191,579.345374 USDC
stale NAV gap = 1,572,406.998739 USDC
stale redeem amount = 74,657,537.417348 USDC
fair post-settlement redeem = 74,183,731.534274 USDC
excess extracted = 473,805.883074 USDC
```

The trace confirms that SwapCore.makePayment()` is called by the attacker contract, emits

H-01 Direct settlement stale-prices vault shares CONTINUED

`SwapClosed`, and settles the adapter-owned swap. The following `vault.redeem()` burns the attacker's shares and transfers `74,657,537.417348 USDC`, even though the fair post-settlement value is `74,183,731.534274 USDC`.

RECOMMENDATION

Add explicit settlement authorization at the `SwapCore` position level. When a swap is opened for a managed integration such as a Vault V2 adapter, store a nonzero settlement controller, for example the adapter address. `makePayment()` should only remain permissionless for swaps that were created with permissionless settlement enabled. For controlled swaps, `makePayment()` should require `msg.sender` to be the recorded settlement controller or another explicitly authorized settlement executor.
`KairosBuyerAdapter` should open Vault V2-owned swaps with restricted settlement, then settle those swaps only during allocator-controlled `VaultV2.deallocate()` accounting. Do not allow permissionless `VaultV2.forceDeallocate()` to execute `ACTION_MAKE_PAYMENT` or any other NAV-changing settlement action. Morpho Vault V2 intentionally exposes `forceDeallocate()` as a permissionless user function, so routing settlement through the adapter is not enough if the adapter still accepts `IVaultV2.forceDeallocate.selector` for settlement.
Do not rely on a generic fresh `accrueInterest()` call as the mitigation. The issue exists because Vault V2 caches the first total-assets value in the transaction. Once that snapshot is set, later adapter NAV changes must either be impossible before same-transaction share mints or burns or the share action must be blocked/deferred.

PROOF OF CONCEPT https://gist.github.com/GuardianAudits/9a05cad601f0f214e06bb476f97c350e

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/761

MEDIUM **M-01** ACKNOWLEDGED

Fresh LP deposits underprice swap fees

CATEGORY	Unexpected Behavior
LOCATION	`SwapCore.sol`
REVIEW	Main Review 2

DESCRIPTION

`buySwap()` prices the utilization fee from the pool's current collateral figures. Fresh LP deposits count immediately in both available liquidity and total collateral.

The only waiting period is enforced later, in `withdrawCollateral()`. It blocks a withdrawal in the same block as the deposit, but the LP can leave in the next block if the remaining pool still covers the locked collateral.

That lets a buyer add temporary LP liquidity, open a swap at a cheaper utilization fee, then withdraw the temporary liquidity one block later. The swap stays open. The remaining LPs keep the exposure, but the fee was calculated as if the temporary liquidity would remain available for the swap term.

The PoC compares two 5,000,000 USDC swaps. With only 1,000 USDC of existing LP collateral, the control swap records a 0.3604222096130042% utilization fee, a 5.4604222096130042% swap rate, and 720.980533 USDC of LP backing. When another LP adds 9,000 USDC immediately before the same-size swap, the recorded utilization fee falls to 0.0360490266762789% and the swap rate falls to 5.1360490266762789%, while LP backing remains the same 720.980533 USDC. After one block, the temporary LP withdraws the full 9,000 USDC. Pool total collateral returns to 1,000 USDC, but the underpriced swap remains open with 720.980533 USDC locked against it.

In the validated `BUY_FIXED` case, the buyer is the winner. The control swap requires 787.371623 USDC of buyer collateral plus a 39.368581 USDC liquidation bounty, for 826.740204 USDC paid into the contract. The temporarily priced swap requires only 740.598273 USDC of buyer collateral plus a 37.029913 USDC liquidation bounty, for 777.628186 USDC total. The buyer pays 49.112018 USDC less up front and also locks a lower fixed rate for the same 5,000,000 USDC notional. The losing side is the remaining LP base. Once the temporary LP exits, the legacy LP collateral is still backing 720.980533 USDC of locked exposure, but the swap was priced at the lower utilization fee created by liquidity that is no longer in the pool.

RECOMMENDATION

M-01 Fresh LP deposits underprice swap fees CONTINUED

Do not let freely withdrawable LP collateral reduce the utilization fee for swaps that can remain open after that collateral leaves. Liquidity that counts toward swap pricing should either stay committed to the exposure it helped price, or it should be excluded from `availableLiquidity` and `totalLiquidity`. Track pricing-eligible liquidity separately from raw `pool.totalCollateral` and `totalLPAvailableCollateral()`. A withdrawal queue or pro-rata liquidity reservation would all address the issue better than a simple deposit aging period.

PROOF OF CONCEPT https://gist.github.com/GuardianAudits/12e8d168317f640cc68b28e61f176ff8

RESOLUTION

Status: Acknowledged.

MEDIUM **M-02** RESOLVED

Morpho adapters accept stale indexes

CATEGORY	Validation
LOCATION	CumulativeIndexBorrowMorpho.sol
REVIEW	Main Review 2

DESCRIPTION

`CumulativeIndexBorrowMorpho.getCumulativeIndex()` tries to project Morpho borrow assets forward when the Morpho market has not accrued in the current block.
If projection fails, `_tryProject()` returns the old `borrowAssets` value:

```plain text
try IIrm(params.irm).borrowRateView(params, m) returns (uint256 rate) {
uint256 interest = borrowAssets * _wTaylorCompounded(rate * elapsed) / WAD;
return borrowAssets + interest;
} catch {
return borrowAssets; // IRM failed - fall back to stale
}
```

The caller then converts that stale asset/share ratio into an index and returns it as valid:

```plain text
uint256 index = (borrowAssets + VIRTUAL_ASSETS) * 1e27
/ (borrowShares + VIRTUAL_SHARES);
return (index, index > 0);
```

That is the wrong failure mode for a settlement oracle. If the adapter knows the Morpho market is stale but cannot compute the current projection, it should not mark the old index as safe to use. RateIndexLib.update(), SwapCore._getFreshIndex() and getSwapNetAmount() trust the isValid flag. As a result, a Morpho cumulative-index market can keep pricing or settling swaps from an old index exactly when the adapter failed to accrue the index to the current timestamp.
CumulativeIndexSupplyMorpho has the same pattern. Its projection helper returns the original supplyAssets and supplyShares when idToMarketParams() or borrowRateView() fails, then getCumulativeIndex() returns the stale share ratio with isValid = true`.

M-02 Morpho adapters accept stale indexes CONTINUED

RECOMMENDATION

Do not report a failed projection as valid.
If `block.timestamp > m.lastUpdate` and the adapter cannot compute the projection, return `(0, false)` or revert with a clear oracle error. Settlement-critical callers should reject that data instead of silently using the old Morpho share ratio. If a stale fallback is intentionally allowed, make it explicit and bounded. For example, cap the allowed staleness window and expose whether the returned index is current, projected or stale so callers can choose whether to accept it.

RESOLUTION

Status: Resolved.

MEDIUM M-03 RESOLVED

Buyer adapter cleanup erases live allocation

CATEGORY	Logical Error
LOCATION	KairosBuyerAdapter.sol
REVIEW	Main Review 2

DESCRIPTION

`KairosBuyerAdapter.deallocate()` has a special case for swaps that were already settled directly through `SwapCore`. It forwards `settlementPayouts(sid)` to the parent vault, removes the settled swap from tracking, and returns only a targeted payout delta. When the payout is larger than `lastReportedAllocation`, the adapter sets `lastReportedAllocation` to zero.

That loses the value of any other tracked swaps. A profitable externally settled swap can pay out more than the adapter's previous total reported allocation because the old allocation only counted buyer-side collateral and mark-to-market value from the last adapter interaction. If another swap is still tracked, `_realAssets()` can remain positive after the payout is forwarded, but the adapter has already told the vault that the allocation for `adapterId` is zero.

Consequently, Morpho cap accounting can understate the vault's remaining Kairos buyer exposure. Allocators can make later decisions against a zero recorded allocation even though the adapter still owns a live swap with recoverable value. This weakens the cap invariant that the adapter's returned `change` keeps the vault's stored allocation aligned with the adapter's current position value.

This is the reintroduced buyer-adapter side of [M-02](https://www.notion.so/guardianaudits/M-02-31e8bda5828c81a8ab71c621ea5184d1?v=3178bda5828c818f8c9c000c8a2b48ea&source=copy_link). The M-02 fix made cleanup use `_allocationDelta(_realAssets())`, but the later cleanup change for [L-03](https://www.notion.so/guardianaudits/L-03-33e8bda5828c816087cffeffbd2395d7?v=3178bda5828c81db8475000c828caca3&source=copy_link) replaced the already-settled accounting with a targeted payout delta to avoid stale-oracle reverts. That fix resolved the L-03 liveness issue while reopening the M-02 allocation invariant for profitable externally settled swaps.

RECOMMENDATION

Make the already-settled cleanup update allocation from the adapter's remaining value, not only from the settled swap's payout. The safest fix is to recompute `newValue = _realAssets()` after forwarding the payout and removing the settled swap, then return

M-03 Buyer adapter cleanup erases live allocation CONTINUED

`int256(newValue) - int256(oldValue)` and store `newValue`. If this branch must avoid a full portfolio valuation during oracle outages, store per-swap reported value so cleanup can subtract only the settled swap's previous contribution. Do not let one payout clamp the entire adapter allocation to zero while other tracked swaps remain.

RESOLUTION

Status: Resolved.

MEDIUM M-04 ACKNOWLEDGED

Morpho share ratios can reset cumulative indexes

CATEGORY	Configuration
LOCATION	`CumulativeIndexBorrowMorpho.sol`, `CumulativeIndexSupplyMorpho.sol`
REVIEW	Main Review 2

DESCRIPTION

`CumulativeIndexBorrowMorpho` and `CumulativeIndexSupplyMorpho` expose Morpho Blue share exchange rates as Kairos cumulative indexes. That is not a safe cumulative-index source when the underlying Morpho market can become empty or close to empty.

Both adapters calculate the index from live Morpho totals and the Morpho virtual offsets:

```plain text
uint256 index = (borrowAssets + VIRTUAL_ASSETS) * 1e27
/ (borrowShares + VIRTUAL_SHARES);
```

and:

```plain text
uint256 index = (supplyAssets + VIRTUAL_ASSETS) * 1e27
/ (supplyShares + VIRTUAL_SHARES);
```

VIRTUAL_ASSETS `is` 1 `and` VIRTUAL_SHARES `is` 1e6`. Therefore, when the relevant Morpho side is emptied, the adapter does not return an invalid value. It returns` 1e21 `with` isValid = true`. If the same market previously had a higher share price from accrued interest, a full repayment or full withdrawal can make Kairos observe a sharp decrease in what it treats as a cumulative floating index. This breaks the core assumption behind` RateConvention.Cumulative`. Kairos stores the adapter value at swap entry and later derives the floating payment from the ratio between the entry index and the settlement index. A stateless Morpho share ratio can move because the market's assets and shares were burned down to zero, not because the realized floating rate became negative. Consequently, a swap can settle from an artificial index drop. For example, the borrow adapter can report about` 1.05e27 `while the Morpho borrow side has` 1,050,000e18 `borrow assets and` 1,000,000e18 `borrow shares. After all borrow shares are repaid, the same adapter reports` 1e21` and still marks the value valid. Kairos' index-ratio math then interprets that reset as a very large negative floating rate over the swap term. The supply adapter has the same structural problem when the supply side can be

M-04 Morpho share ratios can reset cumulative indexes CONTINUED

withdrawn to zero.
The near-empty case is also unsafe. When borrow or supply shares are small relative to the `1e6` virtual shares, the virtual offsets dominate the ratio. Small changes in assets, shares or rounding can then create large percentage moves in the reported index. This makes new, quiet or attacker-dominated Morpho markets unsuitable as settlement indexes for Kairos markets.
Due to this issue, Morpho share exchange rate is not a persistent cumulative index unless the market is guaranteed to remain deep, organic and non-empty for the whole Kairos swap lifecycle.

RECOMMENDATION

Do not use raw Morpho share exchange rates as stateless cumulative index oracles unless the protocol enforces strong non-empty and minimum-liquidity assumptions.
At minimum, both adapters should reject empty and dust states. For example, return `(0, false)` when `totalBorrowShares` or `totalSupplyShares` is below a configured minimum for the corresponding adapter. The minimum should be large enough that Morpho's virtual offsets cannot materially dominate the exchange rate.
A safer design is to maintain a stateful monotonic index wrapper for Morpho markets. The wrapper should store the last accepted index and reject decreases except for a tightly bounded rounding tolerance. If the protocol cannot enforce monotonicity on top of the Morpho share ratio, these adapters should not be eligible as `RateConvention.Cumulative` sources for settlement-critical Kairos markets.

RESOLUTION

Status: Acknowledged.

MEDIUM M-05 ACKNOWLEDGED

Buyer entry fees can be socialized

CATEGORY	Unexpected Behavior
LOCATION	`KairosBuyerAdapter.sol`
REVIEW	Main Review 2

DESCRIPTION

`KairosBuyerAdapter.allocate()` lets a Morpho Vault V2 allocator use vault funds to buy a Kairos buyer swap. By the time the adapter runs, the vault has already transferred `assets` to it. The adapter then calls `swapCore.buySwap(...)`, which pulls buyer collateral, the prefunded liquidation bounty and the protocol fee from the adapter.

The collateral and bounty still belong economically to the vault position. They are counted again when the adapter reports `_realAssets()`. The protocol fee is different. It is transferred out during `SwapCore.buySwap()`, so the vault loses that value as soon as the swap is opened.

That immediate loss can be pushed onto users who stay in the vault. Morpho Vault V2 accrues before it calls the adapter. If an allocator also holds vault shares, or coordinates with a shareholder, the same transaction can buy a fee-bearing swap and then redeem shares while the vault is still using the pre-fee total-assets snapshot. The redeemer exits before their share of the entry fee is reflected in the share price. The remaining depositors absorb the shortfall.

This is a bounded issue, not a full-vault drain. The fee is capped by `Admin.MAX_PROTOCOL_FEE_RATE`, but it can still be meaningful for large or long-dated swaps because it is computed from notional and term:

```plain text
protocolFee = notionalAmount * feeRate * swapTerm / YEAR / WAD;
```

At the maximum 3 percent annual fee, a 30-day swap pays about 0.246 percent of notional, while a one-year swap pays 3 percent. The value shifted to remaining depositors is roughly the fee paid multiplied by the redeemer's share of the vault. The required conditions are allocator control or allocator coordination, same-transaction batching and enough vault liquidity for the redeem.

RECOMMENDATION

Treat fee-bearing buyer allocations as NAV-changing actions that cannot be mixed with same-transaction share exits priced from an older total-assets snapshot. Consider making

M-05 Buyer entry fees can be socialized CONTINUED

vault-owned buyer swaps fee-free or keep the fee small enough that the loss is explicitly negligible for Morpho Vault V2 integrations.
If vault-owned buyer swaps must pay material fees, block share redemptions in the same transaction as those allocations or enforce an allocator policy that prevents fee-paying allocations from being batched with exits by the allocator or a cooperating shareholder. Curators should not enable this adapter for fee-bearing buyer swaps unless that policy is in place.

RESOLUTION

Status: Acknowledged.

MEDIUM **M-06** RESOLVED

Poke spam blocks TWAR ring advancement

CATEGORY	Unexpected Behavior
LOCATION	IndexBaseRateV1.sol
REVIEW	Main Review 2

DESCRIPTION

`IndexBaseRateV1.poke()` decides whether to advance the ring buffer by comparing the current timestamp to `observations[head].timestamp + minSpacingSec`. If the call arrives too early, the function does not advance the ring. It still overwrites the head observation with the current timestamp.

Consequently, a caller can repeatedly call `poke()` just before `minSpacingSec` elapses. Each non-advancing call refreshes `observations[head].timestamp` and pushes the next advancement threshold forward. With the reviewed deployment configuration of `twarWindowSec == minSpacingSec == 12 hours`, this can keep the ring from adding new historical anchors while the oracle continues to accept permissionless pokes.

Reads then fall back to older observations when `_realizedTwar()` searches for the youngest anchor at or before `block.timestamp - twarWindowSec`. The returned rate can be based on a long interpolation chord rather than the intended trailing TWAR window. Since this oracle can be used as a base-rate source, degraded quotes can affect swap entry pricing, collateral requirements, early exits and LP mark-to-market.

RECOMMENDATION

Do not let non-advancing pokes reset the ring-advancement clock. The simplest fix is to ignore pokes before `minSpacingSec` has elapsed or update only the head index while preserving the original head timestamp. If replacement pokes are still needed, track a separate `lastAdvancedTimestamp` and base ring advancement on that value instead of the replaceable head observation timestamp. Limit in-place replacement to a narrow deduplication window, such as same-block updates, so permissionless callers cannot indefinitely delay new ring slots.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/777

MEDIUM M-07 RESOLVED

Permissionless updates starve rate snapshots

CATEGORY	Logical Error
LOCATION	RateIndexLib.sol
REVIEW	Main Review 2

DESCRIPTION

`RateIndexLib._appendSnapshot()` overwrites the latest snapshot whenever a new update is less than `MIN_SNAPSHOT_GAP` seconds after the previous snapshot. That is unsafe because `SwapCore.updateMarketRateIndex()` is permissionless. A caller can keep calling it every 11 seconds and repeatedly move the last snapshot forward instead of letting the snapshot array grow. This can erase the only historical anchor for an oracle. The dangerous case is when the overwrite pattern continues after expiry. Once the earliest stored snapshot has been moved past the target expiry, `getIndexAt()` treats the target as being before the first snapshot and returns that later snapshot value.

Example sequence:

1. A market is initialized at `12:00:00`, and the first snapshot is stored.
2. A caller updates the market every 11 seconds. Each update overwrites the same snapshot slot instead of appending a new one.
3. A swap expires at `12:00:30`.
4. The caller keeps updating after expiry, so the only retained snapshot continues moving forward.
5. By `12:10:00`, the only retained snapshot is post-expiry.
6. Settlement calls `getIndexAt(12:00:30)`. Since `12:00:30` is earlier than the first retained snapshot, the function returns the `12:10:00` index.

That is the post-expiry contamination this issue flags. The bug is not that every sparse snapshot set uses future data. The bug is that permissionless updates can keep moving the retained snapshot until missing history is silently replaced with post-expiry data.

RECOMMENDATION

Do not use a time-window overwrite rule for economically meaningful snapshots. If deduplication is needed, only replace snapshots written in the same block:

```plain text
if (len > 0 && snaps[len - 1].timestamp == uint48(block.timestamp)) {
snaps[len - 1] = Snapshot(uint48(ts), uint208(val));
} else {
```

M-07 Permissionless updates starve rate snapshots CONTINUED

```
snaps.push(Snapshot(uint48(ts), uint208(val)));
}
```

Also make historical lookup fail closed when the target timestamp is earlier than the earliest retained snapshot. Returning the first retained snapshot for an older target silently turns missing history into future pricing data.

RESOLUTION

Status: Resolved.

LOW L-01 RESOLVED

Buyer adapter counts liquidator bounty in NAV

CATEGORY	Logical Error
LOCATION	`KairosBuyerAdapter.sol`
REVIEW	Main Review 2

DESCRIPTION

`KairosBuyerAdapter._realAssets()` values each unsettled buyer swap as `collateralBalance + liquidationBounty`, adjusted by the current net PnL from `SwapCore.getSwapNetAmount()`.
That is not conservative once the buyer side is liquidatable. At that point the bounty is no longer recoverable by the vault, it is reserved for the liquidator. Since `getSwapNetAmount()` caps the buyer's loss at `collateralBalance`, the adapter can still report the prefunded bounty as NAV even though `liquidateSwap()` will pay that same amount away.
This overstates the Morpho vault's adapter value and allocation. If the vault has other liquid assets, users can redeem against a share price that includes value the strategy will not recover. The loss is then left with the remaining depositors when liquidation clears the swap.
The impact is bounded to the bounty on each affected swap, but it can scale with the number and size of buyer-adapter positions.

RECOMMENDATION

Value buyer swaps using the same recoverable amount that `SwapCore` would pay if the position were closed. Consider exposing a conservative `SwapCore` view for buyer recovery and have `_realAssets()` use it. That view should apply the same caps and liquidation rules as settlement and liquidation. At minimum, `_realAssets()` should check whether the buyer side is liquidatable before adding `liquidationBounty`. If liquidation can take the bounty, exclude it from NAV.

RESOLUTION

Status: Resolved.

LOW L-02 RESOLVED

Expired settlement still needs live oracle

CATEGORY	DoS
LOCATION	SwapCore.sol
REVIEW	Main Review 2

DESCRIPTION

`_settleSwaps()` updates the live reference-rate oracle before it checks whether an expired swap already has a usable maturity index.

```plain text
uint256 oracleIndex = rateIndex.update(market.referenceRateOracle);

uint256 expiryIndex = 0;
if (!swap.isEarlyExit) {
uint256 actualExpiry = swap.entryTimestamp + market.swapTerm;
if (block.timestamp >= actualExpiry) {
expiryIndex = rateIndex.getIndexAt(market.referenceRateOracle, actualExpiry);
if (expiryIndex == 0) expiryIndex = oracleIndex;
}
}
```

That order is wrong for ordinary expiry settlement. Once the maturity index has been recorded, settlement should be able to use the historical value without depending on the current oracle. Today, if the oracle becomes stale or starts reverting after maturity, rateIndex.update() reverts before getIndexAt() is reached. The swap is already expired and can be priced from the stored snapshot, but it still cannot settle. This also blocks LP mobility. An expired unsettled swap at the queue head keeps withdrawCollateral()` and related LP flows blocked. A current oracle outage can therefore freeze withdrawals and vault cleanup even though the expired swap's value is already known.

RECOMMENDATION

L-02 Expired settlement still needs live oracle CONTINUED

For normal expiry settlement, read the maturity index first. If `getIndexAt(referenceRateOracle, actualExpiry)` returns a valid value, settle with that value and skip the live `rateIndex.update()`. Only use a fresh oracle update when the historical index is missing and the protocol explicitly wants live-oracle fallback pricing. If no fallback should be allowed, revert with a clear missing-expiry-index error instead. Early exits and liquidations can still require a fresh oracle because they price current state. Expiry settlement should not require it once the maturity snapshot exists.

RESOLUTION

Status: Resolved.

LOW L-03

ACKNOWLEDGED

SpotCompound floor blocks settlement

CATEGORY	DoS
LOCATION	SwapFormulas.sol
REVIEW	Main Review 2

DESCRIPTION

`SwapFormulas.deriveRateFromIndex()` computes the `SpotCompoundRate` log input from a WAD-scaled index ratio:

```plain text
int256 indexRatio = safeCast(Math.mulDiv(closingFloatingIndex, WAD, entryFloatingIndex));

if (convention == Types.RateConvention.SpotCompoundRate) {
int256 lnRatio = FixedPointMathLib.lnWad(indexRatio);
return (lnRatio * int256(SECONDS_IN_YEAR)) / int256(timeElapsed);
}
```

That ratio can round to zero even when closingFloatingIndex is nonzero. A normal swap enters around INITIAL_INDEX (1e27). If the closing snapshot is floored to MIN_INDEX (1e6), the ratio becomes:

```plain text
1e6 * 1e18 / 1e27 == 0
```

The next line calls lnWad(0), which reverts. As a result, a SpotCompoundRate swap with a deeply floored expiry index cannot be quoted or settled. If that swap is expired and still unsettled, it can also keep LP withdrawals blocked through the expired-unsettled guard.

RECOMMENDATION

Do not let a nonzero closing index produce a zero log input. Compute the compound index ratio at higher precision before calling `lnWad()`, or derive the compound settlement rate directly from cumulative rate-time snapshots instead of a rounded WAD-scaled index ratio. If `MIN_INDEX` remains part of the index model, set it high enough that every supported entry index produces a nonzero ratio. Settlement code should also reject floored historical indexes with a clear protocol error instead of reaching `lnWad(0)`.

L-03 SpotCompound floor blocks settlement CONTINUED

RESOLUTION

Status: Acknowledged.

LOW L-04 RESOLVED

Same-block cumulative index is stale

CATEGORY	Logical Error
LOCATION	RateIndexLib.sol
REVIEW	Main Review 2

DESCRIPTION

`RateIndexLib.update()` returns the cached index when the oracle was already updated in the same block:

```plain text
uint256 elapsed = block.timestamp - index.lastUpdates[oracle];
if (elapsed == 0) return index.indices[oracle];

uint256 newIndex;
Types.RateConvention convention = index.conventions[oracle];

if (convention == Types.RateConvention.Cumulative) {
(uint256 ci, bool isValid) = ICumulativeRateOracle(oracle).getCumulativeIndex();
if (!isValid || ci == 0) revert Errors.E606();
newIndex = ci;
}
```

`That early return happens before the code checks whether the oracle uses` RateConvention.Cumulative `. For spot-rate modes this is reasonable because no time has elapsed. For cumulative oracles, the source index can still change inside the same block if the adapter reads live state from another protocol.` buySwap() `stores the value returned by` rateIndex.update() `as` entryFloatingIndex`. If the cumulative oracle has already moved in the same block, a new swap can record the old cached index as its entry value. Later quotes then treat the pre-entry index move as if it happened after the swap opened.

RECOMMENDATION

Do not return the cached value for cumulative oracles only because `elapsed == 0`.
Move the convention check before the same-block early return. For `RateConvention.Cumulative`, read `ICumulativeRateOracle.getCumulativeIndex()` even in the same block, validate `isValid` and update the cached index when the external index has changed.
If same-block snapshot deduplication is still desired, `_appendSnapshot()` can keep overwriting the current timestamp snapshot. The important rule is that state-changing calls such

L-04 Same-block cumulative index is stale CONTINUED

as `buySwap()` must not store an older cumulative index when the oracle already reports a newer one.

RESOLUTION

Status: Resolved.

LOW L-05 RESOLVED

Compound adapter accepts stale index

CATEGORY	Unexpected Behavior
LOCATION	CumulativeIndexBorrowCompound.sol
REVIEW	Main Review 2

DESCRIPTION

`CumulativeIndexBorrowCompound.getCumulativeIndex()` reads Compound's stored `baseBorrowIndex` and projects it forward when `lastAccrualTime` is older than the current timestamp.
Projection is best-effort. If either Compound read fails, `_tryProject()` returns the old index:

```plain text
function _tryProject(uint256 borrowIndex, uint256 elapsed) private view returns (uint256) {
try comet.getUtilization() returns (uint256 utilization) {
try comet.getBorrowRate(utilization) returns (uint256 borrowRate) {
return borrowIndex + borrowIndex * borrowRate * elapsed / FACTOR_SCALE;
} catch {
return borrowIndex;
}
} catch {
return borrowIndex;
}
}
```

The caller then scales that stale value and returns it as valid:

```plain text
return (borrowIndex * SCALE_FACTOR, true);
```

That is unsafe for a cumulative settlement oracle. If the adapter knows the Compound market is stale but cannot compute the current projection, it should not tell SwapCore the old index is valid. SwapCore trusts the isValid flag for cumulative oracles. Quotes, liquidation checks, LP mark-to-market and settlement can therefore use an old Compound borrow index exactly when the adapter failed to compute current accrual.
This is separate from the zero-debt Compound issue. Here totalBorrowBase` is nonzero and projection would be valid if the Compound calls succeeded. The bug is that a failed projection is silently downgraded to a stale valid index.

L-05 Compound adapter accepts stale index CONTINUED

RECOMMENDATION

Do not report a failed projection as valid.
If `elapsed > 0` and either `getUtilization()` or `getBorrowRate()` fails, return `(0, false)` or revert with a clear oracle error. Settlement-critical callers should reject that read instead of pricing from the old stored borrow index.
If a stale fallback is intentionally allowed, make it explicit and bounded. For example, cap the maximum accepted staleness and expose whether the returned value is current, projected or stale.

RESOLUTION

Status: Resolved.

LOW L-06

RESOLVED

Idle LP withdrawals need live oracle

CATEGORY	Unexpected Behavior
LOCATION	SwapCore.sol
REVIEW	Main Review 2

DESCRIPTION

`withdrawCollateral()` always calls `rateIndex.update(market.referenceRateOracle)` and then `getPoolSharePrice()` before it decides how many shares to burn. This happens even when `pool.lockedCollateral == 0`, so no active swap needs mark-to-market pricing and a full withdrawal can be computed from idle pool collateral.

If the reference-rate oracle reverts or is invalid after LP collateral has been supplied, an LP cannot withdraw from an idle market. The funds remain in `SwapCore` until the oracle recovers, even though the market has no open exposure and no settlement price is needed to return the collateral. `supplyCollateral()` has the same unnecessary update before its zero-share branch, but the locked-fund case is the LP withdrawal.

RECOMMENDATION

Skip reference-oracle updates when no active swaps are locked. For idle pools, redeem shares directly against `pool.totalCollateral / pool.totalShares` or use cached accounting that does not call the reference oracle. Keep the fresh oracle update only when `pool.lockedCollateral > 0` and mark-to-market pricing can affect share value.

RESOLUTION

Status: Resolved.

LOW L-07

RESOLVED

Market creation skips base batch API check

CATEGORY	Validation
LOCATION	SwapCore.sol
REVIEW	Main Review 2

DESCRIPTION

`_initializeMarket()` validates the base swap rate oracle by calling only `getRateByTenor(params.swapTerm)`. Later pricing code uses the batch interface through `Utils.getBaseRateOracleBatch()`, which calls `ITenorRateOracle.getRates(tenors)` and reverts if that call fails or returns invalid. A base oracle can therefore pass market creation while still being incompatible with the functions used by normal market accounting. This can happen with a legacy or custom oracle that implements `getRateByTenor()` correctly but does not implement `getRates()` or returns invalid batch results. Once such a market exists, operations that need base rates for multiple tenors can revert. That can block LP actions, buyer actions, or settlement-adjacent accounting until the market is migrated or the oracle is replaced through an available administrative process.

RECOMMENDATION

Validate the same oracle interface that production accounting uses. During market creation, call `getRates()` with the exact tenor set that downstream pricing requires and reject the oracle unless it returns the expected number of rates with `isValid = true`.
If single-tenor oracles should remain supported, add an adapter that implements the full `ITenorRateOracle` batch interface by safely looping over the underlying single-tenor calls.

RESOLUTION

Status: Resolved.

LOW **L-08** RESOLVED

New bounds leave stored rates valid

CATEGORY	Validation
LOCATION	`BaseTimelockedOracle.sol`
REVIEW	Main Review 2

DESCRIPTION

`BaseTimelockedOracle` validates bounds only when new rates are posted. `_validateCoreRateConstraints()` rejects a submitted rate outside `minRateBoundWad` and `maxRateBoundWad`, but `_getRatesInternal()` later returns already-stored rates without checking them against the currently active bounds.

When the owner queues and activates tighter rate bounds, fresh stored rates that now violate the new bounds can continue to be served as valid until they become stale or until a new post is accepted. This breaks the expected safety property of the bound change. The system can keep using a rate that governance has already declared out of bounds. The issue affects all child oracles that inherit this read logic, including fixed-rate pricing and risk-rate oracles. Those values feed market creation, pricing, and mark-to-market calculations through the tenor-rate oracle interface.

RECOMMENDATION

Recheck active bounds in `_getRatesInternal()` before returning `true`. If any stored or interpolated rate is outside the currently active bounds, return invalid or revert with a clear error. When activating a tighter bound, also consider invalidating stored buckets that are outside the new range. This makes the change take effect immediately instead of waiting for staleness.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/779`

LOW L-09 RESOLVED

Buyer adapter drops early-exit slippage

CATEGORY	Validation
LOCATION	KairosBuyerAdapter.sol
REVIEW	Main Review 2

DESCRIPTION

`SwapCore.exitSwapEarly()` supports a per-swap `minExitAmount`. The buyer adapter does not expose that protection. For every adapter-driven early exit, it builds a single `ExitRequest` with `minExitAmount` set to zero.

Consequently, a vault allocator cannot tell the adapter to reject an early exit that returns too little value. The core contract can enforce the bound, but the adapter always disables it before calling `swapCore.exitSwapEarly()`. This leaves vault execution exposed to unfavorable oracle movement, rate movement or stale off-chain quotes between the allocator's decision and the transaction that reaches the chain.

RECOMMENDATION

Include `minExitAmount` in the adapter action data for `ACTION_EARLY_EXIT` and pass it through to `SwapCore.exitSwapEarly()`. If existing deployments need backward compatibility, support the current two-word payload as the unsafe legacy form and add a longer payload that carries the slippage bound. Vault allocators should be able to make early exit conditional on a minimum realized payout, the same way direct `SwapCore` users can.

RESOLUTION

Status: Resolved.

LOW L-10 RESOLVED

Market adapter leaves withdrawal surplus

CATEGORY	Unexpected Behavior
LOCATION	KairosMarketAdapter.sol
REVIEW	Main Review 2

DESCRIPTION

`KairosMarketAdapter.deallocate()` assumes a partial withdrawal from `SwapCore.withdrawCollateral()` returns exactly the `assets` amount that Morpho Vault V2 will pull from the adapter. That is not always true for the last LP.

`SwapCore.withdrawCollateral()` can round a partial withdrawal up to all remaining shares. If the pool has no locked collateral, it then sweeps `pool.totalCollateral` to the receiver rather than returning the requested `assets` amount. The adapter keeps partial withdrawals in pull mode, so the vault only pulls `assets`. Any surplus stays as a raw token balance on the adapter.

That surplus is not reported by `KairosMarketAdapter._realAssets()`, because `_realAssets()` only values remaining LP shares. Consequently, the adapter can hold vault-owned tokens while reporting zero assets. The funds are not immediately stolen, and an allocator can later sweep them, but the vault's adapter accounting is understated and the funds can remain stranded operationally.

RECOMMENDATION

Make partial withdrawals enforce the adapter's exact-amount assumption:

```plain text
if (!fullAmount && withdrawn != assets) revert UnexpectedWithdrawalAmount(withdrawn, assets);
```

Alternatively, handle over-withdrawals explicitly by pushing withdrawn - assets to the parent vault and include any residual token balance in _realAssets() until it has been swept.

RESOLUTION

Status: Resolved.

LOW **L-11** RESOLVED

Stale signed oracle updates appear fresh

CATEGORY	Validation
LOCATION	`FixedRatePricingOracle.sol`, `RiskRateOracle.sol`, `BaseTimelockedOracle.sol`
REVIEW	Main Review 2

DESCRIPTION

The signed tenor-oracle update paths do not bind signatures to the time when the off-chain data was observed. `FixedRatePricingOracle.postForecasts()` signs `nonce`, `deadline`, `tenorsHash`, `baseRatesHash`, `ciLowersHash`, `ciUppersHash`, and `stdDevsHash`. `RiskRateOracle.postRiskPremiums()` signs `nonce`, `deadline`, `tenorsHash`, `upRatesHash` and `loRatesHash`. Neither payload includes an observation timestamp or an equivalent signed validity window. When either update is eventually accepted, the child oracle stores the data timestamp as the relay block timestamp: `FixedRatePricingOracle` writes each forecast with `updatedAt = block.timestamp` and `RiskRateOracle` writes each risk-premium bucket with `updatedAt = block.timestamp`. `BaseTimelockedOracle` then treats that relay timestamp as the data freshness timestamp. A relayer holding a valid signed update can withhold it until a later block, submit it before `deadline` and make stale off-chain data appear fresh for a full `maxAgeSec` window from relay time. This also interacts with the rate-change circuit breaker: `_validateRateVector()` only checks `maxRateChangeWad` when the previous stored bucket for that rate vector is still fresh. Once the previous bucket has aged past `maxAgeSec`, the same delayed signed payload can bypass a change that would have reverted if relayed immediately.

The affected downstream sinks differ by oracle:

- `FixedRatePricingOracle` feeds base-swap tenor rates. Markets using it as `baseSwapRateOracle` can consume stale signed forecasts for market creation, swap entry, collateral requirements, and mark-to-market calculations.
- `RiskRateOracle` feeds sided risk-premium wrappers. `UpgradableSidedPremiumOracle.getRate()` exposes either `getUpRateByTenor()` or `getLoRateByTenor()`, `Utils.getRiskPremiumOracleRate()` only checks `isValid = true` and a non-negative rate, and `SwapCore.buySwap()` adds the returned premium to BUY_FIXED pricing or records it for BUY_FLOATING fee accounting.

A delayed relayer cannot invent rates or deadlines, but if signer deadlines exceed the intended freshness window, markets can price new swaps against stale signed forecasts or risk premiums that appear fresh on-chain.

Example:

1. An oracle has a fresh 5% bucket value.

L-11 Stale signed oracle updates appear fresh CONTINUED

2. Authorized signer(s) sign a 20% update with a deadline two hours in the future.
3. If the update is posted immediately, it reverts because the 15% jump exceeds `maxRateChangeWad`.
4. A relayer waits until the previous 5% bucket is older than `maxAgeSec`, but still before the signed deadline.
5. The relayer posts the same signed update. The change limit is skipped because the previous value is stale and the new value is stored with `updatedAt = block.timestamp`.
6. Downstream reads return the delayed 20% value with `isValid = true`, even though the signed data was produced earlier.

RECOMMENDATION

Include a signed observation timestamp, or a signed validity window such as `validAfter` / `validUntil`, in both `FORECAST_UPDATE_TYPEHASH` and `RISK_PREMIUM_UPDATE_TYPEHASH`. Reject updates whose observation timestamp is older than the configured freshness window at submission time. Base freshness checks should use the signed observation timestamp or a separate stored observation timestamp, rather than the relay timestamp. If the relay timestamp is still useful for operations, store it separately from the data timestamp. Also consider enforcing `deadline <= observedAt + maxAgeSec` so signer deadlines cannot exceed the oracle freshness policy.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/786`

LOW **L-12** RESOLVED

Inherited renounce bypasses wrapper timelock

CATEGORY	Unexpected Behavior
LOCATION	`BaseUpgradableOracleWrapper.sol`
REVIEW	Main Review 2

DESCRIPTION

`BaseUpgradableOracleWrapper` still inherits OpenZeppelin `Ownable2Step`, which in turn inherits `Ownable.renounceOwnership()`. The wrapper does not override that inherited external function. As a result, the owner can immediately set `owner()` to the zero address without using `queueRenounceUpgradability()` and `activateRenounceUpgradability()`.
This bypasses the wrapper's current three-day upgradability renouncement process. It also leaves `upgradabilityRenounced` as `false`, emits no `UpgradabilityRenounced` event and does not clear an in-flight downstream change. Since `queueDownstream()`, `activateDownstream()`, `cancelPendingDownstream()` and the wrapper renouncement functions are all `onlyOwner`, the pending downstream cannot be activated or cancelled after ownership is renounced.

RECOMMENDATION

Override `renounceOwnership()` in `BaseUpgradableOracleWrapper` and make it revert with a wrapper-specific error or route ownership renouncement through the existing timelocked `queueRenounceUpgradability()` and `activateRenounceUpgradability()` process. If the project wants ownership to remain transferable after upgradability is renounced, document that separately from the wrapper's own upgradability state.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/787`

LOW L-13

ACKNOWLEDGED

Long min poke interval forces stale rates

CATEGORY	Validation
LOCATION	AaveIndexBaseRateV1.sol
REVIEW	Main Review 2

DESCRIPTION

The constructor only rejects `_minPokeIntervalSec` values below `_twarWindowSec`. This matches the bracketing requirement immediately after a successful `poke()`, but it is not enough for continuous liveness. If `_minPokeIntervalSec` is greater than `_twarWindowSec`, the target time `block.timestamp - twarWindowSec` moves past `curr.timestamp` after one window. At that point `getRateByTenor()` and `getRates()` return invalid. The oracle still cannot be refreshed because `poke()` reverts with `TooSoon()` until the longer minimum interval elapses. Consequently, a deployment can have a forced invalid interval on every keeper cycle even when a keeper is online and willing to refresh the oracle. `SwapCore` treats an invalid base-rate oracle as a hard failure through `Utils.getBaseRateOracle()` and `Utils.getBaseRateOracleBatch()`, so affected markets can reject buys, early exits and LP mark-to-market operations during those intervals.

The provided Base deployment script sets both values to 12 hours, so this issue is not active for that deployment. The constructor and documentation still permit the unsafe configuration.

RECOMMENDATION

For this two-checkpoint design, require `_minPokeIntervalSec == _twarWindowSec` or remove the configurable minimum and use `twarWindowSec` as the rotation gate. If a separate anti-spam delay is needed, add a distinct maximum liveness requirement and test that no accepted configuration creates a period where reads are invalid while `poke()` is still too soon.

RESOLUTION

Status: Acknowledged.

LOW **L-14** RESOLVED

Tenor wrapper accepts short rate batches

CATEGORY	Validation
LOCATION	`UpgradableTenorRateOracle.sol`
REVIEW	Main Review 2

DESCRIPTION

`UpgradableTenorRateOracle._validateDownstreamHealth()` checks that `getRateByTenor()` succeeds and that `getRates()` returns `isValid = true` for a one-element probe. It does not check that the returned batch has the same length as the requested tenor array. The wrapper then forwards `getRates()` unchanged. `Utils.getBaseRateOracleBatch()` also returns the downstream array after only checking `isValid`. Downstream share-price code assumes the result length matches the active bucket count and `Utils.calculateTotalUnrealizedPnL()` indexes `rates[i]` for every active bucket after the batch call. If the downstream returns a short array with `isValid = true`, the market can pass wrapper validation but later revert during LP share-price calculation.

RECOMMENDATION

Validate batch length in the wrapper and in the shared utility. `UpgradableTenorRateOracle._validateDownstreamHealth()` should require `rates.length == probe.length` for its health probe and `getRates()` should reject any downstream response whose length differs from the requested tenor array. `Utils.getBaseRateOracleBatch()` should also enforce `rates.length == tenors.length` before callers index into the array. That gives the core accounting code a defensive check even if a non-wrapper oracle is configured directly.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/788

LOW **L-15** PENDING

Late recovery pokes smear Aave index growth

CATEGORY	Logical Error
LOCATION	`AaveIndexBaseRateV1.sol`
REVIEW	Main Review 2

DESCRIPTION

`_realizedTwar()` treats any target timestamp bracketed by `prev` and `curr` as valid, then derives `historicalIdx` by linear interpolation across the full checkpoint span. After a keeper outage, `poke()` promotes the old `curr` checkpoint to `prev` and stores the live index in `curr`. The new span can be many times larger than `twarWindowSec`, but reads immediately return `isValid = true`.

That makes the first recovery window depend on an assumed linear index path across the outage rather than the actual trailing window. If the Aave index moved mostly in the last TWAR window, the oracle smears that move over the full outage and materially underquotes the base rate. If the move happened earlier, it can overquote. `SwapCore` consumes the result for buys, early exits and LP mark-to-market, so users can trade against a distorted base rate immediately after recovery.

A missed keeper period can be followed by a valid quote before the oracle has two fresh checkpoints around the trailing window.

RECOMMENDATION

Do not return a valid rate when `curr.timestamp - prev.timestamp` is materially larger than `twarWindowSec`. For this two-checkpoint design, return `(0, false)` after a late recovery poke until a second on-time poke creates a fresh checkpoint pair. If immediate recovery is required, store a larger checkpoint ring buffer and interpolate from the nearest snapshots around `block.timestamp - twarWindowSec`.

RESOLUTION

Status: Pending.

LOW **L-16** RESOLVED

Long update interval forces invalid windows

CATEGORY	Validation
LOCATION	`BaseTimelockedOracle.sol`
REVIEW	Main Review 2

DESCRIPTION

`queueMinUpdateIntervalChange()` accepts any nonzero `_minUpdateIntervalSec`, and `activateMinUpdateIntervalChange()` stores it without checking it against `maxAgeSec`. The constructor has the same nonzero-only check for the initial value. That allows `minUpdateIntervalSec` to be configured above `maxAgeSec`. Once that happens, every successful post creates a forced invalid window. `_getRatesInternal()` returns invalid after the oldest active bucket is older than `maxAgeSec`, but `_validateTenorsAndCadence()` still rejects a new signed post with `UpdateTooFrequent` until `minUpdateIntervalSec` has elapsed from `lastPostTimestamp`.
Consequently, `getRateByTenor()` and `getRates()` can return invalid while an otherwise valid signer update cannot restore liveness. Consumers that require a valid tenor oracle then revert even though signers and relayers are available.

RECOMMENDATION

Reject any `minUpdateIntervalSec` greater than `maxAgeSec` in the constructor and in `queueMinUpdateIntervalChange()`. Also reject `queueMaxAgeChange()` values below the active `minUpdateIntervalSec` or require both values to be updated through a single coordinated timelocked change.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/789`

LOW L-17

ACKNOWLEDGED

Wrappers reject non-ERC165 oracle adapters

CATEGORY	Compatibility
LOCATION	`BaseUpgradableOracleWrapper.sol`
REVIEW	Main Review 2

DESCRIPTION

`BaseUpgradableOracleWrapper` does not accept a downstream oracle just because the oracle has the right read functions. Before construction, queueing or activation succeeds, the wrapper calls `supportsInterface()` on the downstream and requires it to advertise both `IOracleMetadata` and the wrapper's required read interface. That rule is stricter than several repository adapters currently satisfy. For example, `ChainlinkAdapter`, `PythAdapter` and `TenorRiskPremiumWrapper` expose `IRateOracle`-style functions, but they do not implement ERC-165. If one of those adapters is used as the downstream for `UpgradableRateOracle`, the wrapper's interface check fails. A missing `supportsInterface()` function causes `DownstreamUnhealthy`; a false return would cause `InterfaceNotSupported`.

RECOMMENDATION

Choose one wrapper compatibility rule and enforce it consistently. If repository adapters are allowed to sit behind `BaseUpgradableOracleWrapper`, make each adapter inherit `ERC165` and implement `supportsInterface()` for `IOracleMetadata` plus its read interface, such as `IRateOracle`. If some adapters are intentionally direct-use only, document that restriction and add deployment tests that fail when those adapters are wired through `UpgradableRateOracle` or another `BaseUpgradableOracleWrapper` child.

RESOLUTION

Status: Acknowledged.

LOW **L-18** RESOLVED

Pyth adapter reverts on invalid prices

CATEGORY	Unexpected Behavior
LOCATION	`PythAdapter.sol`
REVIEW	Main Review 2

DESCRIPTION

`PythAdapter.getRate()` returns `(rate, isValid)`, but several invalid oracle states revert instead of returning `(0, false)`. `getPriceNoOlderThan()` can revert for missing or stale prices and the adapter also reverts for non-positive prices, future timestamps, stale timestamps and scaling failures. Callers that are written against the `IRateOracle` validity flag cannot handle these cases cleanly. A stale or malformed Pyth price can therefore bubble up as a hard revert in pricing, settlement or maintenance logic that expected an invalid-rate response.

RECOMMENDATION

Wrap the Pyth read in `try/catch` and return `(0, false)` for stale, missing, malformed, future-dated or unscalable prices.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/790`

LOW L-19 RESOLVED

Short min spacing can erase TWAR history

CATEGORY	Validation
LOCATION	IndexBaseRateV1.sol
REVIEW	Main Review 2

DESCRIPTION

`IndexBaseRateV1` only requires `_minSpacingSec` to be nonzero. The ring buffer has eight slots, so a deployment with `minSpacingSec` below `twarWindowSec / (N - 1)` cannot retain enough history when `poke()` is called as often as the contract allows.

This matters because `poke()` is permissionless. For example, with a 12 hour TWAR window and 1 hour spacing, eight hourly pokes overwrite both constructor observations and every slot older than `now - twarWindowSec`. After that, `_findLeftAnchor()` cannot find a left anchor and `getRateByTenor()` returns `(0, false)` even though the cumulative source is healthy. `SwapCore` treats an invalid base-rate oracle as a hard failure through `Utils.getBaseRateOracle()` and `Utils.getBaseRateOracleBatch()`.

The supplied deployment task uses `minSpacingSec == twarWindowSec`, so this is not active for that configuration. The constructor still permits an unsafe configuration and the permissionless update function makes the liveness break externally triggerable once such a deployment exists.

RECOMMENDATION

Reject configurations that cannot retain at least one observation at or before the TWAR target under minimum-spacing pokes. A direct guard is:

```plain text
if (_minSpacingSec * (N - 1) < _twarWindowSec) revert InvalidSpacing();
```

Keep the existing deployment default of minSpacingSec == twarWindowSec.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/792

LOW L-20 RESOLVED

Tenor bucket activation blocks reseeding

CATEGORY	Validation
LOCATION	BaseTimelockedOracle.sol
REVIEW	Main Review 2

DESCRIPTION

`activateTenorBucketsChange()` replaces the active `tenorBuckets` array but leaves `lastPostTimestamp` unchanged. If the new tenor set contains a bucket that has never been posted before, that bucket's stored timestamp is zero. `getRateByTenor()` and `getRates()` then return invalid because `_getRatesInternalWith()` requires every active bucket to have a nonzero fresh timestamp.

This fail-closed behavior is reasonable on its own. The liveness problem is that the next signed post is still gated by the old `lastPostTimestamp`. If signers posted the old bucket set shortly before the owner activates the pending bucket change, `_validateTenorsAndCadence()` rejects the first post for the new bucket set with `UpdateTooFrequent` until `minUpdateIntervalSec` has elapsed.

Consequently, an owner can complete the timelocked bucket change and create a deterministic oracle outage even when fresh signatures for the new bucket set are available. Consumers that require a valid tenor oracle, such as market creation or swap pricing through wrappers, remain unavailable for the remaining cooldown. With the public deployment defaults observed in the scripts, this is bounded by the four-hour update interval, so the issue is a liveness/configuration hazard rather than a direct loss of funds.

RECOMMENDATION

Make tenor-bucket activation start a fresh posting window for the new bucket set. The smallest fix is to reset `lastPostTimestamp` when the bucket set is activated:

```plain text
tenorBuckets = pending.newTenorBuckets;
lastPostTimestamp = 0;
```

Alternatively, allow one immediate post after a bucket-set activation, require the pending bucket change to include initial signed rates for every new bucket, or make the activation function reject unless `minUpdateIntervalSec` has already elapsed since the last successful post.

L-20 Tenor bucket activation blocks reseeding CONTINUED

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/795

LOW L-21

ACKNOWLEDGED

MIN_SHARE_PRICE floor overpays withdrawing LPs

CATEGORY	Unexpected Behavior
LOCATION	`Utils.sol`
REVIEW	Main Review 2

DESCRIPTION

`Utils._computeSharePrice()` does not return the raw pool MtM share price when pool value falls below `MIN_SHARE_PRICE`. It returns `1e12` even if the real value per share is lower or zero. That value is not only a display fallback. `SwapCore.withdrawCollateral()` uses it to compute an LP's `fullValue` and the number of shares burned for an exact-amount withdrawal. Therefore, once a pool is deeply underwater, the first LPs to withdraw burn shares at the artificial floor instead of the true lower price. For example, if a pool has 1 USDC left and 2,000,000e6 shares outstanding, the raw share price is `5e11`, below the floor. `getPoolSharePrice()` reports `1e12`. An LP can withdraw the remaining collateral in two 0.5 USDC withdrawals while another LP still holds 1,000,000e6 shares. The remaining LP is left with shares against an empty pool. The same inflated value is also consumed by `getLpValue()`, `getPoolSharePriceVirtual()` and `KairosMarketAdapter._realAssets()`, so vault accounting can report floor value after the pool's real economic value has fallen below it.

RECOMMENDATION

Separate the raw economic redemption price from any display or safety floor. `withdrawCollateral()`, `getLpValue()`, virtual share price and adapter NAV should use the raw MtM share price or revert when the raw value is below the floor.

If active swaps remain and raw pool value is below `MIN_SHARE_PRICE`, block LP deposits and withdrawals or require proportional liquidation-style exits. The floor can remain as an explicitly labelled UI fallback, but it should not determine how many shares are burned or how many assets are reported.

RESOLUTION

Status: Acknowledged.

LOW L-22 RESOLVED

IndexBaseRateV1 ignores post-target anchors

CATEGORY	Unexpected Behavior
LOCATION	IndexBaseRateV1.sol
REVIEW	Main Review 2

DESCRIPTION

`IndexBaseRateV1._realizedTwar()` looks for the youngest stored observation at or before `block.timestamp - twarWindowSec`. It then interpolates from that left observation directly to the live source index.

This ignores any stored observation after the target timestamp. If the source index did not grow linearly between the left observation and the live read, post-target rate movement can influence the estimated historical index at the start of the TWAR window. The reported trailing rate can therefore be biased when observations are sparse or when rate movement is concentrated near the end of the interpolation span.

For example, assume the TWAR window is 12 hours and the read happens at 12:00, so the target is 00:00. The ring may contain observations at 18:00 yesterday, 06:00 today, and the live index at 12:00. A bracketing lookup would use the 18:00 and 06:00 observations to estimate the index at 00:00. The current code instead uses 18:00 and the live 12:00 index. If most index growth happened after 06:00, that later growth is smeared backward into the estimate for 00:00 and can understate the trailing 00:00 to 12:00 rate.

This is especially visible when ring advancement is delayed or when the oracle has a long gap between useful anchors. In that case `_realizedTwar()` can still return `isValid = true`, but the quote is based on a long chord ending at the live index rather than on observations that bracket the intended TWAR boundary.

RECOMMENDATION

Use a true bracketing lookup around `block.timestamp - twarWindowSec`. The oracle should find both the closest observation at or before the target and the closest observation at or after the target, then interpolate between those two observations.

If no right anchor exists, return invalid unless the live read is within a small configured maximum gap from the target. This keeps valid quotes tied to the intended trailing window instead of accepting a stale-to-live interpolation over an unbounded span.

L-22 IndexBaseRateV1 ignores post-target anchors CONTINUED

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/793

LOW L-23

ACKNOWLEDGED

Market Can Use Wrong Risk Tenor

CATEGORY	Validation
LOCATION	`SwapCore.sol`
REVIEW	Main Review 2

DESCRIPTION

SwapCore.createMarket() accepts a risk premium oracle as a generic IRateOracle. It validates only that getRate() returns valid, and buySwap() later reads the same no-argument getRate() value when pricing each swap.

That loses the market tenor at the risk-premium boundary. Tenor-specific premium wrappers can expose getRate() by reading a fixed configured tenor, such as defaultTenorSeconds = 30 days. If that wrapper is assigned to a 90-day or 180-day market, SwapCore has no way to detect the mismatch. Market creation passes, and every swap in that market is priced with the wrapper's default-tenor premium rather than the market's swapTerm premium.

Because market creation is permissionless, any creator can deploy or choose a valid risk-premium wrapper with the wrong default tenor, create a longer-dated market, and have it pass protocol validation. If LPs later supply that market, buyers can trade long-dated risk using a shorter-tenor premium.

Example: the risk oracle reports a 1% premium for 30 days and an 8% premium for 180 days. A 180-day market is created with a wrapper whose getRate() reads the 30-day bucket. Creation succeeds because getRate() is valid. Buyers then open 180-day swaps while paying the 1% short-tenor premium instead of the intended 8% long-tenor premium. The loss direction depends on the mismatch, but in the underpriced case LPs sell longer-dated exposure without the intended risk compensation.

RECOMMENDATION

Bind risk premium reads to the market term. Either make SwapCore consume a tenor-aware risk premium interface and call getRateByTenor(market.swapTerm), or require wrappers used as riskPremiumOracle to expose their configured tenor and reject market creation unless it equals swapTerm.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-01 ACKNOWLEDGED

Virtual share price scales with expired queue

CATEGORY	Warning
LOCATION	`Utils.sol`
REVIEW	Main Review 2

DESCRIPTION

`computeVirtualSharePrice()` walks the market expiry queue and virtually settles every expired unsettled swap until it reaches the first active swap. `KairosMarketAdapter._realAssets()` calls `SwapCore.getPoolSharePriceVirtual()` on chain whenever the adapter reports its current LP value. That makes adapter accounting depend on the number of expired unsettled swaps in the queue. A market with many expired swaps can make `realAssets()` expensive enough to threaten allocation, deallocation, and vault accounting calls that need the adapter's value. Since swap creation is permissionless, this creates a liveness dependency on keepers or other users settling expired swaps before the queue grows too far. We ran a local test which opened expired unsettled swaps in two markets and measured the adapter view. `KairosMarketAdapter.realAssets()` used 588,897 gas with 10 expired swaps and 1,144,513 gas with 50 expired swaps. The exact gas will vary with market state, but the important property is the unbounded queue-length dependency in an on-chain accounting function.

RECOMMENDATION

Consider adding stronger queue controls: cap open swaps per market, require settlement batches before new activity or make expired-swap cleanup part of the protocol's keeper incentives.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-02 ACKNOWLEDGED

Compound borrow index accrues with zero debt

CATEGORY	Unexpected Behavior
LOCATION	`CumulativeIndexBorrowCompound.sol`
REVIEW	Main Review 2

DESCRIPTION

`CumulativeIndexBorrowCompound.getCumulativeIndex()` reads Compound's `TotalsBasic` state and projects `baseBorrowIndex` forward whenever `lastAccrualTime` is behind the current timestamp.
It does not check `t.totalBorrowBase` first:

```plain text
uint256 borrowIndex = uint256(t.baseBorrowIndex);
if (borrowIndex == 0) return (0, false);

uint256 elapsed = block.timestamp - uint256(t.lastAccrualTime);
if (elapsed > 0) {
borrowIndex = _tryProject(borrowIndex, elapsed);
}

return (borrowIndex * SCALE_FACTOR, true);
```

That means the adapter can report borrow-index growth even when the Comet market has no outstanding borrow principal. If getBorrowRate(0) returns a positive rate, `_tryProject()` applies that rate and returns the projected index as valid. Kairos then treats the value as a cumulative floating index. A listed Compound borrow-index market can therefore mark or settle swaps against synthetic borrow accrual from a period where no Compound borrowers were paying interest.

RECOMMENDATION

Check `t.totalBorrowBase` before projecting the borrow index.
If `totalBorrowBase == 0`, return the stored `baseBorrowIndex` without projection, or return `(0, false)` so settlement-critical callers reject the read. The safer choice depends on whether the protocol wants zero-debt periods to mean flat floating accrual or an invalid oracle state.
If deployments rely on Compound borrow-index markets never reaching zero debt, enforce that as a listing invariant. Do not leave it as an off-chain assumption.

I-02 Compound borrow index accrues with zero debt CONTINUED

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-03 ACKNOWLEDGED

Early exit bypasses liquidation rewards

CATEGORY	Warning
LOCATION	SwapCore.sol
REVIEW	Main Review 2

DESCRIPTION

`SwapCore.exitSwapEarly()` does not check whether the swap is already liquidatable before it marks the position as `isEarlyExit` and settles it through `_settleSwaps()`. That lets the position owner close a liquidatable swap through ordinary early-exit accounting instead of liquidation accounting. This matters because `Utils.settleSwap()` always returns the remaining buyer collateral plus the prefunded `liquidationBounty` to the buyer on a non-liquidation close. By contrast, `Utils.liquidateSwap()` pays the buyer-side bounty to the liquidator when the buyer is liquidatable. The same issue exists when the pool side is liquidatable. Liquidation pays `poolCollateralBacking * liquidationIncentive / WAD` to the liquidator and gives the buyer the remaining pool backing. Early exit can instead pay the buyer the full pool backing, so the buyer captures the amount that liquidation reserves for the liquidator. Consequently, the swap owner can front-run liquidation with `exitSwapEarly()` and take the liquidation reward value. This weakens the keeper incentive that is supposed to make liquidation reliable. It also gives the owner a better close outcome than the liquidation rules allow once the position has already crossed the liquidation threshold.

RECOMMENDATION

Make early exit reject liquidatable swaps before setting `swap.isEarlyExit`. `exitSwapEarly()` should call the same liquidation check used by `liquidateSwap()` with a fresh reference index. If either side is liquidatable, revert and require liquidation, or route the close through liquidation accounting so the liquidation reward is paid consistently.

The simplest rule is to reject voluntary early exit once liquidation is available:

```plain text
(bool isBuyerLiquidatable, bool isPoolLiquidatable,,) = Utils.isSwapLiquidatable(...);
if (isBuyerLiquidatable || isPoolLiquidatable) revert SwapLiquidatable();
```

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-04 PENDING

Kairos oracles freeze negative-rate markets

CATEGORY	Logical Error
LOCATION	`KairosOracleBeta.sol, KairosOracle.sol`
REVIEW	Main Review 2

DESCRIPTION

`KairosOracleBeta.setReferenceRate()` rejects every non-positive rate. Its `getRate()` function also reports the oracle as invalid when the stored rate is zero or negative. This is incompatible with markets whose valid reference rate can be negative, such as funding-rate markets.

The core protocol otherwise supports signed rates. `RateIndexLib.update()` can accumulate negative `SpotRate` values and the shared PoC shows that an otherwise identical SpotRate market settles normally when the oracle accepts a negative rate.

With `KairosOracleBeta`, a negative real-world rate cannot be posted. The market either keeps using the last positive value until it becomes stale, or a newer signed oracle variant can store a negative value that `getRate()` then marks invalid. Once the oracle is invalid, settlement-critical functions that call `RateIndexLib.update()` revert with `E606`. Existing swaps can remain unsettled until the rate turns positive again, and expired unsettled swaps block normal market operation.

`KairosOracle.getRate()` has the same non-positive invalidation check, so the signed N-of-M replacement preserves the same incompatibility even though its `setRate()` function accepts signed bounds.

RECOMMENDATION

Allow signed rates in Kairos rate oracles when the configured market supports negative reference rates. Remove the `rate <= 0` invalidation from `getRate()` and validate only initialization, freshness and configured absolute bounds. If a deployment intentionally supports only strictly positive rates, enforce that at market configuration time and do not use those oracles for funding-rate or other signed-rate markets.

PROOF OF CONCEPT https://gist.github.com/GuardianAudits/4b326434a804044ab80adaac7fa0eb40

RESOLUTION

Status: Pending.

INFORMATIONAL I-05 PENDING

TWAPOracle backfills current spot rate

CATEGORY	Unexpected Behavior
LOCATION	`TWARateOracle.sol`
REVIEW	Main Review 2

DESCRIPTION

`TWAPOracle.getRateWindow()` reads the current spot rate and uses it to calculate a fresh rate index for the entire elapsed interval since `lastUpdate`. The function does this even when the spot rate only became true at the end of that interval.

This means a transient or recent spot-rate change is backfilled over the full time since the last checkpoint. The returned average is therefore not a true time-weighted average unless updates happen exactly when the underlying spot rate changes.

If this oracle is used as a reference-rate source and it is read by markets or wrappers, a current spot spike can inflate or deflate the reported 24-hour rate for the whole gap since the last TWAP update. For example, assume the oracle was last checkpointed one hour ago while the underlying spot rate was 1%, then the spot source jumps to 20% immediately before a market read. `getRateWindow(1 hours)` will synthesize the whole missing hour at 20% and return roughly 20%, even though that rate was only true at the end of the interval.

For the default `getRate()` path, which reads `getRateWindow(24 hours)`, a mature oracle history dilutes the same one-hour error into the 24-hour average, but it still treats the missed hour as if the new spot rate applied for the entire hour. With a 1% to 20% move, that can inflate the reported 24-hour rate by roughly `(20% - 1%) / 24`, about 0.79 percentage points, until the next checkpoint corrects future accrual.

RECOMMENDATION

Do not calculate historical accrual from a single current spot read. The oracle should update checkpoints at a bounded cadence and only report averages over stored historical index values.

If read-time freshness is required, cap the elapsed interval that `getRateWindow()` may fill with the current spot rate. When the elapsed interval exceeds that cap, return invalid instead of synthesizing a historical index from the latest spot value.

RESOLUTION

Status: Pending.

INFORMATIONAL I-06 RESOLVED

Exact expiry can be early exited

CATEGORY	Warning
LOCATION	SwapCore.sol
REVIEW	Main Review 2

DESCRIPTION

`exitSwapEarly()` blocks early exit only when `swapExpiry < block.timestamp`. At the exact maturity timestamp, `swapExpiry == block.timestamp`, so the function still marks the swap as `isEarlyExit` and settles it through the early-exit calculation.
That is a boundary mismatch. Once a swap has reached maturity, it should settle as an ordinary expiry settlement. Treating exact maturity as an early exit can change event data, reward eligibility, accounting labels, and fee treatment for integrations that distinguish normal settlement from early exit.

RECOMMENDATION

Change the expiry check to reject early exit at exact maturity:
``plain text
if (swapExpiry <= block.timestamp) revert Errors.E502();
`
At exact expiry, users and adapters should call makePayment()` and receive normal settlement behavior.

RESOLUTION

Status: Resolved.

INFORMATIONAL I-07 ACKNOWLEDGED

Spot bucket aggregation biases MTM

CATEGORY	Warning
LOCATION	`Utils.sol`
REVIEW	Main Review 2

DESCRIPTION

`calculateBucketUnrealizedPnL()` values a whole bucket from averaged bucket state. It derives one `avgEntryTime`, one harmonic `avgEntryIndex` and one `avgFixedRate`, then uses those aggregate values as if every swap in the bucket had the same entry conditions.

That approximation is exact for the cumulative convention in the tested model, but it is not exact for the spot-rate conventions. For `SpotRate` and `SpotCompoundRate`, the aggregate entry index can produce a small difference from summing the same swaps one by one. Therefore LP share price can include a bounded mark-to-market bias when a bucket contains swaps that entered at different times and the market uses a spot convention.

A local quantification test showed zero bias for `Cumulative`, about \$2.44 of bias on \$10 million notional over 90 days at a 10% sustained rate, about \$40 on \$10 million over one year at 10%, and about \$1,154 in a pathological \$100 million, one-year, 20% scenario. This is still a real accounting approximation, but the observed magnitude is low under the current bucket constraints.

RECOMMENDATION

This is primarily informational. Document that bucket-level MTM is an approximation for spot-rate conventions, including the expected bounded drift under the current bucket-interval constraints.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-08

PENDING

Referral IDs have 48-bit collision space

CATEGORY	Warning
LOCATION	`ReferralIdGenerator.sol`
REVIEW	Main Review 2

DESCRIPTION

`ReferralIdGenerator.generateReferralId()` comments that it takes 64 bits from the hash. It then converts that value to exactly eight base62 characters.

Eight base62 characters do not preserve 64 bits of output space. They encode `62^8` possible values, which is about 47.6 bits. The fixed-length conversion keeps only the low eight base62 digits, so different 64-bit inputs can map to the same referral ID.

The comment overstates the effective collision resistance of the generated ID.

RECOMMENDATION

Either increase the referral ID length or update the documentation to describe the actual output space. If referral collisions matter operationally, store assigned IDs and reject collisions at registration time instead of relying only on deterministic truncation.

RESOLUTION

Status: Pending.

INFORMATIONAL I-09 RESOLVED

Self-downstream permanently bricks wrapper

CATEGORY	Warning
LOCATION	`BaseUpgradableOracleWrapper.sol`
REVIEW	Main Review 2

DESCRIPTION

`queueDownstream` rejects the zero address and the current downstream, but it does not reject `address(this)`. A wrapper also advertises `IOracleMetadata` and `IRateOracle`, so `address(this)` passes `_requireDownstreamInterface`. While the old downstream is still active, `IOracleMetadata(address(this)).decimals()` forwards to the old downstream and passes the decimals check. The same candidate also passes `activateDownstream`. During activation, `_validateDownstreamHealth(address(this))` calls `getRate()` before `downstream` is reassigned. That read still resolves to the old downstream, so activation succeeds. After `downstream` is set to the wrapper itself, `getRate()`, `decimals()` and `description()` recurse until they run out of gas. Consequently, every consumer using the wrapper loses oracle liveness. The owner cannot repair the wrapper through another queued downstream because `queueDownstream` first reads the current downstream decimals, which now recurses through the wrapper itself. `engageEmergencyCutoff` also cannot make `getRate()` return `(rate, false)` because `getRate()` calls the downstream before `_gateValid`.

RECOMMENDATION

Reject direct self-reference in every downstream assignment.

```plain text
if (newDownstream == address(this)) revert SameDownstream();
```

If wrapper-to-wrapper composition is expected, also add a cycle check that rejects a candidate whose downstream chain already reaches the current wrapper. Otherwise, document that wrappers must not be used as downstreams for other wrappers and enforce that rule in deployment tooling.

RESOLUTION

Status: Resolved.

INFORMATIONAL I-10 RESOLVED

Cutoff cannot mask downstream reverts

CATEGORY	Warning
LOCATION	`UpgradableTenorRateOracle.sol`
REVIEW	Main Review 2

DESCRIPTION

`engageEmergencyCutoff()` is documented as forcing wrapper reads to return `isValid = false`. In the tenor wrapper, that only happens after the downstream call has already returned. `getRateByTenor()`, `getRates()` and `getRate()` all call `ITenorRateOracle(downstream)` first and apply `_gateValid()` afterward. If the active downstream starts reverting after activation, enabling `emergencyCutoff` does not turn those reads into stable invalid responses. The wrapper still bubbles up the downstream failure. Operators may expect the cutoff event to put the wrapper into a controlled invalid state, while consumers continue to see raw downstream reverts instead of `(rate, false)`.

RECOMMENDATION

If cutoff is meant to provide a stable invalid response, check `emergencyCutoff` before calling the downstream. `getRateByTenor()` and `getRate()` can return `(0, false)` and `getRates()` can return a zero-filled array with the same length as the requested tenor list plus `isValid = false`. If raw downstream reverts are acceptable during cutoff, update the wrapper comments and monitoring expectations so operators do not rely on cutoff to mask a failing downstream.

RESOLUTION

Status: Resolved.

INFORMATIONAL I-11 RESOLVED

Pyth confidence config can overflow

CATEGORY	Warning
LOCATION	PythAdapter.sol
REVIEW	Main Review 2

DESCRIPTION

`PythAdapter` accepts any `_maxConfRatioBps` value and stores it as an immutable configuration. `getRate()` then evaluates `maxConfRatioBps * uint64(pythPrice.price)` inside the confidence-ratio check. With an excessively large configured ratio, that multiplication can overflow and revert in Solidity 0.8. Even below the overflow boundary, values above a sane basis-point range effectively disable the confidence filter while still appearing configured. This is a deployment-configuration hazard. A bad value can turn a normal oracle read into a hard revert or make the confidence guard meaningless.

RECOMMENDATION

Validate `_maxConfRatioBps` in the constructor. For example, require it to be zero or less than or equal to `10_000`, unless the protocol intentionally supports wider ratios. Also consider rewriting the comparison to avoid multiplication overflow if wider limits are required.

RESOLUTION

Status: Resolved.

INFORMATIONAL I-12 ACKNOWLEDGED

Risk premiums can cross between sides

CATEGORY	Warning
LOCATION	`RiskRateOracle.sol`
REVIEW	Main Review 2

DESCRIPTION

`RiskRateOracle.postRiskPremiums()` validates the upper and lower premium vectors independently. It checks array lengths, signer approval, cadence, bounds and per-side rate-change limits, but it never checks any relationship between `upRates[i]` and `loRates[i]`. The contract documentation says both vectors come from the same Monte Carlo TWAR distribution and must be co-posted to preserve symmetric-quantile coherence. A signer or pricing-pipeline mistake can therefore post internally inconsistent pairs that pass all on-chain checks. For example, an upper premium below the lower premium can be stored if each side is independently within bounds and rate-change limits.

RECOMMENDATION

Encode the required relationship between each upper and lower premium pair. If the intended invariant is simple ordering, reject crossed pairs directly.

```plain text
if (upRates[i] < loRates[i]) revert InvalidPremiumPair();
```

If the pricing model requires a different shape constraint, enforce the weakest critical invariant on-chain and keep the richer model checks in the off-chain pipeline.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-13 ACKNOWLEDGED

Missing source-market caps for rate oracles

CATEGORY	Warning
LOCATION	SwapCore.sol
REVIEW	Main Review 2

DESCRIPTION

`_initializeMarket()` validates market parameters, checks oracle health and initializes the selected reference-rate index. It does not record which external lending market backs the chosen `referenceRateOracle` and it does not enforce any minimum external supply, debt, shares, or source-market depth. That leaves the relationship between Kairos notional and source-market manipulation cost entirely off-chain. `Utils.validateMarketParams()` checks generic market fields and `buySwap()` later limits `notionalAmount` against Kairos LP collateral. Neither check compares the open Kairos exposure with the amount of capital needed to move the referenced Aave, Compound or Morpho market. The deployed Base source oracles were sampled on 2026-05-09. External source-market state was read at Base block `45762123` and live `SwapCore` market usage was checked at block `45762207`. The checked-in production market config had no filled `PROD_MARKETS` entries. The sampled live Kairos markets that matched Aave and Compound source oracles had zero notional and no sampled live market matched the saved Morpho oracle addresses. Therefore this is not evidence of an active exploit.

The snapshot still shows why source-market depth should be a listing invariant:

In the first table, `Utilization at sample` is the source market's current borrow assets divided by supply assets. `Capital to reach 95% utilization` is the extra USDC borrow needed to push the source market from its sampled utilization to 95%. This is capital that must be sourced and kept safe, not the amount permanently spent. The carry columns estimate the interest cost of holding that extra borrow for one hour or one day at the target utilization's borrow rate.

<table header-row="true">
<tr>
<td>Source market</td>
<td>Utilization at sample</td>
<td>Capital to reach 95% utilization</td>
<td>Estimated one-hour carry at 95%</td>
<td>Estimated one-day carry at 95%</td>
</tr>
<tr>

I-13 Missing source-market caps for rate oracles CONTINUED

<td>Aave V3 Base USDC</td>
<td> `86.25%` </td>
<td> `15,529,178.18` USDC</td>
<td> `168.4100` USDC</td>
<td> `4,041.8409` USDC</td>
</tr>
<tr>
<td>Compound III Base USDC</td>
<td> `89.92%` </td>
<td> `490,641.34` USDC</td>
<td> `12.3221` USDC</td>
<td> `295.7293` USDC</td>
</tr>
<tr>
<td>Morpho Blue Base USDC/cbBTC</td>
<td> `89.56%` </td>
<td> `70,946,675.99` USDC</td>
<td> `901.4294` USDC</td>
<td> `21,634.3060` USDC</td>
</tr>
<tr>
<td>Morpho Blue Base USDC/WETH</td>
<td> `89.42%` </td>
<td> `4,250,991.74` USDC</td>
<td> `53.9979` USDC</td>
<td> `1,295.9485` USDC</td>
</tr>
</table>

The same snapshot also showed the current source-market balances:

In the second table, `Supply assets` and `Borrow assets` are the sampled balances of the underlying lending market, denominated in USDC. These balances provide context for the manipulation-cost estimates above. A downward-rate manipulation could also be modeled by adding supply or repaying debt, but that requires a chosen target utilization and is less directly comparable across protocols. The table therefore keeps only the observed balances.

<table header-row="true">
<tr>
<td>Source market</td>
<td>Supply assets</td>
<td>Borrow assets</td>

I-13 Missing source-market caps for rate oracles CONTINUED

</tr>
<tr>
<td>Aave V3 Base USDC</td>
<td> `177,492,726.40` USDC</td>
<td> `153,088,911.91` USDC</td>
</tr>
<tr>
<td>Compound III Base USDC</td>
<td> `9,667,287.75` USDC</td>
<td> `8,693,282.02` USDC</td>
</tr>
<tr>
<td>Morpho Blue Base USDC/cbBTC</td>
<td> `1,304,587,762.43` USDC</td>
<td> `1,168,411,698.31` USDC</td>
</tr>
<tr>
<td>Morpho Blue Base USDC/WETH</td>
<td> `76,220,664.15` USDC</td>
<td> `68,158,639.20` USDC</td>
</tr>
</table>

The carry estimate is first-order. It uses `extraBorrow * targetAPR * holdSeconds / 365 days`, with Aave target rates derived from the live Aave USDC interest-rate strategy, Compound target rates read from `Comet.getBorrowRate(targetUtilization)` and Morpho target rates read by simulating the market state passed into the configured IRM. It does not include collateral requirements, borrow caps, isolation constraints, liquidation risk, price risk, gas or the opportunity cost of capital. Those figures are not exploit proofs. They are risk thresholds. A Kairos market should not allow open notional large enough that the value gained from a manipulated rate can exceed the cost of creating that rate move in the source lending market.

This becomes an exploitable vector when a trader can hold the profitable side of a Kairos swap, move the referenced source market during the rate observation or settlement window, and earn more from the Kairos payout than the cost and risk of the source-market action. In simple terms, the dangerous condition is:

```plain text
Kairos notional * rate move * affected time / 365 days
> source-market carry + gas + slippage + liquidation/price-risk cost
```

I-13 Missing source-market caps for rate oracles CONTINUED

The risk is highest when the source market is small, already close to the steep part of its interest-rate curve or can be moved with a modest borrow or supply operation. It is also higher when the Kairos oracle reads a live rate, backfills a current rate over a stale interval, uses a short TWAR or can be updated at predictable settlement times. In those cases, the attacker does not need to manipulate the source market permanently. They only need the manipulated rate to be accepted for the window that affects entry pricing, mark-to-market or settlement.

The Morpho sources have an additional edge case. If a listed Morpho market is new, thin, or can be emptied by one actor, share-ratio based indexes can move for accounting-state reasons rather than accumulated yield. That can make the source cheaper to manipulate than the large-market utilization numbers above suggest.

This is not exploitable in the same way when the Kairos market has no open notional, buying is paused, the source-market move cannot be sustained through the oracle window or the Kairos notional cap is small enough that the maximum rate distortion is below the attacker's expected cost. Deep source markets and long, checkpointed TWARs reduce the risk, but they do not replace explicit notional caps because the safe cap changes as source liquidity changes.

RECOMMENDATION

Add source-market risk limits to market listing and ongoing monitoring. For each allowed oracle source, store the underlying protocol, asset or market id, minimum source supply, minimum source debt, minimum Morpho shares where relevant and maximum Kairos open notional. Set the notional cap from a conservative manipulation-cost model for the configured oracle window. Pause new swaps or mark the source invalid when source liquidity falls below the threshold.

For Morpho share-ratio sources, also block low-share or empty markets unless a stateful monotonic oracle is used. TWARs and checkpointed sources help, but they should not replace explicit source-market depth checks.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-14

PARTIALLY RESOLVED

Flat curve extrapolates tenors

CATEGORY	Warning
LOCATION	BaseTimelockedOracle.sol
REVIEW	Main Review 2

DESCRIPTION

`BaseTimelockedOracle._interpolateFromMemory()` returns the first bucket rate for tenors below the configured curve and the last bucket rate for tenors above it.

```plain text
if (!foundLower) return bucketRates[0];
if (!foundUpper) return bucketRates[len - 1];
```

This means an out-of-range tenor is still reported as a valid oracle value. SwapCore._initializeMarket() only checks that getRateByTenor(params.swapTerm) returns ok == true, so a market whose swapTerm is outside the oracle's configured tenor buckets can pass creation if the oracle serves the endpoint bucket.

This is not currently an active long-dated deployment issue in the Base artifacts checked. The deployed fixed-rate curves use 7d, 14d, 30d, 60d, 90d, 180d and 365d buckets. The shorter deployed RiskRateOracle_aave_v3_usdc_borrow curve stops at 90d, but its deployed sided wrappers are both configured for 30d. The current Base SwapCore market list contains only 1h, 12h, 1d and 7d markets, and we did not find an active market whose swapTerm` exceeds the referenced bucketed curve.

The residual risk is operational. A future market or wrapper can be configured beyond the curve domain and receive the endpoint rate instead of being rejected. If term risk is expected to increase outside the configured curve, flat endpoint reuse can underprice longer-dated exposure.

RECOMMENDATION

Make the oracle domain explicit. The safest behavior is to return invalid for tenors below the first bucket or above the last bucket, then require market creation to stay within that domain.

If flat extrapolation is intentional, encode that assumption in deployment validation and documentation. Market creation scripts should check `swapTerm <= max(tenorBuckets)` for bucketed base-rate and risk-premium oracles and wrapper deployment should reject default tenors outside the downstream curve unless an operator deliberately opts into endpoint reuse.

I-14 Flat curve extrapolates tenors CONTINUED

RESOLUTION

Status: Partially resolved.

https://github.com/kairosswap/protocol/pull/806

INFORMATIONAL I-15 RESOLVED

Nonexistent markets report default share price

CATEGORY	Warning
LOCATION	SwapCore.sol
REVIEW	Main Review 2

DESCRIPTION

`getPoolSharePrice()` returns `WAD` whenever `market.pool.totalShares == 0` before it checks whether the market exists. For an arbitrary `marketId` that was never created, the default `Market` struct has `exists == false` and `pool.totalShares == 0`. Therefore the function reports a valid-looking `1e18` share price for a nonexistent market. This can be confusing for frontends, indexers, monitoring and other off-chain consumers that treat a successful `getPoolSharePrice()` response as evidence that the market exists and has a normal initial share price.

RECOMMENDATION

Reject nonexistent markets before applying the empty-pool default.

```plain text
function getPoolSharePrice(bytes32 marketId) public view returns (uint256) {
Types.Market storage market = markets[marketId];
if (!market.exists) revert Errors.E300();
if (market.pool.totalShares == 0) return WAD;
...
}
```

RESOLUTION

Status: Resolved.

MEDIUM **M-01** RESOLVED

Morpho sources can brick base-rate TWAR pokes

CATEGORY	DoS
LOCATION	`IndexBaseRateV1.sol, AaveIndexBaseRateV1.sol`
REVIEW	Remediation Review 2

DESCRIPTION

`IndexBaseRateV1.poke()` and `AaveIndexBaseRateV1.poke()` require every new cumulative source index to be greater than or equal to the last stored observation. If the source returns any lower value, the poke reverts with `IndexNonMonotonic()`.

That rule is too strict for Morpho share-ratio cumulative adapters. `CumulativeIndexBorrowMorpho` and `CumulativeIndexSupplyMorpho` document that normal Morpho rounding can make the reported share-price index decrease by sub-wei amounts even when the underlying borrow or supply rate is non-negative. The core cumulative-index consumer now clamps decreases before settlement uses them, but the base-rate TWAR oracles do not clamp. They stop accepting pokes.

Consequently, a Morpho cumulative source can be usable enough for `RateConvention.Cumulative` settlement while still being unsafe as the source for a base-rate TWAR oracle. Once a rounding-driven decrease blocks `poke()`, the retained observations age out and base-rate reads can become invalid. Markets that depend on that base-rate oracle can then reject swap entry, early exits and LP mark-to-market operations until the source index moves back above the last stored observation or the oracle is replaced.

RECOMMENDATION

Do not plug raw Morpho share-ratio cumulative adapters into base-rate TWAR oracles that require strict monotonicity.

For generic cumulative sources, allow a bounded monotonic clamp in `poke()`:

```plain text
if (idx < lastIdx) {
if (lastIdx - idx > maxAllowedRoundingDrift) revert IndexNonMonotonic();
idx = lastIdx;
}
```

M-01 Morpho sources can brick base-rate TWAR pokes CONTINUED

RESOLUTION

Status: Resolved.

MEDIUM M-02

PARTIALLY RESOLVED

Oracle updates can sandwich LP shares

CATEGORY	MEV
LOCATION	`SwapCore.sol`
REVIEW	Remediation Review 2

DESCRIPTION

M-02 Oracle updates can sandwich LP shares CONTINUED

`supplyCollateral()` updates the reference index, reads the current pool share price and immediately mints LP shares at that price. `withdrawCollateral()` uses the same current mark-to-market share price for redemption, while its only deposit-age guard blocks withdrawal in the same block as the deposit.

The share price is therefore sensitive to discrete oracle updates. If a pending base-rate or reference-index update will increase LP mark-to-market value, an informed actor can deposit before the update and withdraw after the one-block lock at the higher share price. If the pending update will decrease share value, the actor can withdraw before the update and leave the loss with remaining LPs.

For example, consider a `BUY_FIXED` market whose `baseSwapRateOracle` is `FixedRatePricingOracle`, with open swaps and enough idle collateral for withdrawals:

1. A relayer broadcasts `FixedRatePricingOracle.postForecasts(tenors, baseRates, ciLowers, ciUppers, stdDevs, observedAt, deadline, signatures)` with lower `baseRates`. The calldata reveals that the base-rate curve will move down, but the oracle still reports the old 5% curve until the relayer transaction executes. A watcher sends `supplyCollateral()` with higher priority so it lands earlier in the same block. The pool share price is still based on the higher projected floating leg, so the watcher receives shares at the pre-update price.
2. Later in that same block, the relayer's `postForecasts()` transaction executes and stores the lower curve. The same active swaps are now marked with a lower projected floating leg, so `getPoolSharePrice()` increases and the newly minted shares share in that NAV uplift.
3. The watcher cannot withdraw in the same block because `lastDepositBlock` blocks same-block withdrawal. Instead, they wait one block and call `withdrawCollateral()`. The withdrawal burns the short-lived shares at the higher post-update share price. The watcher exits with more collateral than they deposited, while incumbent LPs lose part of the oracle-update gain that belonged to their pre-existing exposure.

This is not prevented by the signed-oracle freshness checks. A valid oracle update can still be relayed at a chosen time and LP share minting or burning has no oracle-epoch guard. Consequently, passive LPs can be diluted or left with losses when share accounting changes across a known oracle update.

RECOMMENDATION

Consider replacing the one-block LP withdrawal guard with a real cooldown for active markets. When `pool.lockedCollateral > 0`, `supplyCollateral()` should set an unlock timestamp or unlock block for the LP position and `withdrawCollateral()` should reject withdrawals until that delay has elapsed. The delay should cover at least the oracle update cadence, so a watcher cannot deposit before `postForecasts()` and redeem immediately after the update lands.

M-02 Oracle updates can sandwich LP shares CONTINUED

RESOLUTION

Status: Partially resolved.

LOW L-01

RESOLVED

Max live gap can revive stale base-rate quotes

CATEGORY	Unexpected Behavior
LOCATION	IndexBaseRateV1.sol
REVIEW	Remediation Review 2

DESCRIPTION

`IndexBaseRateV1` uses `maxLiveGapSec` to reject stale interpolation spans. `_realizedTwar()` returns invalid when the left anchor sits too far before the TWAR boundary, which prevents a single late poke after a long outage from making a stale-to-live interpolation look usable.

`setMaxLiveGapSec()` lets the owner change that safety bound immediately. The function has no timelock and no upper bound. An owner can therefore widen the accepted gap after an outage and make the same stale quote valid in the same transaction.

This is owner-only, so it is not a permissionless attack. It is still a protocol safety issue because `maxLiveGapSec` is an oracle-quality control, not a harmless display parameter. Markets and integrators may rely on the gap bound as the reason stale base-rate quotes fail closed. The current setter lets governance or an operator weaken that guarantee without the notice period used for other sensitive oracle controls.

RECOMMENDATION

Cap `newMaxLiveGapSec` to a conservative multiple of `twarWindowSec`, such as `2 * twarWindowSec` or to a deployment-specific maximum set at construction.

Timelock increases to `maxLiveGapSec` because they make the oracle more permissive. Decreases can remain immediate because they only make stale data harder to accept. Emit the old and new value as the contract already does and document the operational meaning of the configured gap for market integrators.

RESOLUTION

Status: Resolved.

LOW L-02 RESOLVED

Expired adapter swaps keep liquidation haircuts

CATEGORY	Unexpected Behavior
LOCATION	KairosBuyerAdapter.sol
REVIEW	Remediation Review 2

DESCRIPTION

`KairosBuyerAdapter._realAssets()` values every unsettled swap with the same liquidation assumptions. When the buyer is owed value, it caps pool-side recovery at `poolBacking - liquidatorReward` even if the swap has already expired. When the buyer owes more than its remaining collateral, it reports zero and drops the prefunded `liquidationBounty`. The code comment treats this post-expiry undercount as conservative.

That is not equivalent to normal expiry settlement. `Utils.isSwapLiquidatable()` returns false once `block.timestamp >= swap.entryTimestamp + market.swapTerm`, so no liquidator can take a reward after expiry. `Utils.settleSwap()` then settles against full available collateral and returns any remaining buyer collateral plus the prefunded liquidation bounty to the buyer.

Consequently, an expired unsettled adapter-owned swap can be worth more at settlement than `realAssets()` reports before settlement. A Vault V2 depositor can mint shares while adapter NAV is understated, then anyone can settle the swap and lift adapter NAV to the realized value. The gap is bounded by the post-expiry liquidation haircut or returned bounty on each affected swap, but it transfers value from existing vault shareholders to deposits made before settlement.

RECOMMENDATION

Do not duplicate settlement math inside `KairosBuyerAdapter`. Instead, add or reuse a `SwapCore` view that returns the settlement-equivalent buyer value for an expired unsettled swap, using the same expiry index, caps, direction, collateral release and bounty rules that `makePayment()` would use.

`KairosBuyerAdapter._realAssets()` should call that core preview for expired unsettled swaps. Keep the liquidation-adjusted conservative valuation only while a swap is still live and can still be liquidated.

If you want a simpler and more conservative fix, block vault deposits, withdrawals and other adapter valuation-sensitive actions whenever the buyer adapter has expired unsettled swaps. Require those swaps to be settled first, then let `_realAssets()` value only live swaps and already-realized token or escrow balances.

L-02 Expired adapter swaps keep liquidation haircuts CONTINUED

RESOLUTION

Status: Resolved.

LOW L-03

RESOLVED

Buy swap lacks total input bound

CATEGORY	Unexpected Behavior
LOCATION	SwapCore.sol
REVIEW	Remediation Review 2

DESCRIPTION

`buySwap()` lets callers bound the rate and the markup, but it does not let them bound the total tokens pulled from the buyer. After the slippage checks, the function calculates `requiredBuyerCollateral`, the protocol fee, the prefunded liquidation bounty and then pulls their sum from `msg.sender`.
For `BUY_FLOATING`, `rateBound` is a floor on `baseRate`. A higher base rate can pass that check because it is favorable for the fixed leg the buyer receives, but `calculateBuyerCollateral()` uses `abs(baseRate) + utilizationFee + riskPremium`. Therefore the same execution can require more posted collateral than the caller expected while still satisfying the on-chain rate bound and markup bound.
A user or adapter with a large allowance can enter a position that locks materially more collateral than intended.

RECOMMENDATION

Add explicit input caps to `buySwap()`, such as `maxBuyerCollateral` and `maxTotalIn`. Enforce them after computing `requiredBuyerCollateral`, `protocolFee` and `liquidationBounty`, but before storing the swap and transferring tokens. Adapters should expose and pass through the same bounds.

RESOLUTION

Status: Resolved.

LOW **L-04** RESOLVED

Oversized indexes hide bucket floating exposure

CATEGORY	Unexpected Behavior
LOCATION	`Utils.sol`
REVIEW	Remediation Review 2

DESCRIPTION

`Utils.addSwapToBucket()` records each swap's entry index in `weightedInverseIndex` through `Math.mulDiv(swap.notionalAmount, INITIAL_INDEX, swap.entryFloatingIndex)`. If a cumulative reference oracle reports an unusually large index, or a wrapper migration changes the index scale, this term can round down to zero for small token-native notionals.

The bucket still records nonzero notional, collateral, pool backing, fixed-rate terms, and entry time. However, `calculateBucketUnrealizedPnL()` reconstructs `avgEntryIndex` from `weightedInverseIndex` and sets it to zero when the accumulator is zero. The accrued floating rate is then skipped entirely because the code only calls `deriveRateFromIndex()` when `avgEntryIndex > 0`. `VirtualShareLib` mirrors the same memory-bucket reconstruction for virtual share pricing.

Consequently, active bucket mark-to-market can ignore real floating exposure while the swap remains open. LP share price and adapter NAV can then be overstated or understated until settlement removes the swap and prices it individually. This requires a malformed, mis-scaled or poorly migrated cumulative oracle domain; ordinary RAY-style indexes near `1e27` should not trigger the truncation for normal notionals.

RECOMMENDATION

L-04 Oversized indexes hide bucket floating exposure CONTINUED

Reject swap entry when the bucket inverse contribution would be zero:

```plain text
uint256 inverseContribution = Math.mulDiv(
swap.notionalAmount,
SwapFormulas.INITIAL_INDEX,
swap.entryFloatingIndex
);
if (inverseContribution == 0) revert BucketInverseUnderflow();
```

Also bound the maximum supported cumulative index scale at market creation and wrapper activation. If very large index domains must be supported, store the bucket inverse accumulator with higher precision and downscale only at the final PnL calculation.

RESOLUTION

Status: Resolved.

HIGH **H-01**

Cumulative index basis drains LP backing

CATEGORY	Unexpected Behavior
LOCATION	IndexBaseRateV1.sol
REVIEW	Remediation Review 4

DESCRIPTION

`IndexBaseRateV1._applyTenorBump()` converts a trailing cumulative-index TWAR into an exact-exponential tenor quote:

```solidity
baseRate(T) = (exp(r * T / YEAR) - 1) * YEAR / T
```

That generic compounding model does not match every cumulative source that can be paired with the oracle. The Aave adapters settle against Aave's supply and borrow index formulas. The Compound adapters settle against Comet's simple `accruedInterestIndices()` projection. The Morpho adapters settle against Morpho's three-term Taylor projection and, for supply, fee-share dilution.

When a BUY_FLOATING market uses IndexBaseRateV1 over one of these CumulativeIndex* adapters as the base oracle and uses the same adapter as the cumulative reference oracle, SwapCore.buySwap() stores the exact-exp quote as swap.baseRate. At settlement, the floating leg is derived from the lower source-specific cumulative index growth. The buyer therefore receives a fixed leg that the reference index cannot realize.

The issue affects the in-scope Aave, Compound and Morpho cumulative borrow and supply adapters:

- CumulativeIndexBorrowAave
- CumulativeIndexSupplyAave
- CumulativeIndexBorrowCompound
- CumulativeIndexSupplyCompound
- CumulativeIndexBorrowMorpho
- CumulativeIndexSupplyMorpho

The local PoCs use two-year BUY_FLOATING markets, 18x leverage, high source rates and 10,000,000 USDC of LP backing. In every tested adapter pairing, the stored exact-exp base rate exceeded the floating rate later derived from the same cumulative reference source. Settlement paid the buyer their posted collateral, the liquidation bounty and the full 10,000,000 USDC of locked LP backing.

The full-LP-drain impact is configuration-sensitive. Let B be the annualized IndexBaseRateV1 exact-exp base rate, R` be the annualized floating rate later derived from the

H-01 Cumulative index basis drains LP backing CONTINUED

cumulative reference index, `F` be `utilFee + riskPremium` and `L` be the market leverage multiplier. With a max-sized `BUY_FLOATING` swap and ignoring integer rounding, the condition for full LP backing loss is:

```plain text
L * (B - R - F) / B >= 1
```

Equivalently, full loss requires F <= B - R - B / L. Any buyer-positive settlement only requires F < B - R. The per-adapter sections below give the concrete model used to compute R and the exact requirements observed in the PoCs.

Adapter Analysis

1. **CumulativeIndexBorrowAave**

CumulativeIndexBorrowAave exposes Aave V3's getReserveNormalizedVariableDebt(asset) as the cumulative reference index. The adapter intentionally follows Aave's projected variable debt index, which uses Aave's binomial approximation instead of exact exponential compounding.

The issue appears when the same Aave borrow adapter is used as the IndexBaseRateV1 source and as the market's cumulative reference oracle. IndexBaseRateV1 observes the trailing Aave borrow-index growth, converts it into a continuous APR and then quotes the full swap tenor with exact expWad. Settlement later derives the floating leg from Aave's lower projected debt index. In the PoC, the stored exact-exp base rate was 3542457441174355189 WAD while the Aave borrow floating rate was 2897811585586913779 WAD. The spread was enough to pay the buyer all locked LP backing.

For this adapter, compute R from Aave's normalized-variable-debt factor over n seconds:

```plain text
AaveBorrowFactor = 1 + n * y + n * (n - 1) * y^2 / 2 + n * (n - 1) * (n - 2) * y^3 / 6
y = variableBorrowRatePerSecond
R = (AaveBorrowFactor - 1) / (n / YEAR)
```

With no fees, 18x leverage and a 2-year term, full backing loss starts when variableBorrowRate * term / YEAR >= 1.2674196508444446, which is about 63.37% APR for two years. In the PoC, variableBorrowRate * term / YEAR = 2.09. The minimum no-fee leverage for the PoC rates was 5.4952x; at 18x leverage, fees and risk premium could be as high as 0.447842664411088e18 annualized before full LP backing loss stopped.

1. **CumulativeIndexSupplyAave**

CumulativeIndexSupplyAave exposes Aave V3's getReserveNormalizedIncome(asset)` as the cumulative reference index. This is the liquidity index used for suppliers. It is a supply-side index and can grow more slowly than borrow indexes because supply yield is reduced by utilization and reserve factors.

H-01 Cumulative index basis drains LP backing CONTINUED

The same basis exists when this supply adapter feeds `IndexBaseRateV1` and also settles the floating leg. `IndexBaseRateV1` turns the observed supply-index growth into an exact-exp tenor quote. Settlement uses the actual Aave supply-index growth. In the PoC, the stored exact-exp base rate was `3536420647291896271` WAD while the Aave supply floating rate was `1045000000000000000` WAD. The buyer therefore received a fixed leg far above what the Aave supply index realized and drained all locked LP backing.

For this adapter, compute `R` from Aave's linear liquidity-index factor:

```plain text
AaveSupplyFactor = 1 + liquidityRate * term / YEAR
R = liquidityRate
```

With no fees and 18x leverage, full backing loss starts when liquidityRate * term / YEAR >= 0.11324817937107448. That is about 5.6624% APR for a 2-year term, 11.3248% APR for a 1-year term or 22.6496% APR for a 6-month term. In the PoC, liquidityRate * term / YEAR = 2.09. The minimum no-fee leverage for the PoC rates was 1.4195x; at 18x leverage, fees and risk premium could be as high as 2.2949528335534577e18 annualized before full LP backing loss stopped.

1. **CumulativeIndexBorrowCompound**

CumulativeIndexBorrowCompound reads Comet's baseBorrowIndex and projects it forward with the same simple-interest formula Comet uses in accruedInterestIndices():

```solidity
projected = borrowIndex + borrowIndex * borrowRate * elapsed / 1e18
```

This is correct for Compound V3 settlement, but it is not the model used by IndexBaseRateV1. A BUY_FLOATING market that quotes from this adapter with exact exponential compounding and settles against the same simple-projected borrow index gives the buyer a fixed leg that the Compound borrow index cannot match over an idle interval. In the PoC, the stored exact-exp base rate was 3536420647279726867 WAD while the Compound borrow floating rate was 1044999999998927949 WAD. The settlement drained all locked LP backing.

For this adapter, compute R from Comet's simple borrow-index projection:

```plain text
CompoundBorrowFactor = 1 + borrowRatePerSecond * term
R = borrowRatePerSecond * YEAR
```

With no fees and 18x leverage, full backing loss starts when borrowRatePerSecond * term >= 0.11324817937107448. Equivalently, annualizedBorrowRate * term / YEAR must be at least 0.11324817937107448. That is about 5.6624% APR for a 2-year term, 11.3248% APR for a 1-year term or 22.6496% APR for a 6-month term. In the PoC, annualizedBorrowRate * term / YEAR = 2.09. The minimum no-fee leverage for the PoC rates was 1.4195x`; at 18x

H-01 Cumulative index basis drains LP backing CONTINUED

leverage, fees and risk premium could be as high as `2.2949528335430363e18` annualized before full LP backing loss stopped.

1. **CumulativeIndexSupplyCompound**

`CumulativeIndexSupplyCompound` is the supply-side version of the same Comet integration. It reads `baseSupplyIndex` and projects it with Comet's simple supply-interest formula:

```solidity
projected = supplyIndex + supplyIndex * supplyRate * elapsed / 1e18
```

When IndexBaseRateV1 uses this adapter as its source, it converts the observed simple supply-index growth into an exact-exp tenor quote. The market then settles from the lower simple-projected baseSupplyIndex. In the PoC, the stored exact-exp base rate was 3536420647279726867 WAD while the Compound supply floating rate was 1044999999998927949 WAD. The buyer received all locked LP backing.

For this adapter, compute R from Comet's simple supply-index projection:

```plain text
CompoundSupplyFactor = 1 + supplyRatePerSecond * term
R = supplyRatePerSecond * YEAR
```

With no fees and 18x leverage, full backing loss starts when supplyRatePerSecond * term >= 0.11324817937107448. Equivalently, annualizedSupplyRate * term / YEAR must be at least 0.11324817937107448. That is about 5.6624% APR for a 2-year term, 11.3248% APR for a 1-year term or 22.6496% APR for a 6-month term. In the PoC, annualizedSupplyRate * term / YEAR = 2.09. The minimum no-fee leverage for the PoC rates was 1.4195x; at 18x leverage, fees and risk premium could be as high as 2.2949528335430363e18 annualized before full LP backing loss stopped.

1. **CumulativeIndexBorrowMorpho**

CumulativeIndexBorrowMorpho exposes Morpho Blue's borrow share exchange rate. If the Morpho market state is stale, the adapter projects totalBorrowAssets with Morpho's three-term Taylor approximation:

```solidity
wTaylorCompounded(x) = x + x^2 / 2 + x^3 / 6
```

The exact-exp quote in IndexBaseRateV1 is higher than the Taylor-projected borrow-index growth used by settlement. A same-source BUY_FLOATING market therefore stores a fixed leg above the Morpho borrow index's realizable growth. In the PoC, the stored exact-exp base rate was 3542457581112360799 WAD while the Morpho borrow floating rate was 2897802416661012885 WAD. The settlement paid the buyer all locked LP backing.

For this adapter, compute R from Morpho's Taylor borrow-asset projection:

```plain text
```

H-01 Cumulative index basis drains LP backing CONTINUED

MorphoBorrowFactor = 1 + x + x^2 / 2 + x^3 / 6
x = borrowRatePerSecond * term
R = (MorphoBorrowFactor - 1) / (term / YEAR)
``
With no fees and 18x leverage, full backing loss starts when borrowRatePerSecond * term >= 1.2674197525974966. Equivalently, annualizedBorrowRate * term / YEAR must be at least 1.2674197525974966, which is about 63.37% APR for two years. In the PoC, annualizedBorrowRate * term / YEAR = 2.09. The minimum no-fee leverage for the PoC rates was 5.4952x; at 18x leverage, fees and risk premium could be as high as 0.4478519655006615e18 annualized before full LP backing loss stopped.
1. **CumulativeIndexSupplyMorpho**
CumulativeIndexSupplyMorpho exposes Morpho Blue's supply share exchange rate. Its projection first computes borrower interest with the same Taylor approximation, then adds that interest to supply assets. If the market charges fees, the adapter also mints fee shares, diluting suppliers.
This makes the supply adapter especially incompatible with a generic exact-exp tenor quote.
IndexBaseRateV1 treats observed supply exchange-rate growth as if it can be extended with exact exponential compounding. Settlement instead uses Morpho's Taylor-projected supply assets and fee-share dilution. In the PoC, the stored exact-exp base rate was 3542457581112360799 WAD while the Morpho supply floating rate was 2897802416661012885 WAD. The buyer drained all locked LP backing.
For this adapter, compute R from the projected supply exchange-rate factor:
`plain text
interest = totalBorrowAssets * (x + x^2 / 2 + x^3 / 6)
x = borrowRatePerSecond * term
projectedSupplyAssets = totalSupplyAssets + interest
projectedSupplyShares = totalSupplyShares + feeShares
MorphoSupplyFactor = projectedExchangeRate / entryExchangeRate
R = (MorphoSupplyFactor - 1) / (term / YEAR)
`
With 100% utilization, zero Morpho fee, no Kairos fees and 18x leverage, this reduces to the same Taylor threshold as the borrow adapter: full backing loss starts when borrowRatePerSecond * term >= 1.2674197525974966, about 63.37% APR for two years. In the PoC, annualizedBorrowRate * term / YEAR = 2.09. The minimum no-fee leverage for the PoC rates was 5.4952x; at 18x leverage, fees and risk premium could be as high as 0.4478519655006615e18 annualized before full LP backing loss stopped. Lower utilization reduces the supply exchange-rate growth and therefore requires a higher borrow-rate-times-term product. Morpho fee-share dilution reduces the supply exchange-rate growth at settlement, so it must be included in MorphoSupplyFactor` rather than treated as a protection by itself.

H-01 Cumulative index basis drains LP backing CONTINUED

RECOMMENDATION

Do not use generic exact-exponential `IndexBaseRateV1` tenor quotes for markets settled from protocol-specific cumulative indexes unless the quote model matches the reference index model.
The safest fix is to add a listing and wrapper-retarget guard that rejects `BUY_FLOATING` markets where `IndexBaseRateV1` is sourced from the same `CumulativeIndex*` adapter used as the cumulative reference oracle. If those markets are required, deploy source-specific base-rate oracles that quote tenors with the same model used by settlement: Aave's index math for Aave adapters, Comet's `accruedInterestIndices()` formula for Compound adapters and Morpho's Taylor compounding plus supply fee-share dilution for Morpho adapters.

PROOF OF CONCEPT https://gist.github.com/GuardianAudits/bebc0ea30c7ad373086748ab184e745d

RESOLUTION

Status: Reported during the fix review period.

HIGH **H-02**

Interim liquidation drains LP backing

CATEGORY	Math
LOCATION	`Utils.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`calculateLiquidationObligation()` values a live swap by deriving the elapsed floating rate and then passing the elapsed `swapDuration` into `_calculatePayments()`. For `BUY_FLOATING`, `_calculatePayments()` uses the stored `swap.baseRate` as the fixed leg and `SwapFormulas.calculateSwapPayment()` prorates that annual rate linearly over the elapsed time.
That is unsafe for markets whose base oracle returns a full-tenor compounded-equivalent quote. `IndexBaseRateV1._applyTenorBump()` returns `(exp(r * T / YEAR) - 1) * YEAR / T` for the whole swap term. At maturity this matches an exact exponential cumulative reference. Before maturity, however, the liquidation check values the fixed leg as `(t / T) * (exp(r * T / YEAR) - 1)` while the cumulative floating leg realizes `exp(r * t / YEAR) - 1`. The first expression is larger for positive convex growth, so the pool can appear liquidatable even though the full-term swap is fairly priced.
A local test used an exact exponential cumulative reference, the real `IndexBaseRateV1` and `SwapCore`. The stored full-term base rate was `3542457582152521274` WAD and the maturity floating rate was `3542457582152530087` WAD, showing the maturity quote is effectively fair. At half term, the buyer self-liquidated a maximum-sized `BUY_FLOATING` swap and received their prefunded collateral plus all `10,000,000` USDC of LP backing, leaving `totalCollateral` and `lockedCollateral` at zero.
This is distinct from the prior cumulative-index basis findings because it does not require Aave, Compound or Morpho to realize less than the exact exponential quote at maturity. The loss comes from applying a full-tenor compounded-equivalent rate as a simple pro-rata accrued fixed payment during liquidation.

RECOMMENDATION

Do not use the full-tenor compounded-equivalent `baseRate` directly for interim liquidation accrual. Store or derive a rate representation that is linear over elapsed time or compute the elapsed fixed leg from the same compounded curve and elapsed duration used by the base oracle. For `IndexBaseRateV1`-backed markets, the interim fixed payment should use `exp(r * elapsed / YEAR) - 1`, not `elapsed / swapTerm * (exp(r * swapTerm / YEAR) - 1)`.

H-02 Interim liquidation drains LP backing CONTINUED

PROOF OF CONCEPT https://gist.github.com/GuardianAudits/3c8ca1cdee9c6096d2b7f17f7dd8ba61

RESOLUTION

Status: Reported during the fix review period.

HIGH **H-03**

Dead-oracle fallback drains LP backing

CATEGORY	Unexpected Behavior
LOCATION	`Utils.sol`
REVIEW	Remediation Review 4

DESCRIPTION

Normal-expiry settlement now has a dead-oracle recovery path: `_settleSwaps()` first tries `updateMarketRateIndex()` and then calls `Utils.resolveExpiryIndex()` when no snapshot exists at the swap maturity. After `settlementGracePeriodSec`, if the guaranteed retry still cannot read the oracle, `resolveExpiryIndex()` settles with `RateIndexLib.getIndexAt(rateIndex, refOracle, actualExpiry)` and terminates the market.

That fallback value is not a conservative last-known-good index. When maturity is after the latest stored snapshot, `getIndexAt()` calls `_extrapolate()` and linearly extends the slope between the last two pre-failure snapshots to the maturity timestamp. A short-lived pre-outage trend can therefore be projected across the missing interval and used as the final floating-leg settlement value even though no real oracle data exists at or after maturity.

This turns a reference-oracle outage into value transfer rather than only liveness recovery. A buyer whose position benefits from the stale trend can wait until the grace window elapses and permissionlessly settle against the extrapolated index. If the extrapolated net obligation exceeds `swap.poolCollateralBacking`, settlement pays the buyer the full LP backing cap and clears the lock, even when a last-known-index fallback would have paid far less.

LPs can lose the full collateral backing of affected swaps during a reference-oracle outage. The loss is capped per swap by `poolCollateralBacking`, but an attacker or exposed buyer can size positions against available market liquidity before the outage and settle all eligible in-the-money positions after the grace period. The fallback also permanently terminates the market, so the loss is realized as part of recovery rather than blocked by the old fail-closed behavior.

For this issue to occur, the oracle does not need to be malicious. The precondition is an outage after at least two pre-maturity snapshots and before any snapshot at or after maturity. The last pre-outage slope only needs to be buyer-favorable enough that extrapolating it to maturity exceeds the LP backing cap. This is realistic for cumulative/index references during volatile rate periods or integration outages.

Attack scenario:

1. A BUY_FIXED position is opened while LP liquidity is available.

H-03 Dead-oracle fallback drains LP backing CONTINUED

2. Before maturity, the reference index moves upward and `updateMarketRateIndex()` records a pre-failure snapshot.
3. The reference oracle becomes unreadable before any post-maturity snapshot can be written.
4. Settlement reverts with `E450` until `expiry + settlementGracePeriodSec`.
5. After grace, `resolveExpiryIndex()` retries the oracle, observes the read still fails and settles from `getIndexAt()`'s extrapolated maturity index.
6. The buyer receives the capped LP backing even though the last known snapshot alone would not justify that payout.

RECOMMENDATION

Keep the post-grace recovery so a dead reference oracle cannot permanently pin expired swaps, but separate recovery from valuation. When the adjudicated retry fails and `lastUpdate < actualExpiry`, `resolveExpiryIndex()` should not call `RateIndexLib.getIndexAt()` because that function extrapolates from the last two snapshots.
Use a dedicated no-extrapolation fallback instead. The simplest safe option is to settle from the latest real snapshot at or before the oracle failure and then terminate the market for new swaps. If the protocol wants a fairer recovery value, add a one-shot governance or guardian adjudication that publishes a final expiry index after the grace period. That value should be immutable per `(referenceOracle, actualExpiry)`, reused for every swap with that maturity, emitted in an event and bounded against the last real snapshot by an explicit maximum drift. A real oracle update must always take precedence over the fallback.

RESOLUTION

Status: Reported during the fix review period.

`https://github.com/kairosswap/protocol/pull/974`

HIGH **H-04**

Early exit drains LP backing via linear accrual

CATEGORY	Math
LOCATION	Utils.sol
REVIEW	Remediation Review 4

DESCRIPTION

`SwapCore.exitSwapEarly()` is a public settlement path for swap owners and authorized callers. It marks each requested swap as an early exit and delegates to `_settleSwaps()`, which fetches a fresh remaining-tenor base rate and reference index before calling `Utils.settleSwap()`.
For early exits, `Utils.calculateEarlyExitSettlement()` computes both the accrued-to-date leg and the projected remaining leg through `_calculatePayments()`:

```solidity
(int256 accruedFixedPayment, int256 accruedFloatingPayment) = _calculatePayments(
market,
swap,
swapDuration,
currentFloatingRate
);
```

For `BUY_FLOATING`, `_calculatePayments()` always values the fixed leg by linearly prorating the stored full-tenor `swap.baseRate` over the requested duration:

```solidity
fixedPayment = SwapFormulas.calculateSwapPayment(swap.notionalAmount, swap.baseRate, duration, 0, 0);
```

That is inconsistent with the protocol's current compounded-accrual handling for cumulative markets. `IndexBaseRateV1` advertises that its tenor quotes are full-tenor compounded-equivalent rates and `_calculateAccruedPayments()` explicitly switches to `SwapFormulas.accruedFixedLeg()` for cumulative markets whose base oracle advertises `isCompoundedTenorQuote()`. The source comments state the reason: straight-line proration of a compounded-equivalent quote overstates interim fixed-leg value before maturity.
The early-exit path still bypasses `_calculateAccruedPayments()` and uses the old straight-line accrued leg. In an early-exit-enabled `BUY_FLOATING` cumulative market backed by a

H-04 Early exit drains LP backing via linear accrual CONTINUED

compounded-equivalent base oracle such as `IndexBaseRateV1`, the buyer can wait until the remaining-tenor quote moves down enough for `projectedFavorsBuyer` to become true. The remaining leg is then forfeited and only the accrued leg is settled. At that point the overstated straight-line accrued fixed leg is no longer offset by any remaining-term calculation, so the buyer can call the public `exitSwapEarly()` path and receive LP backing solely from the linear-vs-compounded accrued-leg gap.
The local PoC opens a one-year `BUY_FLOATING` swap against a compounded-equivalent full-term quote, advances half the term and sets the cumulative reference index to the matching exact exponential half-term value. `getSwapNetAmount()` reports a near-zero accrued mark under the corrected compounded-accrual logic. After the remaining-tenor base quote is lowered to zero, the buyer calls `exitSwapEarly()` and receives posted margin plus the full locked LP backing. The pool collateral falls by that backing amount while LP shares are unchanged. This is direct token loss from the affected pool's collateral balance; it scales with notional and locked LP backing, with blast radius bounded by affected market liquidity and open-position backing.

RECOMMENDATION

Make early-exit settlement split compounded-equivalent fixed legs consistently, but do it in a way that preserves the existing full-term settlement invariant.
The accrued component should use the same logic as `_calculateAccruedPayments()` when the market is cumulative and the base oracle advertises a compounded-equivalent quote. The remaining fixed component should not be computed by calling the same compounded-accrual helper over `timeRemaining`; that would value the remaining interval as a fresh curve from zero and would not necessarily add back to the contractual full-term fixed payoff.
Instead, derive the remaining fixed leg as:

```solidity
fullFixedPayment = fixed leg over market.swapTerm using the stored entry fixed rate;
remainingFixedPayment = fullFixedPayment - accruedFixedPayment;
```

For BUY_FIXED, fullFixedPayment should match the current maturity settlement fixed leg using swap.swapRate. For BUY_FLOATING, it should use the stored swap.baseRate. This preserves the invariant:

```plain text
correctedAccruedFixed + remainingFixed == fullTermFixed
```

Then keep the existing early-exit rule:
- if the remaining term favors the buyer, forfeit the remaining leg and settle only the corrected accrued fixed/floating payments;
- if the remaining term favors the pool, include remainingFixedPayment` and the existing

H-04 Early exit drains LP backing via linear accrual CONTINUED

projected remaining floating leg, including the cumulative floating cross-term.
For minimum behavioral risk, this corrected split can be applied only when `convention == Types.RateConvention.Cumulative && _baseQuoteCompounded(market.baseSwapRateOracle)`. For all other markets, the existing linear split can be preserved exactly except for harmless rounding-dust improvements if the team chooses to consolidate the code path.
Also update early-exit event/accounting values to use the same corrected fixed/floating split, or return the gross fixed/floating totals from `calculateEarlyExitSettlement()`. Otherwise settlement can be fixed while `SwapClosed` continues to report the old straight-line accrued fixed amount for early exits.

PROOF OF CONCEPT https://gist.github.com/GuardianAudits/0acabde10c5c144d98ca61c595ea470e

RESOLUTION

Status: Reported during the fix review period.

https://github.com/kairosswap/protocol/pull/978

HIGH **H-05**

LP adapter NAV ignores liquidation floor

CATEGORY	Unexpected Behavior
LOCATION	`KairosMarketAdapter.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`KairosMarketAdapter._realAssets()` values the vault's LP position as `shares * swapCore.getPoolSharePriceVirtual(marketId) / WAD`. That share price can still value live pool-side-liquidatable swaps through the normal active-bucket mark. The bucket mark is capped at `totalPoolBacking`, but it does not switch to the value that would remain after an immediately executable `liquidateSwap()`.

If a live swap becomes pool-side liquidatable before liquidation is executed, a Morpho-style vault can price shares from idle cash plus an LP adapter NAV that is above the liquidation floor. This window is realistic because liquidation is permissionless, vault share pricing reads adapter NAV before liquidation is mined and redemptions can be served from idle vault cash without unwinding the Kairos LP position first. A fast shareholder can therefore redeem at the inflated vault share price, receive too much idle cash and leave the crystallized liquidation loss with the shareholders who remain in the vault.

In the validated scenario, a vault has 50,000,000 USDC from a victim, 10,000,000 USDC from an attacker and a 10,000,000 USDC LP allocation through `KairosMarketAdapter`. After a max-sized swap becomes pool-side liquidatable, the adapter reports 22,109,999.999998 USDC of NAV even though immediate liquidation leaves only 0.000001 USDC in the pool. The attacker redeems before liquidation for 12,018,333.333333 USDC. At the liquidation floor, the same shares are worth 8,333,333.333333 USDC, so the attacker extracts 3,685,000 USDC-equivalent from the remaining vault shareholder.

This root cause was already flagged in the resolution section of [M-10] (/p/33e8bda5828c817199faf38a76774728?v=3178bda5828c81db8475000c828caca3&pvs=25) and acknowledged there as the remaining open pool-liquidation side: bucket share pricing still caps LP losses against full `bucket.totalPoolBacking` even though pool-side liquidation removes value for the liquidator. This writeup should therefore be treated as a stronger impact evidence for that acknowledged issue, not as a separate root cause. The additional evidence is that the LP adapter/Morpho vault surface, where `KairosMarketAdapter._realAssets()` feeds the overstated share price into redeemable vault NAV and lets idle cash be withdrawn before liquidation crystallizes the loss.

H-05 LP adapter NAV ignores liquidation floor CONTINUED

RECOMMENDATION

Fix the shared LP share-price calculation rather than special-casing only the adapter. The accounting price used by `SwapCore.getPoolSharePrice()`, `SwapCore.getPoolSharePriceVirtual()`, LP mint/burn paths and `KairosMarketAdapter._realAssets()` should be derived from the value that can survive the protocol's currently executable close path.

For each live unsettled swap or an equivalent conservative per-swap valuation path, reuse the same accrued obligation and liquidation predicate used by `Utils.isSwapLiquidatable()`. If the pool side is liquidatable, value that swap as a full LP loss of `swap.poolCollateralBacking`, because `Utils.liquidateSwap()` removes the entire backing from the pool: `poolBacking - liquidatorReward` is paid to the buyer and `liquidatorReward` is paid to the liquidator. If the buyer side is liquidatable, value the swap as an LP gain of `swap.collateralBalance`, because the buyer's liquidator bounty is prefunded outside `collateralBalance`. Non-liquidatable swaps should still be capped by the settlement amounts that can actually be collected.

Apply this rule consistently in both the storage share-price path in `Utils` and the memory/virtual path in `VirtualShareLib`. If exact per-swap liquidation-aware pricing is too expensive for on-chain accounting, fail closed instead: make LP minting, LP burning and adapter NAV reads revert until executable pool-side liquidations are processed or use a deliberately conservative value that cannot exceed the immediate post-liquidation pool value. Do not allow vault share minting or redemption to continue from the optimistic active-bucket mark once liquidation can be executed immediately.

PROOF OF CONCEPT https://gist.github.com/GuardianAudits/f16c696ecbb3076f1d4f992d2db2d28d

RESOLUTION

Status: Reported during the fix review period.

https://github.com/kairosswap/protocol/pull/981

HIGH **H-06**

Force-dealloc forfeits buyer liquidation value

CATEGORY	Validation
LOCATION	`KairosMarketAdapter.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`KairosMarketAdapter.deallocate()` allows a permissionless `forceDeallocate()` caller to make a partial LP withdrawal whenever `fullAmount` is false and the profit-vest cap is not binding. That guard is too narrow. It does not cover a live swap that is already buyer-liquidatable while `getPoolSharePrice()` is still below the value the pool would have immediately after liquidation. This happens because active LP MTM is computed from projected full-term bucket payments and only capped by aggregate buyer collateral. It does not check whether a specific buyer can already be liquidated. In contrast, `Utils.liquidateSwap()` transfers all of the buyer's posted collateral to the pool and removes the swap from bucket accounting. Therefore the current share price can understate the pool's immediately executable recovery.

A co-LP can call `forceDeallocate()` before liquidation and choose a partial `assets` amount. `SwapCore.withdrawCollateral()` burns the vault adapter's shares at the understated share price and returns exactly that amount to the vault. The caller then liquidates the buyer-side swap, causing the recovery to be distributed over fewer vault shares and more remaining co-LP shares. A focused Foundry PoC in `test-foundry/zz_POCs/BuyerSideLiquidationForceDeallocatePoc.t.sol` validates the issue. In the test, a vault has a 5,000,000 USDC LP allocation and an outside co-LP also has 5,000,000 USDC in the same market. A third-party buyer opens a 500,000,000 USDC notional swap. After two days the buyer is liquidatable, but `getPoolSharePrice()` is only `0.945205479452` while immediate buyer liquidation would price the pool at `1.055218615124`8. The outside co-LP calls `forceDeallocate()` for `2,363,013.698630` USDC, burning half of the vault adapter's shares at the stale-low mark. After liquidation, the vault holds only `5,092,737.849502` USDC of cash plus LP value.

The burned share slice would be worth `2,729,724.150872` USDC at the final attack-branch share price, so the vault receives `366,710.452243` USDC less than that slice's final value. The same PoC also runs a clean no-force counterfactual from the exact same pre-liquidation state. If liquidation executes without the forced burn, the vault ends with `5,276,093.075624` USDC of LP value. Under the force-deallocate sequence it ends with `5,092,737.849502` USDC of cash plus LP value, so vault depositors are `183,355.226122` USDC worse off

H-06 Force-dealloc forfeits buyer liquidation value CONTINUED

while the force-deallocating co-LP's remaining LP shares are `183,355.226121` USDC better off. The buyer and liquidator outcomes are unchanged because liquidation still consumes the same buyer collateral and pays the same prefunded bounty.
Prior force-deallocation findings cover different root causes: vest-cap forfeiture, dust share-burn rounding or fully exiting through a partial request. Those checks compare the burned shares against the current share price. They do not address this case because the current share price itself omits an immediately executable buyer-side liquidation recovery.

RECOMMENDATION

Make permissionless `forceDeallocate()` fail closed while the adapter's market contains any live buyer-liquidatable swap whose immediate liquidation recovery would increase LP share price. The same liquidation-aware valuation should be used by `Utils.calculateBucketUnrealizedPnL()` and `VirtualShareLib`, so LP deposits, withdrawals and adapter NAV all reflect executable buyer-side liquidations before shares can be minted or burned.
If exact per-swap liquidation-aware pricing is too expensive, use a conservative guard for `KairosMarketAdapter`: block partial `forceDeallocate()` whenever an active market may contain executable liquidations and require allocator-controlled deallocation after liquidation processing.

PROOF OF CONCEPT https://gist.github.com/GuardianAudits/ea15a530fc1aa7ab65960031267fd4ee

RESOLUTION

Status: Reported during the fix review period.

https://github.com/kairosswap/protocol/pull/981

MEDIUM M-01

RESOLVED

LP MTM Omits Cumulative Cross-Term

CATEGORY	Logical Error
LOCATION	`Utils.sol, VirtualShareLib.sol`
REVIEW	Remediation Review 4

DESCRIPTION

Active LP share pricing still omits the Cumulative-convention remaining floating-leg cross-term that early-exit settlement now applies.

Under the Cumulative convention, index growth is multiplicative. The accrued leg is priced from entryIndex to currentIndex, while the remaining leg is projected from the base rate. To match the full-term floating leg that settlement realizes, the fee-free remaining floating leg must be scaled by the already-realized currentIndex / entryIndex growth.

The recent fix for "Early exits omit current index multiplier" added this term to calculateEarlyExitSettlement(), but active LP MTM still uses the pre-fix composition. calculateBucketUnrealizedPnL() adds accruedFloatingPayment + remainingFloatingPayment directly. The virtual share-price path used by KairosMarketAdapter.realAssets() mirrors the same logic in VirtualShareLib, so Morpho LP adapter NAV inherits the stale valuation while buckets are active.

This misprices LP shares until early exit or settlement realizes the corrected value. In BUY_FLOATING markets with positive cumulative growth and positive projected remaining rates, shares are understated, letting new LP deposits mint too many shares and dilute existing LPs once hidden value realizes. In BUY_FIXED markets, the sign flips: shares are overstated, allowing withdrawals against inflated NAV before remaining LPs realize the loss.

For example, if the cumulative index has already grown from 1.0x to 1.149x, and the remaining projected floating rate is another 14.95%, the full floating leg should compound to 1.149 * 1.1495 = 1.3207755. The LP MTM path instead adds the two pieces linearly as 14.9% + 14.95% = 29.85%, missing the 2.22755% cross-term.

RECOMMENDATION

Apply the same Cumulative cross-term adjustment in active bucket LP MTM and the VirtualShareLib memory path. Recompute the fee-free remaining floating base, add cumulativeRemainingCrossTerm(remainingBase, avgEntryIndex, oracleIndex, convention), and keep util/risk fee accrual simple as settlement does.

M-01 LP MTM Omits Cumulative Cross-Term CONTINUED

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/876

MEDIUM **M-02** RESOLVED

Stale anchors let top-ups bypass vesting

CATEGORY	Unexpected Behavior
LOCATION	`SwapCore.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`SwapCore` stores only one profit-vesting anchor for each LP position. When an LP tops up a position that still has any shares and is still inside its vesting window, `supplyCollateral()` mints the new shares at the current share price but keeps the old `entryPrice`.

That is unsafe when the old anchor is higher than the current share price. The top-up receives shares at the lower price, but those shares inherit the old higher anchor. `withdrawCollateral()` only applies the vesting cap when the current share price is above `entryPrice`, so any recovery from the top-up price up to the old anchor is treated as uncapped value.

Consequently, fresh LP capital can earn and withdraw recovery gains during its vesting window simply by joining an existing position with a stale high anchor. A standalone deposit at the same lower share price would have been anchored to that lower price and capped during the same recovery. This lets the topping-up LP dilute other LPs' exposure to the recovery that came from pre-existing pool risk.

For example, an LP deposits when the share price is `1.0`, then the pool moves down while the LP is still vesting. The LP tops up at the lower price, but because the position still has shares, `entryPrice` remains `1.0`. If the pool later recovers to `0.99`, the cap is still inactive because `0.99 < 1.0`. The fresh top-up shares nevertheless gained value from their lower mint price and that gain can be realized before the fresh capital has actually vested.

RECOMMENDATION

Track profit vesting per deposit tranche. Existing shares can keep their previous `entryPrice` and vest expiry, but newly minted top-up shares should receive the current `sharePrice` and their own vest expiry.

If per-tranche accounting is too heavy, use a conservative aggregate basis that does not let below-anchor top-ups inherit a higher anchor. For example, store a share-weighted entry price for the combined position, rounded in the pool's favor or reject mid-vest top-ups when the current `sharePrice` is below the stored `entryPrice`. The fix should still prevent upward anchor ratchets, but it should not let fresh capital use stale high anchors.

M-02 Stale anchors let top-ups bypass vesting CONTINUED

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/880

MEDIUM M-03

RESOLVED

Force-deallocation can burn capped LP shares

CATEGORY	Unexpected Behavior
LOCATION	`KairosMarketAdapter.sol, SwapCore.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`KairosMarketAdapter.deallocate()` allows the parent Vault V2 to call `ACTION_WITHDRAW` through both allocator-controlled `deallocate()` and permissionless `forceDeallocate()`. The adapter blocks `fullAmount=true` when the call comes from `forceDeallocate()`, but it still accepts `fullAmount=false` with any nonzero `assets` value.

That guard is incomplete when the adapter's Kairos LP position is inside an active profit-vest window. During that window, `KairosMarketAdapter._realAssets()` reports the value the adapter could realize immediately, not the uncapped mark-to-market value. If the current share price is above the position's `entryPrice`, `_realAssets()` floors the share price to `entryPrice`. `SwapCore.withdrawCollateral()` applies the same cap before deciding whether the requested `amount` is large enough to redeem the whole position.

This means a public `forceDeallocate()` caller can pass `fullAmount=false` while setting `assets` equal to the adapter's capped full-position value. The adapter forwards that amount to `SwapCore.withdrawCollateral()`. Since the amount is equal to the capped full value, `SwapCore` treats it as a full withdrawal and burns all of the adapter's LP shares at the capped price.

Consequently, an outsider can force a Morpho vault to exit its Kairos LP position at the vest-capped value even when the fair mark-to-market share price is higher. The vault loses exposure to any value above the cap and cannot wait for the vest window to expire. If the caller or an affiliated account is also a remaining LP in the same market, the value not paid to the vault stays in the pool and can accrue to the remaining LP shares.

RECOMMENDATION

M-03 Force-deallocation can burn capped LP shares CONTINUED

Do not allow permissionless `forceDeallocate()` to crystallize a cap-active Kairos LP position. The adapter should reject `ACTION_WITHDRAW` through `forceDeallocate()` when the position is still vesting and the uncapped share price is above `entryPrice`.
If a public force-deallocation mechanism is required, make it explicitly bounded. A permissionless caller should not be able to burn all adapter shares through the exact-amount withdrawal interface unless an authorized allocator supplied the relevant slippage and share-burn limits. A simpler conservative fix is to reject permissionless withdrawals that would redeem the adapter's entire Kairos LP balance.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/878

MEDIUM **M-04** RESOLVED

Morpho bad debt freezes supply-rate high-water

CATEGORY	Oracle
LOCATION	`CumulativeIndexSupplyMorpho.sol, IndexBaseRateV1.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`CumulativeIndexSupplyMorpho` reports the Morpho supply share exchange rate as a cumulative index. That share ratio can decrease after real supply-side losses, including bad debt. `IndexBaseRateV1` then clamps any lower source index to the previously stored value in both `poke()` and `_realizedTwar()`.
That clamp is useful for tiny rounding drift, but it treats a material supply loss the same way. If bad debt reduces the Morpho supply share ratio, the base-rate oracle keeps using the old pre-loss high-water mark. Later supply yield is not recognized until the share ratio climbs above that old high.
Consequently, markets using the affected Morpho supply base-rate oracle can receive stale or understated supply-rate quotes after a real supply share-price decrease. The oracle hides the loss and also hides real post-loss recovery below the old high-water mark. This can distort swap pricing, LP mark-to-market and any integration that expects the base-rate oracle to reflect realized Morpho supply yield.
For example, if the Morpho supply share ratio reaches `1.10x` and later falls to `1.05x` after bad debt, `IndexBaseRateV1` continues to price from `1.10x`. If the market then recovers from `1.05x` to `1.09x`, that recovery is real yield for post-loss suppliers, but the oracle still reports zero growth because the old `1.10x` high has not been exceeded.

RECOMMENDATION

Do not apply an unbounded high-water clamp to Morpho supply sources. Clamp only decreases that are provably within a small rounding tolerance.
Material decreases should either make the oracle invalid until an operator reseeds it or be represented through an explicit epoch/reset mechanism so future yield is measured from the new economic baseline. If the base-rate oracle must remain monotone, the Morpho supply adapter should expose enough information for the wrapper to distinguish harmless rounding drift from real supply-side losses.

M-04 Morpho bad debt freezes supply-rate high-water CONTINUED

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/879

MEDIUM **M-05** ACKNOWLEDGED

Spot-rate index floor erases negative accrual

CATEGORY	Logical Error
LOCATION	`SwapFormulas.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`SpotRate` markets accumulate signed rate-time in `RateIndexLib`, but settlement does not store or compare the signed accumulator for each swap. Instead, `SwapFormulas.indexFromCumulativeRateTime()` converts the accumulator into an unsigned index and clamps that index to `MIN_INDEX`. Once the accumulator has fallen far enough to hit the floor, additional negative accrual produces the same closing index.

`SwapFormulas.deriveRateFromIndex()` later receives only the entry index and the closing index. For `SpotRate`, it derives the raw index-ratio rate and then scales it by `entryFloatingIndex / INITIAL_INDEX`. If a swap is opened shortly before the global spot index reaches the floor, the closing index can no longer represent how far below the floor the signed accumulator moved during the swap.

The entry guard in `SwapCore.buySwap()` only rejects new positions when the current index is already at or below `MIN_INDEX`. It does not protect positions opened while the index is still slightly above the floor. For example, a position can enter when the index is about `1%` of `INITIAL_INDEX`, then close after enough negative rate accrual to push the signed accumulator past the floor. The closing index is fixed at `MIN_INDEX`, so the derived floating rate only reflects the remaining visible index decline instead of the full signed rate-time change.

Consequently, existing swaps can understate negative floating-rate accrual during settlement, early exit and liquidation checks. In a `BUY_FIXED` market, this can reduce the buyer's fixed-minus-floating obligation, return collateral that should have been paid to LPs and delay or prevent liquidation even when the buyer's true obligation is larger than the protocol records.

RECOMMENDATION

For spot-rate conventions, preserve the signed cumulative rate-time value that backs the index. Store each swap's entry accumulator as well as its entry index, then derive the realized spot rate from the difference between the signed closing accumulator and signed entry accumulator.

M-05 Spot-rate index floor erases negative accrual CONTINUED

Historical expiry pricing should resolve the signed accumulator at expiry, not only the floored unsigned index. If interpolation is needed, interpolate the signed accumulator snapshots. Keeping the entry floor check is still useful as a temporary guard, but it is not a complete fix because it cannot protect positions that were opened before the floor was reached.

RESOLUTION

Status: Acknowledged.

MEDIUM **M-06** RESOLVED

Vest-Capped Adapter NAV Dilutes Vault LPs

CATEGORY	Logical Error
LOCATION	`KairosMarketAdapter.sol, SwapCore.sol`
REVIEW	Remediation Review 4

DESCRIPTION

KairosMarketAdapter.realAssets() reports the adapter's LP position at the profit-vest-capped withdrawal value instead of the uncapped mark-to-market value.

The LP profit vest cap is meant to limit what a Kairos LP can realize through an immediate withdrawal while its vest window is active. SwapCore.withdrawCollateral() applies the cap by flooring the burn price to entryPrice when the current share price is higher. The recent LP profit vest fix mirrored that same cap into KairosMarketAdapter._realAssets(), so the adapter reports shares * entryPrice while the position is cap-active, even though the adapter still owns LP shares whose full value can be realized after vest expiry.

That capped value is unsafe as parent-vault NAV. A Morpho vault or any vault accounting layer that prices shares from adapter.realAssets() will understate total assets whenever the adapter has unvested LP profit. New depositors can enter against the temporarily capped NAV, mint too many vault shares, and later capture part of the pre-existing LP profit once the vest expires or the adapter deallocates. Existing vault shareholders are diluted even though the profit was generated before the new depositor entered.

For example, Alice deposits into a vault and the vault allocates to Kairos through the adapter. The Kairos LP position becomes profitable while the adapter is still inside its vest window, so the uncapped LP share value is above entryPrice. adapter.realAssets() nevertheless reports the lower capped value. Bob then deposits into the vault and receives shares priced against the capped NAV. After vest expiry, the adapter's full LP value becomes realizable, and Bob's vault shares are worth more than his deposit, paid out of Alice's pre-existing hidden profit.

RECOMMENDATION

Separate adapter fair-value NAV from immediately withdrawable value. Parent vault share pricing should use uncapped LP MTM, while deallocation and withdrawal paths can continue using the vest-capped realizable value. If the vault cannot distinguish those values, block deposits/mints while the adapter has cap-active hidden profit or add vault-level profit vesting so newly minted vault shares do not participate in pre-existing unvested LP gains.

M-06 Vest-Capped Adapter NAV Dilutes Vault LPs CONTINUED

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/878

MEDIUM M-07

RESOLVED

Source rollback quotes frozen index

CATEGORY	Logical Error
LOCATION	IndexBaseRateV1.sol
REVIEW	Remediation Review 4

DESCRIPTION

`IndexBaseRateV1` does not apply the same high-water rule in its read logic that it applies when observations are written. `poke()` clamps any source-index decrease to the latest stored head. `_realizedTwar()` only clamps the live source index up to the left anchor or the interpolated historical index. It never checks the live source index against the newest stored head.

This creates a mismatch when the cumulative source has previously reached a high value and later rolls back. The settlement-side cumulative index can remain frozen at the prior high-water mark, while `IndexBaseRateV1` can still quote a positive base rate from a lower live value as long as that value is above the older historical anchor.

Consequently, a market that uses the same rollback-prone cumulative source for settlement and base-rate pricing can accept a `BUY_FLOATING` swap with a positive fixed leg while the floating settlement index remains clamped at the prior high-water mark. The buyer can receive the fixed leg while paying little or no floating leg until the source recovers above the old high. This is most relevant for sources that are only approximately monotonic or whose share ratio can materially decrease.

RECOMMENDATION

Make `IndexBaseRateV1` reject material source rollbacks instead of treating every decrease as harmless rounding drift. The read logic should compare the live source index against the newest stored head and return invalid when the rollback exceeds an explicit, economically negligible tolerance.

If small rounding decreases must remain supported, keep the clamp only inside that configured tolerance. Larger decreases should require reseeding, an explicit epoch reset or a source-specific adapter that can distinguish harmless rounding from a real economic reset.

RESOLUTION

Status: Resolved.

M-07 Source rollback quotes frozen index CONTINUED

https://github.com/kairosswap/protocol/pull/879

MEDIUM M-08

ACKNOWLEDGED

Morpho cadence skews spot accrual

CATEGORY	Oracle
LOCATION	`ReferenceRateMorpho.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`ReferenceRateMorpho` reads Morpho's `IIrm.borrowRateView()` and returns the result as a normal `IRateOracle` spot rate. That is unsafe for Morpho Adaptive Curve markets. In that IRM, `borrowRateView()` returns the average borrow rate over the interval since `market.lastUpdate`, not the instantaneous borrow rate at the current block.

Kairo handles `SpotRate` and `SpotCompoundRate` reference oracles by multiplying the current `getRate()` value by only the time elapsed since Kairo's own previous index update. Anyone can call `SwapCore.updateMarketRateIndex()`, which means the same Morpho market state can produce different Kairo floating indexes depending on how often the Kairo index is refreshed. Repeated updates apply overlapping Morpho prefix averages to successive Kairo subintervals instead of integrating disjoint rate intervals.

Consequently, settlement becomes update-frequency dependent when `ReferenceRateMorpho` is used with a spot-style rate convention for a Morpho market whose IRM changes with elapsed time. If the adaptive borrow rate is rising, frequent Kairo updates understate floating accrual compared with one update at expiry. A `BUY_FLOATING` trader can reduce the floating leg they pay while still receiving the fixed leg. If the adaptive rate is falling, the distortion reverses and can favor `BUY_FIXED` positions.

This requires the referenced Morpho market to use an elapsed-time-dependent IRM, such as Adaptive Curve and to remain unaccrued in Morpho for a meaningful interval. It does not require oracle signer control, governance access or privileged Kairo permissions.

RECOMMENDATION

Do not use `ReferenceRateMorpho` as a `SpotRate` or `SpotCompoundRate` reference oracle for Morpho Adaptive Curve markets.

Prefer `CumulativeIndexBorrowMorpho` with `RateConvention.Cumulative` or implement a Morpho adapter that exposes a projected cumulative borrow index instead of a prefix-average borrow rate. If a spot-rate Morpho adapter remains available, restrict it to IRMs whose read value is a true current spot rate and reject incompatible oracle/convention combinations during market creation.

M-08 Morpho cadence skews spot accrual CONTINUED

RESOLUTION

Status: Acknowledged.

MEDIUM **M-09**

Dust force-dealloc burns valuable LP shares

CATEGORY	Rounding
LOCATION	`KairosMarketAdapter.sol`
REVIEW	Remediation Review 4

DESCRIPTION

Dust force-dealloc burns valuable LP shares

RECOMMENDATION

Do not let permissionless `forceDeallocate()` realize a large gap between the value of adapter LP shares burned and the `assets` returned to the vault.
For partial `forceDeallocate()` withdrawals, compute the shares that `SwapCore.withdrawCollateral()` would burn at the current share price. Reject the call unless the fair value of those shares is close to the requested `assets` amount, allowing only a small rounding tolerance. A stricter fix is to block partial `forceDeallocate()` withdrawals for `KairosMarketAdapter` and require allocator-controlled deallocation. Do not rely on the caller-provided `maxSharesToRedeem` field for this protection. A caller can choose a cap that still permits the small rounded share burn while preserving the value transfer.
Description
`KairosMarketAdapter.deallocate()` allows permissionless `forceDeallocate()` partial withdrawals whenever `_capBinding()` is false. In that case the permissionless caller chooses the `assets` amount and the adapter forwards that exact amount to `SwapCore.withdrawCollateral()`.
`SwapCore.withdrawCollateral()` handles exact-amount partial withdrawals by burning `ceil(assets * WAD / sharePrice)` LP shares and returning exactly `assets`. This rounding is normally harmless when the caller owns the LP position, because the caller bears the cost of burning any rounded-up share. Here the caller does not own the adapter's LP shares. The adapter belongs to the vault, while `forceDeallocate()` is permissionless.
If the market share price is high, a caller can request a dust withdrawal such as `assets = 1`. The vault receives only one atomic token unit, but the rounded share calculation can still burn one adapter LP share unit. That LP share can be worth far more than the dust returned to the vault. The burned value is not paid to the vault or the force-deallocation caller. It remains in the Kairos pool, raising the value of the remaining LP shares.
This creates an incentive for a normal user who also holds LP shares in the same market.

M-09 Dust force-dealloc burns valuable LP shares CONTINUED

They can call `forceDeallocate()` with a dust `assets` value, make the vault-owned adapter burn an economically valuable LP share and receive the benefit through their separate LP position as the remaining pool shares become more valuable.
The existing `forceDeallocate()` penalty does not fully protect the vault because it is based on the caller-provided `assets` amount. For a dust withdrawal, the penalty is also dust-sized even when the adapter's LP-share loss is much larger. The 128-byte withdrawal payload makes this easier because it leaves `maxSharesToRedeem` disabled. The 160-byte payload is not a complete defense either, because an attacker can set `maxSharesToRedeem` to the small rounded share burn that creates the value transfer.
Because of this, a permissionless caller can burn vault-owned LP share value while returning only a dust amount of assets to the vault. The lost value is redistributed to the remaining LPs in the Kairos market. A caller who is also a co-LP can therefore move value from the vault adapter to their own LP position.
The issue is limited to states where the adapter's `_capBinding()` guard is not active, such as after the vest window expires or when the adapter entered a market that was already high priced. The loss per call is bounded by the value of the rounded-up LP share burn, but repeated dust withdrawals can keep burning adapter shares while the vault is compensated only by dust-sized transfers and dust-sized force-deallocation penalties.
Recommendation
Do not let permissionless `forceDeallocate()` realize a large gap between the value of adapter LP shares burned and the `assets` returned to the vault.
For partial `forceDeallocate()` withdrawals, compute the shares that `SwapCore.withdrawCollateral()` would burn at the current share price. Reject the call unless the fair value of those shares is close to the requested `assets` amount, allowing only a small rounding tolerance. A stricter fix is to block partial `forceDeallocate()` withdrawals for `KairosMarketAdapter` and require allocator-controlled deallocation. Do not rely on the caller-provided `maxSharesToRedeem` field for this protection. A caller can choose a cap that still permits the small rounded share burn while preserving the value transfer.

RESOLUTION

Status: Reported during the fix review period.

`https://github.com/kairosswap/protocol/pull/985`

MEDIUM M-10

Base cumulative oracles still use stale adapters

CATEGORY	Configuration
LOCATION	`CumulativeIndexBorrowCompound_compound_v3_usdc_borrow.json`, `CumulativeIndexBorrowMorpho_morpho_blue_usdc_cbbtc_borrow.json`, `CumulativeIndexSupplyMorpho_morpho_blue_usdc_cbbtc_supply.json`
REVIEW	Remediation Review 4

DESCRIPTION

The current Solidity sources for `CumulativeIndexBorrowCompound`, `CumulativeIndexBorrowMorpho` and `CumulativeIndexSupplyMorpho` fail closed when a stale upstream market cannot be projected. On projection-read failure, the current implementations return `(0, false)` instead of continuing from an old borrow index or Morpho share ratio.

The checked-in Base deployment artifacts and the active Base oracle routes still use older adapter runtimes that fall back to stale values during that projection-failure path:

- `CumulativeIndexBorrowCompound_compound_v3_usdc_borrow` returns the previous `baseBorrowIndex` from a `catch` block when `getUtilization()` or `getBorrowRate()` fails.
- `CumulativeIndexBorrowMorpho_morpho_blue_usdc_cbbtc_borrow` falls back to the stale borrow share conversion when `idToMarketParams()` or `borrowRateView()` fails.
- `CumulativeIndexSupplyMorpho_morpho_blue_usdc_cbbtc_supply` falls back to the stale supply share conversion when `idToMarketParams()` or `borrowRateView()` fails.

The affected live Base routes are:

- Compound borrow: `UpgradableTenorRateOracle_compound_v3_usdc_borrow` at `0x848d1e76923ECd272406B52c3624173f6F346c83` points to `IndexBaseRateV1` at `0x3e94b445F735fbd54713446B78AD520797764Aa5`, whose `source()` is `0x70c7B077D5FABb647b37Da0a3C1F9a626b125F44`.
- Morpho borrow: `UpgradableTenorRateOracle_morpho_blue_usdc_cbbtc_borrow -> IndexBaseRateV1_morpho_blue_usdc_cbbtc_borrow -> 0x5decDf756E7afD1DC4cfC2B230db27555638CddF`.
- Morpho supply: `UpgradableTenorRateOracle_morpho_blue_usdc_cbbtc_supply -> IndexBaseRateV1_morpho_blue_usdc_cbbtc_supply -> 0xEca849ed340606512D5D168aa39CF863FC7F6Bd7`.

Consequently, markets depending on those Base base-rate stacks can quote from stale Compound or Morpho cumulative data during exactly the projection-failure condition the current local source rejects.

RECOMMENDATION

Redeploy the affected cumulative-index adapters from the current source and deploy new `IndexBaseRateV1` instances seeded against those fixed sources. Then repoint the corresponding `UpgradableTenorRateOracle` wrappers through the timelocked downstream update flow.

M-10 Base cumulative oracles still use stale adapters CONTINUED

Before activation, verify each live runtime is not a stale-fallback artifact by normalizing immutable constructor arguments and comparing bytecode against the current build. Update or retire stale deployment artifacts so operators do not treat the old adapters as production-safe implementations.

RESOLUTION

Status: Reported during the fix review period.

MEDIUM M-11

Aave index jumps inflate BUY_FLOATING profit

CATEGORY	Unexpected Behavior
LOCATION	`CumulativeIndexSupplyAave.sol, IndexBaseRateV1.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`CumulativeIndexSupplyAave` exposes Aave V3 `getReserveNormalizedIncome(asset)` directly as a cumulative source. `IndexBaseRateV1` then computes the base rate from the live source index divided by the historical TWAR anchor and annualizes that ratio over `twarWindowSec`. A one-time increase in Aave normalized income is therefore treated as a trailing-window rate and can be immediately quoted as the fixed/base leg.

For `BUY_FLOATING`, `SwapCore.buySwap()` reads this base quote before opening the position, sizes the allowed notional from the same quoted rate and then stores both the quoted `baseRate` and the current reference index as `entryFloatingIndex`. If the same Aave supply index jump has already been captured by the reference index at entry, the future floating leg excludes that jump, while the fixed leg keeps paying the annualized quote for the full swap term.

This creates a timing profit vector for a normal user when the external cost of creating a small Aave supply-index jump is lower than the Kairos LP backing that can be claimed. With a 12-hour TWAR window, even a tiny relative source jump is annualized. Because `calculateAvailableLiquidity()` lets notional scale inversely with the same quoted rate, the maximum trade can still lock most of the pool's backing at very low source-jump costs.

A local test modeled an Aave supply-index income jump of roughly `2 * MIN_RATE` annualized over a 12-hour TWAR, then bought a 30-day `BUY_FLOATING` swap immediately after the jump. The entry stored the jumped base quote and the post-jump floating index. With a 177,000,000 USDC source-supply proxy, the modeled source-index cost is `48.493150` USDC, while the buyer received `98,999.999999` USDC above posted swap collateral from a 100,000 USDC LP pool.

This is related to prior `I-14` ("Missing source-market caps for rate oracles"), but the concrete mechanism is narrower and more severe than the carry-cost utilization examples in that report: a discontinuous Aave supply normalized-income jump can be converted into a full-term fixed-leg payment while the floating leg starts after the same jump.

Therefore, if a listed Aave supply-index market has enough Kairos LP backing relative to the cost of moving the source normalized income, a normal user can buy immediately after the source-index jump and extract LP collateral at settlement. The blast radius is bounded by the affected market's locked LP backing, but the trade can be net profitable even when

M-11 Aave index jumps inflate BUY_FLOATING profit CONTINUED

the source-index move is very small.
The attack requires a market where:
- `CumulativeIndexSupplyAave` or an equivalent Aave supply normalized-income source backs both the base-rate TWAR and the cumulative reference path.
- The user can create or coincide with a one-time Aave supply-index income jump, such as a reserve income/premium event, at a cost below the Kairos LP backing available to the trade.
- New swaps are open immediately after the source jump and before governance or keepers pause the market.
Likelihood depends on source-market depth, Aave reserve mechanics, configured TWAR window, swap term, leverage multiplier, fees and Kairos pool size. It is not a privileged-only scenario.
Example:
1. The attacker identifies a `BUY_FLOATING` market whose base oracle is an `IndexBaseRateV1` over the Aave supply cumulative index and whose reference oracle uses the same post-jump cumulative source.
2. The attacker causes or times a one-time increase in Aave normalized income.
3. Immediately afterward, the attacker buys as much `BUY_FLOATING` notional as `SwapCore` allows.
4. `buySwap()` stores the annualized jumped base quote as `swap.baseRate` and stores the already-jumped cumulative index as `entryFloatingIndex`.
5. If the source index is flat or grows only normally afterward, settlement excludes the initial jump from the floating leg but pays the stored fixed leg for the entire term.
6. The buyer receives posted collateral plus LP backing. When LP backing exceeds the external source-jump cost, the sequence is free profit net of gas and capital costs.

RECOMMENDATION

Add source-market risk caps for every external cumulative source used by base-rate or reference-rate markets. The cap should bound maximum Kairos open notional or locked LP backing against conservative source-market manipulation cost for the configured TWAR, term, leverage and fee settings. New swaps should pause or fail closed when the source supply, debt, shares or modeled manipulation cost falls below the required threshold.
For Aave supply-index base-rate markets specifically, also prevent discontinuous normalized-income jumps from being immediately tradable as full-term fixed-leg quotes. Options include capping per-window source-index deltas, adding a cooldown after large source-index jumps before `buySwap()` can consume the quote, using longer/checkpointed windows that cannot be satisfied by a single live-read jump or separating base-rate quote sources from entry reference-index capture so the same jump cannot be priced into one leg and excluded from the other.

M-11 Aave index jumps inflate BUY_FLOATING profit CONTINUED

RESOLUTION

Status: Reported during the fix review period.

MEDIUM **M-12**

Base rate batch quotes can diverge

CATEGORY	Validation
LOCATION	`Utils.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`probeOraclesForMarketCreation()` accepts a base-rate oracle when `getRateByTenor(swapTerm)` succeeds and `getRates([swapTerm])` returns one value. It does not require the two values to match for the same tenor.

This lets a market be created with an oracle that reports one base rate to swap entry and closeout logic, but a different base rate to LP NAV and share-price logic. `Utils.getBaseRateOracle()` reads `getRateByTenor()`, while `Utils.getBaseRateOracleBatch()` reads `getRates()`. Consequently, swaps can be opened with one fixed-rate input while the same positions are marked for LP accounting with another.

For example, an oracle can return 5% from `getRateByTenor(swapTerm)` and 50% from `getRates([swapTerm])`. Market creation accepts it because both calls are individually valid. The opened swap stores the 5% base rate, but `getPoolSharePrice()` uses the 50% batch quote when calculating unrealized PnL. This can skew LP share price, NAV, deposits, withdrawals and settlement accounting until the oracle is replaced or corrected.

RECOMMENDATION

When probing the base-rate oracle at market creation, compare the one-element batch quote with the single-tenor quote after applying the same normalization rules used by the live readers. Revert if the values differ.

The same invariant should be enforced anywhere a tenor oracle is activated or accepted for market use: `getRates([tenor])[0]` must equal `getRateByTenor(tenor)` for every required tenor.

RESOLUTION

Status: Reported during the fix review period.

MEDIUM M-13

Partial forceDeallocate can fully exit LP

CATEGORY	Unexpected Behavior
LOCATION	KairosMarketAdapter.sol
REVIEW	Remediation Review 4

DESCRIPTION

`KairosMarketAdapter.deallocate()` rejects permissionless `forceDeallocate()` only when the decoded `fullAmount` flag is true or the profit-vest cap is binding. In the partial case, it forwards caller-controlled `assets` to `SwapCore.withdrawCollateral()` and only checks that `withdrawn == assets`.
That does not prove the withdrawal was partial. `SwapCore.withdrawCollateral()` treats any `amount >= fullValue` as a full withdrawal and burns all of the LP's shares. In an idle market where the adapter is the sole LP and the cap is not binding, this can return exactly `assets` because the last-LP sweep pays the full pool collateral. The adapter's equality check passes even though the adapter's share balance is zero.
Consequently, any caller of Vault V2 `forceDeallocate()` can force the vault adapter out of a Kairos market by setting `fullAmount=false` and choosing `assets` equal to the adapter's full idle position value. This bypasses the documented rule that full withdrawal is blocked through `forceDeallocate()`. It can grief the vault strategy by moving the allocation back to idle vault assets without allocator intent.

RECOMMENDATION

For `selector == IVaultV2.forceDeallocate.selector`, reject any withdrawal that burns the adapter's whole LP position. Record `sharesBefore` and `sharesAfter`, then revert with `FullAmountNotAllowedViaForceDeallocate()` when `sharesBefore > 0 && sharesAfter == 0`.
If permissionless partial withdrawals should remain available, also require a meaningful `maxSharesToRedeem` limit and reject requests whose requested `assets` amount is greater than or equal to the adapter's current realizable LP value.

RESOLUTION

Status: Reported during the fix review period.

https://github.com/kairosswap/protocol/pull/985

MEDIUM M-14

Factory misses non-exit adapter guard

CATEGORY	Validation
LOCATION	`KairosBuyerAdapterFactory.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`KairosBuyerAdapterFactory.createAdapter()` validates that the market exists and that its `swapToken` matches the vault asset. It does not reject markets where `earlyExitAllowed` is false.
`KairosBuyerAdapter` already documents why that configuration is unsafe for redeemable vault accounting. For live buyer positions on non-early-exit markets, `realAssets()` is only an accrued fixed-versus-floating mark. It ignores the locked remaining term because the position cannot be closed before maturity. The comment also warns that a redeemable vault using idle cash plus `realAssets()` can let early redeemers shift value to or from depositors who remain until expiry.
That warning is useful, but it is not a code-level control. A vault authority can still use the official factory to deploy a buyer adapter for a non-early-exit market and then add it to a redeemable vault. Consequently, vault share pricing can consume the accrued-only adapter value even though the code already recognizes that instant redemption is unsafe for this exposure.

RECOMMENDATION

Enforce the documented restriction in `KairosBuyerAdapterFactory.createAdapter()`. After loading the market, reject non-early-exit markets before deploying the adapter.

```plain text
if (!market.earlyExitAllowed) revert NonEarlyExitMarketUnsupported();
```

If non-early-exit buyer exposure must be supported, expose it through a separate hold-to-maturity adapter or vault mode that prevents instant redemption while those swaps are live.

RESOLUTION

Status: Reported during the fix review period.

MEDIUM M-15

Pre-poke reference indexes skew payouts

CATEGORY	Unexpected Behavior
LOCATION	SwapCore.sol
REVIEW	Remediation Review 4

DESCRIPTION

`liquidateSwap()` refreshes the market reference index by calling `rateIndex.update(market.referenceRateOracle)` directly. `withdrawCollateral()` does the same at `contracts/SwapCore.sol:374-376`. Neither function first calls `Utils.pokeMarketOracles(...)`.

This is inconsistent with `updateMarketRateIndex()`, which pokes the market's reference, base and risk oracles before it snapshots the reference index. `Types.CreateMarketParams` and `SwapCore._initializeMarket()` document that a market's `referenceRateOracle` must be poke-invariant, meaning its read must not change across `poke()`. That precondition is only caller-enforced. `validateMarketParams()` and `probeOraclesForMarketCreation()` can still accept an `IPokeable` reference oracle whose valid output changes only after `poke()` is called.

Consequently, a market that violates the documented reference-oracle precondition can execute value-bearing state changes against stale pre-poke reference data even though fresh data is available through the protocol's own refresh function. In `liquidateSwap()`, stale data can make a swap liquidatable even though it would be healthy after the reference oracle is poked, causing buyer collateral or pool backing to move under the wrong index. In `withdrawCollateral()`, an LP can redeem shares at a stale share price and leave the later post-poke loss with remaining LPs.

RECOMMENDATION

Enforce the documented reference-oracle precondition at market creation. Reject `referenceRateOracle` addresses that advertise `IPokeable` unless they are explicitly allowlisted as read-invariant across `poke()`. This closes `liquidateSwap()`, `withdrawCollateral()` and any future value-bearing function that directly refreshes the reference index.

As defense in depth, make every value-bearing direct reference-index refresh consume the same committed oracle state as `updateMarketRateIndex()`. Before `rateIndex.update(market.referenceRateOracle)` in `liquidateSwap()` and `withdrawCollateral()`, call:

```solidity
Utils.pokeMarketOracles(
rateIndex,
```

M-15 Pre-poke reference indexes skew payouts CONTINUED

```
market.referenceRateOracle,
market.baseSwapRateOracle,
market.riskPremiumOracle,
admin.pokeMinSpacingSec()
);
```

Prefer a shared internal helper so future value-bearing index refreshes cannot accidentally bypass the poke step.

RESOLUTION

Status: Reported during the fix review period.

MEDIUM M-16

Negative tenor quote inflates LP NAV

CATEGORY	Validation
LOCATION	`Utils.sol`
REVIEW	Remediation Review 4

DESCRIPTION

This issue requires a specific oracle configuration: the market's tenor base-rate oracle must be able to return a negative shorter-tenor quote for a cumulative `BUY_FIXED` market while the full-term quote remains non-negative. If deployments enforce non-negative rates for every tenor off-chain, this issue is not reachable in that deployment. The on-chain code only enforces that invariant for the full-term quote read during `buySwap()`.

`SwapCore.buySwap()` recognizes that a cumulative reference index is monotone and rejects a negative `BUY_FIXED` base rate at entry with `E512`. That check only covers the full `market.swapTerm` quote. Active LP mark-to-market later collects each bucket's remaining tenor in `calculateTotalUnrealizedPnL()`, reads that shorter tenor through `getBaseRateOracleBatch()` and passes the signed result into `calculateBucketUnrealizedPnL()`. That function prices the remaining floating leg with `projectedRate` without rejecting negative rates for cumulative `BUY_FIXED` markets.

Consequently, a curve that is positive at the full swap term but negative at a shorter remaining tenor can pass swap entry while later marking active LP buckets as if the monotone floating index could accrue negative future growth. A mature LP can withdraw at the inflated NAV before settlement realizes the actual cumulative-index outcome. The remaining LPs absorb the shortfall.

We developed a test that used a 1-year cumulative `BUY_FIXED` market with a valid positive 5% full-term base quote and a negative -50% 6-month remaining-tenor quote. After half the term, the attacker LP withdrew `1,026,315.789474` USDC from a `1,000,000` USDC deposit. After settlement, the remaining LP's value fell to `923,684.210526` USDC.

RECOMMENDATION

M-16 Negative tenor quote inflates LP NAV CONTINUED

Apply the same cumulative `BUY_FIXED` non-negative-rate invariant to every remaining-tenor base-rate consumer, not only to `buySwap()`. Reject negative `projectedRate` values in LP mark-to-market when `market.rateType == BUY_FIXED` and the reference convention is `Cumulative`. Also reject negative remaining-tenor rates before early-exit settlement.
As a defense-in-depth layer, tenor oracles and wrappers used by cumulative `BUY_FIXED` markets should reject negative rates for every supported tenor, not only for the default or full-term tenor.

RESOLUTION

Status: Reported during the fix review period.

`https://github.com/kairosswap/protocol/pull/989`

MEDIUM M-17

RawIndex underprices BUY_FIXED LP backing

CATEGORY	Configuration
LOCATION	IndexBaseRateV1.sol:
REVIEW	Remediation Review 4

DESCRIPTION

`RawIndexBaseRateV1` is explicitly documented as a BUY_FLOATING-only base oracle. It returns the raw trailing TWAR rate `r` for every tenor instead of the full-tenor compounded-equivalent quote. That lower-bound quote is safe when the BUY_FLOATING buyer receives the fixed leg, because any extra cumulative-index growth accrues to the pool. The same quote is unsafe for BUY_FIXED. In BUY_FIXED, the buyer pays the fixed leg and receives the cumulative floating leg. If the market uses `RawIndexBaseRateV1` as the fixed-side base oracle while settling against the same compounding cumulative source, `buySwap()` stores only the raw simple rate as `swapRate`. Settlement later derives the floating payment from the higher cumulative-index growth.

The code relies on comments for this side-specific precondition. `SwapCore.createMarket()` accepts an arbitrary `fixedBaseSwapRateOracle` and stores it in the BUY_FIXED market, while `Utils.probeOraclesForMarketCreation()` only checks that `getRateByTenor()` and `getRates()` return valid values. It never checks whether the oracle is safe for the side it is being assigned to. LP backing in an affected BUY_FIXED market can be transferred to buyers at normal expiry. A local test used `CumulativeIndexBorrowAave` adapter as the reference oracle and a `RawIndexBaseRateV1` sourced from the same Aave borrow index. With a 180-day term, 18x leverage and a 104.5% APR Aave-borrow stress rate, the buyer locked a raw fixed rate of about 104.5% APR, but the Aave-style borrow index realized a materially larger cumulative floating leg. Settlement paid the buyer their posted margin plus the full `9,999,999.971370` USDC of locked LP backing.

The issue requires a market to use `RawIndexBaseRateV1` in the fixed-side base-oracle slot. That is an unsafe configuration, but it is possible because market creation validates only interface shape and successful reads, not whether the oracle's quote semantics match the market side. If a market creator, integration, listing process or wrapper retargeting puts the first-party RawIndex oracle into a BUY_FIXED market and normal LPs supply that market, positive compounding sources can create a material gap that grows with `rate * term`, while LP backing is sized from only the raw fixed rate divided by leverage.

This is the inverse of the existing bumped-quote BUY_FLOATING basis issue ([H-02](/p/3968bda5828c812fb48ee3b51e2e773d?v=6fd3feaed4d64627a1aaf48aaa9e085f&pvs=25)). That issue covers `IndexBaseRateV1` overpaying the fixed leg to BUY_FLOATING buyers. This issue covers

M-17 RawIndex underprices BUY_FIXED LP backing CONTINUED

`RawIndexBaseRateV1` undercharging BUY_FIXED buyers while the floating leg still compounds.

RECOMMENDATION

Enforce the side-specific oracle precondition on-chain. The minimal fix is to add an interface or immutable metadata flag for tenor-oracle quote semantics and reject `RawIndexBaseRateV1`-style unbumped quotes in the BUY_FIXED base-oracle slot during market creation and wrapper retargeting.
More generally, cumulative markets should distinguish at least:
- pool-safe upper-bound or compounded-equivalent tenor quotes allowed for BUY_FIXED;
- lower-bound raw/simple tenor quotes allowed only for BUY_FLOATING.

RESOLUTION

Status: Reported during the fix review period.

https://github.com/kairosswap/protocol/pull/989

LOW **L-01** RESOLVED

LP withdrawals lack max-share slippage

CATEGORY	Warning
LOCATION	`SwapCore.sol`
REVIEW	Remediation Review 4

DESCRIPTION

Partial amount-based LP withdrawals do not let the caller cap the number of LP shares burned. When `amount` is below the LP's full position value, `withdrawCollateral` computes `sharesToRedeem = ceil(amount * WAD / sharePrice)` and then sets `withdrawalAmount = amount`. The only slippage check is `minCollateralOut`, which compares against `withdrawalAmount`. In this mode, `minCollateralOut <= amount` always passes, regardless of how far the execution-time share price moved or how many shares the call now needs to burn.

The API has no `maxSharesIn` or `maxSharesToRedeem` parameter. If the share price moves down between quote and execution, the LP still receives the requested token amount, but more shares are consumed than the LP or integrator may have intended. `KairosMarketAdapter.deallocate` exposes the same amount-mode behavior for Morpho vault deallocations: it decodes only `minCollateralOut` and `fullAmount`, forwards the request to `swapCore.withdrawCollateral` and checks only that the token amount withdrawn equals the requested `assets`.

RECOMMENDATION

For amount-based withdrawals, add a `maxSharesIn` or `maxSharesToRedeem` parameter and revert when the computed `sharesToRedeem` exceeds it. Adapters should decode and forward this cap so parent vaults can protect share inventory during deallocation.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/855`

LOW L-02 RESOLVED

Short-tenor liquidity floors to zero

CATEGORY	Rounding
LOCATION	SwapFormulas.sol, SwapCore.sol, ImpliedBaseRate.sol
REVIEW	Remediation Review 4

DESCRIPTION

`SwapFormulas.calculateAvailableLiquidity()` first computes:

```plain text
Math.mulDiv(poolCollateral, leverageMultiplier, effectiveRate(baseRate))
```

and only then scales that intermediate result by SECONDS_IN_YEAR / term. If poolCollateral * leverageMultiplier is smaller than effectiveRate(baseRate), the first division floors to zero before the short-tenor multiplier can restore the value. A single full-precision calculation over the combined numerator and denominator preserves that capacity:

```plain text
Math.mulDiv(poolCollateral, leverageMultiplier * SECONDS_IN_YEAR, effectiveRate(baseRate) * term)
```

The same floored value is used by SwapCore.buySwap() for the E508 liquidity gate and by ImpliedBaseRate.getBaseRate()` when it compares paired-market liquidity.

RECOMMENDATION

Replace the two-step calculation with a single full-precision `mulDiv` over the combined numerator and denominator, for example:

```plain text
Math.mulDiv(poolCollateral, leverageMultiplier * SECONDS_IN_YEAR, effectiveRate(baseRate) * term)
```

If the combined numerator can overflow before mulDiv receives it, split the expression into a safe 512-bit decomposition rather than multiplying in unchecked uint256` space.

RESOLUTION

Status: Resolved.

L-02 Short-tenor liquidity floors to zero CONTINUED

https://github.com/kairosswap/protocol/pull/851

LOW L-03

ACKNOWLEDGED

Dust collateral blocks full adapter exits

CATEGORY	DoS
LOCATION	`KairosMarketAdapter.sol`, `SwapCore.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`KairosMarketAdapter.deallocate()` treats `fullAmount=true` as the adapter's full LP withdrawal mode. In the current implementation, that mode requires `assets == 0` and passes `type(uint256).max` to `SwapCore.withdrawCollateral()`, which triggers a full redemption of the adapter's LP position. `SwapCore.withdrawCollateral()` computes the full-position value from the LP share balance and the current share price. If active swaps exist, it keeps one share outstanding, but the resulting withdrawal amount is still almost the full LP position. The function then rejects the withdrawal when that amount is greater than `totalLPAvailableCollateral()`, which is `pool.totalCollateral - pool.lockedCollateral`.

This preserves `SwapCore`'s locked-collateral invariant, but the adapter's full-exit mode has no way to withdraw the currently available collateral while keeping the residual locked shares and claim. As a result, any active swap with nonzero LP backing, even a dust-sized one, blocks the adapter's full-market exit until the swap settles or is otherwise closed.

RECOMMENDATION

Change the adapter's full-exit behavior to be lock-aware. It should withdraw the maximum currently available collateral and keep tracking the residual locked shares or claim or expose a separate `withdrawAvailableAndKeepResidual` path. Do not route operational full exits exclusively through an all-share redemption that reverts whenever `lockedCollateral` is nonzero.

RESOLUTION

Status: Acknowledged.

LOW L-04 RESOLVED

PythAdapter rejects valid negative rates

CATEGORY	Oracle
LOCATION	PythAdapter.sol
REVIEW	Remediation Review 4

DESCRIPTION

KairoSwap's shared rate consumers are built around signed rates. `IRateOracle.getRate()` returns an `int256`, `Utils.getOracleRate` documents that reference rates can be negative, `RateIndexLib` accumulates signed spot-rate time and `SwapFormulas` handles negative rates when deriving and collateralizing swap exposure. Pyth also represents `PythStructs.Price.price` as a signed integer. `PythAdapter.getRate()` rejects that signed domain by returning `(0, false)` whenever `pythPrice.price <= 0` and then rejecting again if the scaled value is non-positive. A Pyth rate feed that legitimately becomes negative is therefore treated as invalid instead of being returned as a signed rate.

RECOMMENDATION

Permit negative Pyth prices for rate feeds and scale them while preserving sign. Only reject genuinely invalid timestamp/staleness/overflow conditions or return (0,false) for invalid data so callers can handle invalidity consistently rather than reverting in the adapter.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/852

LOW L-05

RESOLVED

Settled swap ID reverts settlement batch

CATEGORY	Warning
LOCATION	SwapCore.sol
REVIEW	Remediation Review 4

DESCRIPTION

`SwapCore.makePayment` is permissionless and accepts an arbitrary array of swap IDs. It forwards the array to `_settleSwaps`, which processes entries sequentially. For each ID, `_settleSwaps` loads the swap and reverts with `Errors.E501()` when `swap.settled` is already true instead of skipping the entry or returning a per-item failure status.

This lets any third party poison a keeper or integration settlement batch. The third party can settle one expired swap ID that is included in the pending batch. When the keeper's batch later reaches that pre-settled ID, the whole transaction reverts. If the poisoned ID is near the end of the array, the keeper also spends gas on the earlier settlement work before the final revert rolls those state changes back.

RECOMMENDATION

Make batch settlement idempotent for already-settled swap IDs. For example, skip settled entries or return a per-item status so one pre-settled ID cannot revert unrelated settlements in the same batch.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/853

LOW L-06

ACKNOWLEDGED

LP adapter bypasses market whitelist

CATEGORY	Validation
LOCATION	`KairosMarketAdapter.sol, SwapCore.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`KairosMarketAdapter.allocate()` receives the original vault caller in its `sender` parameter but does not use it. The adapter instead supplies collateral with `SwapCore.supplyCollateral(marketId, assets, address(this), 0)`, so `SwapCore` evaluates the per-market LP whitelist against the adapter address, not against the vault user or allocator that initiated the allocation.

This means that once a market owner whitelists the `KairosMarketAdapter`, the whitelist admits the adapter's parent vault as the economic LP wrapper. An address that is not permitted to supply LP collateral directly can still gain exposure to the whitelisted market through the vault's allocation and share ownership model, because the only LP address recorded and checked by `SwapCore` is the adapter.

RECOMMENDATION

If whitelisted markets require end-user permissioning, enforce the original `sender` against a market- or vault-level allowlist before the adapter supplies collateral. Vault shares should also be restricted or non-transferable for those markets if share ownership is meant to represent approved LP exposure.

If adapter-level permissioning is intended, document this explicitly for market owners: whitelisting a Kairos LP adapter admits all current and future economic participants of the adapter's parent vault, not only the adapter deployer or allocator.

RESOLUTION

Status: Acknowledged.

`https://github.com/kairosswap/protocol/pull/854`

LOW **L-07** RESOLVED

Idle LP value query needs live oracle

CATEGORY	Warning
LOCATION	`SwapCore.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`getLpValue()` is documented as the value an LP would receive on an immediate full withdrawal, but it always calls `getPoolSharePrice()` before returning that value. `getPoolSharePrice()` reads the market reference oracle through `Utils.getFreshIndex()` whenever the pool has shares, so `getLpValue()` can revert if the oracle is unavailable.

`withdrawCollateral()` does not require the oracle in the same idle-pool case. When `pool.lockedCollateral == 0`, it computes the withdrawal price directly from `pool.totalCollateral / pool.totalShares` and bypasses the reference oracle entirely.

Therefore, an LP can be able to withdraw from an idle pool while `getLpValue()` reverts. Integrations that rely on `getLpValue()` to display, size or account for the LP's realizable position can become unavailable during an oracle outage even though the collateral is unlocked and withdrawable.

RECOMMENDATION

Move effective LP withdrawal pricing into a shared internal helper used by both `getLpValue()` and `withdrawCollateral()`. In the idle-pool case, the helper should compute the share price from stored pool totals without calling the reference oracle, then apply the same vesting cap for both functions.

If the `MIN_SHARE_PRICE` redemption floor is removed or changed separately, the shared helper should use that corrected idle-pool pricing rule as well.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/882/commits/ff8145d00bfe7f2d6e8a0558c375b3bf7d4eea72`

LOW L-08

ACKNOWLEDGED

First market can lock spot oracle convention

CATEGORY	DoS
LOCATION	RateIndexLib.sol
REVIEW	Remediation Review 4

DESCRIPTION

`RateIndexLib.initialize()` stores one `RateConvention` per oracle address the first time any market uses that oracle. If the oracle has already been initialized, any later market that asks for a different convention reverts with `E607`.

`SwapCore.createMarket()` is permissionless and forwards the creator-supplied `rateConvention` into this initialization. This is safe for `Cumulative` oracles because they must implement `getCumulativeIndex()`. It is not safe for the two spot conventions, because both `SpotRate` and `SpotCompoundRate` validate through the same `IRateOracle.getRate()` function and start from the same `INITIAL_INDEX`.

Therefore, the first market creator can lock an unused spot oracle address to the wrong spot convention. For example, if an oracle is meant to be used as `SpotCompoundRate`, another creator can initialize it first as `SpotRate`. The intended market then reverts and every future market that tries to use the same oracle address with `SpotCompoundRate` also reverts.

Consequently, this creates a low-cost market-creation griefing issue for newly deployed spot oracle addresses. The attacker does not need to corrupt oracle data or interact with existing funds. Recovery requires deploying a replacement oracle or wrapper address, then using that new address for the intended markets.

RECOMMENDATION

Do not let the first market creator define the convention for a shared oracle address.

Make the convention intrinsic to the oracle through immutable metadata and verify it during `initialize()`. Alternatively, key the rate-index state by `(oracle, convention)` so `SpotRate` and `SpotCompoundRate` cannot block each other. If neither design is acceptable, restrict first-time oracle initialization to an authorized registry or market-creation role that has already approved the oracle and its convention.

RESOLUTION

Status: Acknowledged.

LOW L-09 RESOLVED

Long-tenor base-rate compounding can revert

CATEGORY	Unexpected Behavior
LOCATION	IndexBaseRateV1.sol
REVIEW	Remediation Review 4

DESCRIPTION

`IndexBaseRateV1.getRateByTenor()` and `IndexBaseRateV1.getRates()` first compute a valid realized TWAR rate, then pass that rate and the requested tenor into `_applyTenorBump()`. For positive rates, `_applyTenorBump()` computes `rWad * tenorSeconds / SECONDS_IN_YEAR` and passes the result directly to `FixedPointMathLib.expWad()`.

There is no explicit bound on the rate-tenor product before the exponential call. A realized rate can be valid for ordinary tenors but too large to exponentiate for a very long tenor. For example, a 200% realized APR combined with a 100-year tenor exceeds the supported `expWad()` input range even though both the rate and the tenor are otherwise representable.

Consequently, a base-rate read that should be reported as invalid can hard-revert instead. Markets that depend on the oracle for that tenor can then reject market creation, new swaps, early exits and LP valuation-dependent supply or withdrawal operations until either the rate decreases or the queried remaining tenor becomes small enough.

RECOMMENDATION

Validate the complete rate-tenor computation before calling `expWad()`. If `rWad * tenorSeconds / SECONDS_IN_YEAR` is outside the supported exponential range, return `(0, false)` from `getRateByTenor()` and `(rates, false)` from `getRates()` instead of reverting.

Also guard the multiplication after `expWad()`, since an exponent just inside the input range can still produce an output too large for the final arithmetic. If long-tenor compounding is intentionally unsupported above a threshold, enforce that threshold during market creation or document it as an oracle-level bound.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/882/commits/ad46caf18f1ce21420f281f1b1e6d03c95265308

LOW L-10 RESOLVED

Pool-zero event misses exact drains

CATEGORY	Events
LOCATION	`Utils.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`PoolCollateralZeroed` is only emitted after settlement when `_applyPoolValueChanges()` returns `true`. That helper returns `true` only when the requested decrease is greater than `pool.totalCollateral`. If the decrease exactly equals the remaining pool collateral, the helper subtracts the amount, leaves `pool.totalCollateral` at zero and returns `false`.

This exact drain is a normal reachable case because settlement losses are capped by the swap's `poolCollateralBacking`. A swap can consume the final remaining pool backing without the helper reporting that the pool moved from positive collateral to zero.

Pool-side liquidation has the same reporting gap. `Utils.liquidateSwap()` calls `_applyPoolValueChanges(pool, 0, poolBacking)` when the pool is liquidated, but the return value is ignored. `SwapCore.liquidateSwap()` also has no independent check that the pool moved to zero before emitting close events.

Consequently, event-driven monitors can miss the moment a market becomes a zero-collateral pool while LP shares may still exist. Indexers, routing systems and automated LP controls that rely on `PoolCollateralZeroed` can continue treating the pool as not-yet-wiped unless they recompute pool state from storage or every balance-changing event.

RECOMMENDATION

Make `_applyPoolValueChanges()` report positive-to-zero transitions, not only strict underflow clamps. Capture the pool collateral before applying the change and return `oldCollateral > 0 && pool.totalCollateral == 0`.

Propagate that result through both settlement and liquidation. `SwapCore` should emit `PoolCollateralZeroed` exactly once whenever a market's pool collateral moves from a positive value to zero, including exact drains and pool-side liquidations.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/882/commits/4209d1e3da7ae0724dd8617b585c36567582309c`

LOW **L-11** RESOLVED

SwapClosed reports accrued unsigned legs

CATEGORY	Events
LOCATION	`Utils.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`SwapClosed` exposes `floatingPayment` and `fixedPayment` as `uint256` fields. `Utils.settleSwap()` fills those fields by taking the absolute value of the signed fixed and floating payments. This removes the sign of each leg, so the event cannot describe settlements where supported negative rates make one leg negative. For early exits, the event fields also do not describe the closeout that determined the transfer. `calculateEarlyExitSettlement()` computes the real obligation from accrued payments, projected remaining payments, the cumulative cross-term where needed and the early-exit fee. After that, `settleSwap()` recomputes only the elapsed accrued fixed and floating payments for event reporting. The remaining-term value that can drive `amountSettled` is not included in `fixedPayment` or `floatingPayment`.

Consequently, event-only accounting cannot reliably reconstruct realized PnL or reconcile the reported legs to `amountSettled`. Integrations that subtract the two unsigned magnitudes can get the wrong sign when one leg is negative and integrations that use the fields for early exits can miss the projected closeout component entirely.

Liquidation events are even less informative: `SwapCore.liquidateSwap()` emits the unified `SwapClosed` event with both leg fields set to zero even though the liquidation decision used an accrued net obligation.

RECOMMENDATION

Version the close event instead of reusing unsigned accrued-leg fields for multiple closure types.

Emit signed fixed and floating components for normal settlement. For early exits, also emit the signed projected remaining components and the retained fee or emit a signed pre-fee closeout amount plus the retained fee. For liquidations, emit the signed accrued obligation that made the position liquidatable. At minimum, rename or deprecate the current leg fields so consumers do not treat them as a complete realized closeout decomposition.

L-11 SwapClosed reports accrued unsigned legs CONTINUED

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/891

LOW **L-12** RESOLVED

Sweep leaves stale tracked swap

CATEGORY	Unexpected Behavior
LOCATION	`KairosBuyerAdapter.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`ACTION_SWEEP` transfers the buyer adapter's entire token balance to the parent vault. It then falls through to the common allocation update, but it does not inspect `_swapIds` or remove any settled swap whose payout was already paid into the adapter.

This matters when a tracked buyer swap is settled outside `ACTION_MAKE_PAYMENT`, so the payout is sitting in the adapter's token balance. A later sweep can move that payout to the vault and reduce the adapter's reported allocation to zero. The settled swap can still remain in `_swapIds` and `totalTrackedSwaps` still counts it.

For example, a vault curator configures a buyer adapter with a small `maxTrackedSwaps` limit and uses every slot for a rolling rate strategy. One swap reaches maturity and is settled directly through `SwapCore`, so the payout lands on the adapter as idle token balance while the swap id remains tracked. The allocator sees the idle balance and calls `ACTION_SWEEP` to recover it to the vault. The sweep succeeds, but the adapter reports zero remaining allocation and does not remove the settled id. The next attempt to roll into a replacement swap then fails with `SwapCapExceeded`, while ordinary settled-swap cleanup can also be blocked because the vault now records zero allocation for the adapter id.

The direct asset impact is limited because the swept payout reaches the parent vault. Existing vault depositors are not deprived of that cash. The practical impact is instead a persistent strategy outage: the vault can no longer roll the adapter into a replacement buyer swap while the stale id consumes the final tracking slot. Depositors remain exposed to a vault whose advertised Kairos buyer allocation may be inactive and the vault can miss intended rate exposure, hedge protection or yield until an operator restores allocation or replaces the adapter.

Consequently, an adapter that was already at `maxTrackedSwaps` can be left unable to buy another swap because `allocate()` rejects new buys when `totalTrackedSwaps >= maxTrackedSwaps`. Ordinary `ACTION_REMOVE_SETTLED` cleanup can also become unavailable after the sweep, since the adapter documents that `VaultV2` rejects deallocation calls when the recorded allocation is zero. The stale slot can therefore persist even though the underlying swap is already settled and the cash has been recovered. Operators may need governance or curator intervention to restore nonzero allocation, deploy a replacement adapter or otherwise repair the stale tracking state.

L-12 Sweep leaves stale tracked swap CONTINUED

RECOMMENDATION

Do not let `ACTION_SWEEP` create a zero-allocation state while fully settled swaps remain tracked. Reconcile settled buyer swaps before or during the sweep and call `_removeSwapId()` for any tracked swap that is settled, has no pending escrow and no longer has a settlement payout to account for.
Alternatively, make `ACTION_SWEEP` reject while settled buyer swaps are still tracked and require the operator to use the settlement-specific actions first. This keeps token recovery and swap tracking consistent and prevents a sweep from consuming the adapter's future swap capacity.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/883`

LOW L-13

RESOLVED

Delayed relays bypass max-rate-change breaker

CATEGORY	Oracle
LOCATION	`BaseTimelockedOracle.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`BaseTimelockedOracle._validateRateVector()` decides whether to enforce `maxRateChangeWad` from relay time, not from the signer-attested observation time. The previous rate timestamp is now the stored `observedAt`, but the freshness check still uses `block.timestamp - prevUpdatedAt <= maxAgeSec`.

This lets a relayer control whether the rate-change breaker applies. If a signed update would exceed `maxRateChangeWad` when posted immediately, the relayer can wait until the previous observation is older than `maxAgeSec` while the new signed observation is still fresh. The same update then passes because `_validateRateVector()` treats the previous value as stale and skips the delta check.

Consequently, a valid signed rate jump larger than the configured circuit-breaker limit can be installed without signer compromise. This can abruptly reprice fixed-rate forecasts or sided risk premiums. Markets that consume those oracle values can misprice newly opened swaps, early exits, LP mark-to-market values and risk-premium calculations until the next accepted update corrects the curve.

For example, assume a 5% rate was stored with observation time `T0`. Signers later attest a 20% rate at `T1`, where the 15% jump exceeds `maxRateChangeWad`. Posting that update at `T1` reverts with `RateChangeTooLarge`. If the relayer waits until `block.timestamp > T0 + maxAgeSec` but still within `T1 + maxAgeSec` and before the signed deadline, the same signed update is accepted.

RECOMMENDATION

Validate rate-change cadence against observation time, not relay time. Pass the incoming `observedAt` into `_validateRateVector()` and enforce `maxRateChangeWad` when `observedAt - prevUpdatedAt <= maxAgeSec`.

Also require observation timestamps to move forward for each stored tenor. The delta check should only be skipped after a genuine observation gap larger than `maxAgeSec` or after an explicit governance reset that intentionally clears the prior rate state.

L-13 Delayed relays bypass max-rate-change breaker CONTINUED

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/882/commits/aa7c5b0a7c27c85e2fa177929c23b54cabc16b20

LOW **L-14** RESOLVED

Duplicate tenor probe accepts bad batches

CATEGORY	Validation
LOCATION	`UpgradableTenorRateOracle.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`UpgradableTenorRateOracle._validateDownstreamHealth()` validates a replacement downstream by reading `getRateByTenor(defaultTenorSeconds)` and then calling `getRates()` with the same default tenor twice. The check only requires `isValid = true` and a response array with the same length as the probe. That proves the downstream can answer one repeated tenor. It does not prove that `getRates()` returns each rate at the same position as the requested tenor when the input contains distinct tenors. It also does not compare the batch result against the corresponding `getRateByTenor()` value. A downstream can therefore pass activation while returning correct single-tenor reads and wrong batch results for the heterogeneous tenor arrays used by LP mark-to-market accounting.

Consequently, active LP share pricing can use rates attached to the wrong bucket. If the resulting share price is understated, new LPs can deposit too cheaply and dilute existing LPs. If it is overstated, withdrawing LPs can exit against inflated NAV before the remaining LPs absorb the correction.

RECOMMENDATION

Probe at least two distinct tenors during downstream activation. For every probed tenor, require the batch value at that index to match the corresponding `getRateByTenor()` result, in addition to checking `isValid` and array length.

If one wrapper can serve markets with several terms, keep an explicit required-tenor set or required tenor range and validate those terms before accepting a new downstream.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/882/commits/23e1f88c1126b4c952644e371170dd6bc11fd8a6`

LOW L-15

RESOLVED

Default tenor probe misses live tenors

CATEGORY	Validation
LOCATION	`UpgradableTenorRateOracle.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`UpgradableTenorRateOracle._validateDownstreamHealth()` only checks the configured `defaultTenorSeconds`. The batch probe repeats that same tenor, so a candidate downstream can pass activation even if it does not support other market terms or remaining-tenor values already used through the wrapper.

This can happen with a non-malicious oracle migration. For example, a downstream may truthfully support the wrapper's 30-day default tenor while returning invalid for 90-day or 180-day markets. `BaseUpgradableOracleWrapper.activateDownstream()` still accepts it because the health check never asks for those live tenors.

Consequently, existing markets that rely on unsupported tenors can lose oracle liveness after a successful downstream activation. New swaps, early exits, LP share pricing and adapter valuation can revert or return invalid until another downstream is queued and the three-day timelock elapses.

RECOMMENDATION

Do not use `defaultTenorSeconds` as the only compatibility check for an upgradeable tenor oracle. Bind each wrapper to the market terms it is expected to serve, then validate every required tenor before activation.

At minimum, include a heterogeneous batch probe and compare each batch result with the corresponding single-tenor read. If the candidate returns invalid or reverts for any required tenor, reject the downstream change.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/882/commits/d6bc210b760ca6fd8b78ad04874e5abd047b8ab7`

LOW **L-16** RESOLVED

Tenor removal reuses old premium anchors

CATEGORY	Oracle
LOCATION	`BaseTimelockedOracle.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`activateTenorBucketsChange()` replaces the active `tenorBuckets` array and resets `lastPostTimestamp`, but it does not clear stored rates for tenor keys that remain in the new bucket set. Those retained rates were signed for the old tenor configuration. After an intermediate bucket is removed, `_getRatesInternalWith()` can still load the surviving anchors, interpolate across the new bucket layout and return `isValid = true`. Consequently, a bucket removal can immediately change the premium returned for a wrapper's immutable tenor even though signers never attested to that new curve. For example, a signed `[30d, 60d, 90d]` curve with premiums `[1%, 10%, 1%]` can become a valid `[30d, 90d]` curve after activation. A 60-day wrapper then interpolates `1%` from the retained endpoints instead of becoming invalid until signers post a curve for the new bucket set. New swaps opened during that window snapshot the unsigned premium into their pricing.

RECOMMENDATION

Clear stored rates whenever a tenor bucket change is activated. After assigning the new `tenorBuckets`, call `_resetStoredRates()` and keep `lastPostTimestamp = 0` so signers can immediately post a fresh curve for the new bucket list. Reads should remain invalid until that post succeeds.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/882/commits/5e565e32b6bfdae9c31e9eba36f592142d01db5b`

LOW L-17

RESOLVED

No-op wrapper poke carries cooldown

CATEGORY	Oracle
LOCATION	`OraclePokeLib.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`OraclePokeLib.poke()` records `lastPokes[oracle] = block.timestamp` before it calls `oracle.poke()`. It only rolls that timestamp back when the low-level call fails. This interacts badly with `BaseUpgradableOracleWrapper.poke()`, which returns successfully without touching any downstream oracle when the active downstream does not advertise `IPokeable`.

As a result, `SwapCore.updateMarketRateIndex()` can record a successful poke against the stable wrapper address even though the wrapper did nothing. If governance then activates a pokeable downstream before the protocol-level poke spacing expires, automatic pokes through `updateMarketRateIndex()` are skipped until the old wrapper-level cooldown has elapsed.

Consequently, a newly activated downstream can become stale or invalid before its first eligible automatic refresh. Rate-dependent operations can temporarily revert or use a near-stale quote. The issue is bounded because a direct call to the wrapper's `poke()` still reaches the new downstream, but permissionless automatic refreshes can be suppressed during the inherited cooldown window.

RECOMMENDATION

Do not let a successful wrapper no-op consume the protocol-level poke cooldown for a future downstream. One option is to make wrapper `poke()` expose whether it actually forwarded to the downstream and only update `lastPokes` after a real downstream poke.

Alternatively, key poke cooldown state by a wrapper downstream generation rather than only by the stable wrapper address. When activating a downstream that advertises `IPokeable`, consider poking it before the final health check and validating the post-poke read.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/882/commits/76177ad5b06573ddf75ce9e6ed728b5c665d9699`

LOW **L-18** RESOLVED

Negative premiums brick swap entry

CATEGORY	Oracle
LOCATION	`RiskRateOracle.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`postRiskPremiums()` validates upper and lower premium vectors only against the signed rate bounds inherited from `BaseTimelockedOracle`. If those bounds include negative values, authorized signers can post a negative risk premium and the side-specific wrapper can report it as `(rate, true)`. Wrapper health checks and market creation only check that the oracle value is valid.

`Utils.getRiskPremiumOracleRate()` later applies a stricter consumer rule and reverts with `E612` when the returned rate is negative. Consequently, a market can be created or a wrapper upgrade can be activated while the current premium is already unusable by swap entry. Every new swap on the affected side then reverts until a nonnegative premium is posted.

RECOMMENDATION

Use one sign policy across the risk premium producer, wrappers and `SwapCore` consumer. If risk premiums are meant to be unsigned, require nonnegative minimum bounds and reject negative `upRates` and `loRates` in `postRiskPremiums()`. If signed premium adjustments are intended, update `SwapCore` pricing, collateral and markup calculations to consume signed values without reverting.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/882/commits/a87bf3c78473c86b7b7a404c695dafc81453d2ed`

LOW L-19

ACKNOWLEDGED

Virtual NAV skips settlement pokes

CATEGORY	Unexpected Behavior
LOCATION	`VirtualShareLib.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`VirtualShareLib.computeVirtualSharePrice` computes the virtual LP share price from the current oracle state without executing the market oracles' `poke()` hooks. It reads the current reference index through `Utils.getFreshIndex` and then uses that value to model expiry settlement and remaining exposure. Actual settlement does not use the same oracle state. `SwapCore.updateMarketRateIndex` calls `Utils.pokeMarketOracles` before updating the market index and expired-swap settlement calls that function before resolving the expiry index. A healthy `IPokeable` oracle can return valid data before the poke and different valid data after the poke.

Therefore, `getPoolSharePriceVirtual` can report a share price that differs from the share price immediately after the same expired swaps are actually settled. `KairosMarketAdapter._realAssets` converts this virtual share price into vault-facing assets, so a parent vault can price deposits or withdrawals against a pre-poke NAV and then crystallize a different value once settlement is triggered. The issue requires a pokeable market oracle whose valid output changes on `poke()` and it is most relevant when active exposure remains after the expired swaps are removed.

RECOMMENDATION

Make virtual pricing and actual settlement consume the same committed oracle state. One safe approach is to require oracle pokes to be committed in a separate synchronization transaction before either `getPoolSharePriceVirtual` or settlement uses the data. Another option is to provide a standardized view-only post-poke preview and use it consistently for both virtual share pricing and settlement.

Settlement should not implicitly change oracle inputs after a virtual NAV has already been quoted from pre-poke data.

RESOLUTION

Status: Acknowledged.

LOW L-20

RESOLVED

Forced Bounds Reset Keeps Old Posting Cooldown

CATEGORY	Oracle
LOCATION	BaseTimelockedOracle.sol
REVIEW	Remediation Review 4

DESCRIPTION

Forced rate-bounds activation can leave the oracle invalid while blocking the recovery post that is supposed to restore it.

activateRateBoundsChange(true) skips the strict in-bounds check and calls _resetStoredRates(), causing all reads to return isValid=false until signers post a fresh in-bounds curve. The function documentation explicitly describes recovery as a single immediate post. However, the forced path does not reset lastPostTimestamp. If keepers post under the old bounds shortly before governance activates the forced bounds change, _validateTenorsAndCadence() still measures the next post against that old timestamp and reverts with UpdateTooFrequent. The stored rates have already been cleared, so the oracle remains invalid until minUpdateIntervalSec elapses. With the deployed-style 4 hour cadence, a forced emergency activation can therefore create a deterministic multi-hour oracle outage.

This affects both FixedRatePricingOracle and RiskRateOracle, because both inherit the shared cadence gate and both clear their stored rate state through _resetStoredRates(). During the outage, dependent market creation, swap entry, early-exit pricing, LP MTM paths, and risk-premium reads can fail because the oracle reports invalid data.

RECOMMENDATION

Reset lastPostTimestamp inside the forced activation branch after _resetStoredRates(), so the first recovery post can land immediately and the cadence re-arms on that successful post.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/882/commits/6a96a617a3321dc7b184eb35fadf41a7257534e6

LOW L-21

RESOLVED

Over-aged seed starts oracle invalid

CATEGORY	Validation
LOCATION	IndexBaseRateV1.sol
REVIEW	Remediation Review 4

DESCRIPTION

`IndexBaseRateV1` requires a constructor seed so the oracle can serve rates immediately after deployment. The constructor only checks that `_seedTimestamp` is not newer than `block.timestamp - twarWindowSec`. It does not check that the seed is close enough to that initial TWAR boundary to satisfy the oracle's own freshness rule.

If the seed is older than the allowed `maxLiveGapSec` window, deployment succeeds but the first rate read is already invalid. `_realizedTwar()` selects the old seed as the left anchor and then rejects it because `targetTime - left.timestamp > maxLiveGapSec`.

Therefore, an operator can deploy an oracle that looks successfully initialized but cannot pass market-creation probing or serve rates. Ordinary pokes cannot immediately repair the state because new observations are initially to the right of the TWAR boundary. The oracle remains unusable until enough time passes for fresh observations to bracket the boundary.

RECOMMENDATION

Validate the seed against the first TWAR boundary during construction. The constructor should reject any seed where:

```plain text
block.timestamp - twarWindowSec - _seedTimestamp > maxLiveGapSec
```

Equivalently, require the seed timestamp to match the intended boundary within a small explicit tolerance. Every accepted deployment should return a valid rate immediately when the source itself is valid.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/882/commits/a5649d4b0f6dc0181e93646b437db923be6b1333

LOW L-22

PARTIALLY RESOLVED

Tenor shortening can brick live markets

CATEGORY	Unexpected Behavior
LOCATION	`BaseTimelockedOracle.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`BaseTimelockedOracle` lets governance replace the active `tenorBuckets` set without checking whether existing markets still need longer tenors. `activateTenorBucketsChange()` assigns the pending bucket list directly and resets `lastPostTimestamp`, but it does not know which `swapTerm` values or remaining-tenor values are already consumed through the oracle.

This is unsafe because the read logic now fails closed when a requested tenor is greater than the longest active bucket. If governance shortens a curve from, for example, 180 days to 90 days while a 180-day market still has active exposure, reads for that market can start returning invalid immediately after activation.

Consequently, a locally valid bucket update can remove oracle support from a live market. New swaps can fail because `buySwap()` reads the market's full `swapTerm`. Active positions can also lose early-exit and LP mark-to-market availability while their remaining tenor is above the new curve maximum. This can block LP share-price reads, LP deposits or withdrawals that depend on active-pool valuation and `KairosMarketAdapter.realAssets()` for vaults using the affected market.

RECOMMENDATION

Do not allow a tenor-bucket change to remove terms still required by live markets.

Track the maximum required tenor for each oracle or keep an explicit registry of market terms and active remaining-tenor requirements. `activateTenorBucketsChange()` should reject a new bucket set whose longest bucket is below any registered live requirement. If tenor removal is needed, keep the old curve available until all dependent exposure has expired or migrated.

RESOLUTION

Status: Partially resolved.

`https://github.com/kairosswap/protocol/pull/890`

LOW L-23

Ghost swap escrow cannot be claimed

CATEGORY	Unexpected Behavior
LOCATION	KairosBuyerAdapter.sol
REVIEW	Remediation Review 4

DESCRIPTION

`KairosBuyerAdapter` documents that third parties can call `SwapCore.buySwap()` with `onBehalfOf = address(adapter)`. Those externally funded swaps are owned by the adapter, but they are not inserted into `_swapIds` because they did not go through `allocate()`.

That is mostly harmless when settlement pays tokens directly to the adapter, since the token balance is included in `_realAssets()`. The escrow case is different. If settlement or liquidation cannot transfer the buyer payout to the adapter, `SwapCore` stores the amount in `escrowedCollateral[swapId]` and leaves `settlementPayouts[swapId]` as zero. The adapter is then the only authorized caller that can recover the escrow, but `ACTION_CLAIM_ESCROW` always calls `_removeSwapId(swapId)` when `settlementPayouts(swapId) == 0`.

For a ghost swap, `_removeSwapId()` reverts with `swap not tracked`. The revert rolls back the `SwapCore.claimEscrow()` call, so the escrow remains stuck even after the token can be transferred again. The value is externally funded rather than vault principal, so the impact is limited to donated or mistaken adapter-owned swaps. Still, the contract-level comment says these positions self-heal once proceeds land at the adapter and that is not true when the proceeds first enter SwapCore escrow.

RECOMMENDATION

Make escrow recovery tolerate adapter-owned swaps that are not in `_swapIds`. For example, check whether the swap is tracked before pruning:

```plain text
if (payout == 0 && _isTracked(swapId)) {
_removeSwapId(swapId);
}
```

`Alternatively, reject claims for untracked swaps with a dedicated error before calling SwapCore.claimEscrow()` and document that externally funded adapter-owned swaps are unsupported when payout transfer fails. The safer option is to allow the claim and skip only the tracking cleanup for untracked swaps.

L-23 Ghost swap escrow cannot be claimed CONTINUED

RESOLUTION

Status: Reported during the fix review period.

https://github.com/kairosswap/protocol/pull/934

LOW L-24

Long min spacing creates oracle outages

CATEGORY	Validation
LOCATION	IndexBaseRateV1.sol
REVIEW	Remediation Review 4

DESCRIPTION

`IndexBaseRateV1` rejects `_minSpacingSec` values that are too short to retain enough ring-buffer history, but it does not reject values that are too long for the oracle's freshness rule. The constructor only checks `_minSpacingSec * (N - 1) >= _twarWindowSec`, then seeds `maxLiveGapSec` to `_twarWindowSec`.
That leaves a configuration where the left anchor is already too old for `_realizedTwar()` but `poke()` still reverts `TooEarly()`. For example, with a one-hour TWAR window and seven-hour `minSpacingSec`, the oracle starts valid. Two hours and one second after deployment, the TWAR boundary is more than `maxLiveGapSec` after the deploy head, so `getRateByTenor()` returns invalid. A keeper still cannot advance the ring because seven hours have not elapsed since `lastAdvanceTimestamp`.
Consequently, an accepted deployment can create predictable base-rate oracle outages even when the cumulative source is healthy and a keeper is ready to update the oracle. `SwapCore` treats invalid base-rate reads as hard failures through `Utils.getBaseRateOracle()` and `Utils.getBaseRateOracleBatch()`, so affected markets can reject buys, early exits and LP mark-to-market reads during those intervals.

RECOMMENDATION

Reject spacing values that can create an invalid-read interval before the next permitted poke. With the current default freshness bound, require `_minSpacingSec <= 2 * _twarWindowSec` or use the stricter operational default `_minSpacingSec == _twarWindowSec`.
If longer spacing is intentionally supported, derive the constructor check from the initial freshness rule:

```plain text
if (_minSpacingSec > _twarWindowSec + _twarWindowSec) revert InvalidSpacing();
```

Keep the existing lower-bound retention check as well, since the short-spacing and long-spacing failures are distinct.

L-24 Long min spacing creates oracle outages CONTINUED

RESOLUTION

Status: Reported during the fix review period.

https://github.com/kairosswap/protocol/pull/990

LOW L-25

BaseSwapRate average accepts bad rates

CATEGORY	Unexpected Behavior
LOCATION	BaseSwapRate.sol
REVIEW	Remediation Review 4

DESCRIPTION

`baseSwapRateWeightedAverage()` reads each source through `latestRoundData()`, discards the round metadata and casts the signed `answer` directly to `uint256` before applying borrow weights. Consequently, a negative source rate is not preserved as a signed value and is not rejected. It is reinterpreted as a huge unsigned integer. A source answer of `-0.01e18`, for example, becomes almost `type(uint256).max` and can make the weighted result nonsensical. The function also accepts stale or unanswered Chainlink-compatible rounds because it ignores `updatedAt`, `answeredInRound` and the other returned metadata.

The zero-borrow case is also unsafe as an oracle value. If every configured source reports zero borrow, the function returns `1e18`, which is a 100% base rate, instead of returning invalid or using a documented fallback.

This function is not currently used by `getRate()` or `latestRoundData()`, which limits the immediate blast radius. The risk is still real for direct consumers of `baseSwapRateWeightedAverage()` and for any future change that wires this helper into the live oracle read.

RECOMMENDATION

Keep the weighted-average calculation signed from source read through final result. Reject or return invalid when a source round is stale, unanswered or otherwise unhealthy. At minimum, validate `updatedAt != 0` and `answeredInRound >= roundId` for Chainlink-compatible sources before using the answer.

Define the zero-borrow behavior explicitly. Safer options are returning invalid, using a configured last-good value or using a clearly documented fallback rate. Do not return `1e18` unless a 100% rate is intentionally part of the oracle design.

RESOLUTION

Status: Reported during the fix review period.

LOW L-26

ProtocolFeeForegone omits fee context

CATEGORY	Events
LOCATION	`SwapEvents.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`ProtocolFeeForegone` is used for more than one kind of waived fee, but the event only reports `marketId`, `payer` and `amount`.
`SwapCore.createMarket()` emits this event when a market-creation fee cannot be delivered to the protocol multisig. `Utils.distributeProtocolFee()` emits the same event when a swap protocol fee cannot be delivered. Those two cases need different accounting context, but the event does not identify the fee class, fee token, intended recipient or the paired market created alongside the reported market ID. Consequently, an event-only accounting or monitoring system cannot reliably classify the waived fee. It can record a foregone market-creation fee as a swap protocol fee, assign the amount to the wrong token or attribute a paired market-creation fee to only one market. On-chain balances are not corrupted, so the impact is limited to revenue reconciliation, dashboards, alerts and downstream monitoring that relies on logs.
The adjacent `MarketCreationFeeCollected` event includes both market IDs and the `feeToken` for the successful creation-fee case. The foregone creation-fee case loses those same fields.

RECOMMENDATION

Use separate events for separate fee classes, for example `ProtocolSwapFeeForegone` and `MarketCreationFeeForegone`. The market-creation event should include both market IDs, the creator, `feeToken`, `amount` and the intended protocol recipient.
If preserving one event is required, add a fee-type enum, fee token, recipient and optional paired-market ID so indexers can reconstruct the waived fee from logs alone.

RESOLUTION

Status: Reported during the fix review period.

LOW L-27

Permissionless markets bloat required tenors

CATEGORY	Validation
LOCATION	`UpgradableTenorRateOracle.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`UpgradableTenorRateOracle.setRequiredTenors()` accepts an arbitrary-length tenor array. It stores the full array and immediately validates the whole set against the current downstream oracle. `_validateDownstreamHealth()` then performs one single-tenor read for the default tenor and every required tenor, followed by a batch read over the same full list.

This becomes fragile because market creation is permissionless. `SwapCore.createMarket()` lets any caller create markets against a shared base-rate wrapper with distinct `swapTerm` values. Each non-terminated market keeps constraining that wrapper even if it has no locked collateral. Since only the market owner can terminate the market, governance cannot prune attacker-created terms on its own.

Consequently, a griefer can create many idle markets that force the wrapper's required-tenor maintenance set to grow. If governance keeps the set complete, `setRequiredTenors()` and later downstream activation must scan an increasingly large list in one transaction. If governance omits attacker-created terms, the replacement check no longer proves that the downstream oracle serves every live market term.

RECOMMENDATION

Do not make wrapper safety depend on an unbounded manually supplied tenor array. Cap the number of required tenors, deduplicate entries before storing them and reject new markets that would exceed the cap for a shared wrapper.

If arbitrary market creation must remain open, add a way for governance to exclude or retire idle markets from required-tenor maintenance when they have no locked collateral and no active LP exposure.

RESOLUTION

Status: Reported during the fix review period.

LOW L-28

ReferenceRate ignores configured rates

CATEGORY	Unexpected Behavior
LOCATION	ReferenceRate.sol
REVIEW	Remediation Review 4

DESCRIPTION

`ReferenceRate.setReferenceRate()` writes `newRate` into `_referenceRate`, but no public read uses `_referenceRate`. `referenceRateWad()` instead derives a value between 5% and 10% from `keccak256(abi.encodePacked(block.timestamp / 300, address(this)))` and `latestRoundData()` plus `getRate()` both forward that computed value while reporting it as valid.

Consequently, the oracle owner cannot set the rate that `SwapCore` consumes through `IRateOracle.getRate()`. Markets that use this oracle with `SpotRate` or `SpotCompoundRate` accrue their floating index from a predictable time/address formula rather than from the configured reference rate or an external source.

This creates a concrete timing profit when the market index is idle. `RateIndexLib.update()` applies the current spot rate to the whole elapsed period since the last update. A normal user can therefore compute a future high five-minute slot, open a `BUY_FIXED` swap exactly one term before that slot, avoid intermediate index updates and settle at the chosen timestamp. The first settlement update then applies the high pseudo-rate to the whole swap term. If the fixed/base rate is lower, the buyer receives LP collateral even though no external reference market ever produced that realized rate.

This also contradicts the protocol assumption that oracle rates are external trusted inputs and that Kairos does not use pseudo-random number generation for rates.

RECOMMENDATION

Remove the pseudo-random calculation from `referenceRateWad()` and return `_referenceRate` instead. Keep the storage type signed if negative reference rates are supported by the protocol and have `latestRoundData()` and `getRate()` forward the same configured value. If this contract is intended only as an unsafe chaos/mock oracle, make it impossible to use in market creation for non-test deployments or rename and document it as unsafe.

For deployed spot-rate markets, also avoid treating one endpoint spot reading as the realized rate for a stale interval. Use cumulative/checkpointed reference-rate sources or tightly bound how much elapsed time one spot update may backfill.

L-28 ReferenceRate ignores configured rates CONTINUED

RESOLUTION

Status: Reported during the fix review period.

LOW L-29

PythAdapter rejects fresh zero-percent rates

CATEGORY	Validation
LOCATION	`PythAdapter.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`PythAdapter.getRate()` treats any Pyth price equal to zero as invalid. It returns `(0, false)` before freshness and confidence checks can distinguish a stale or missing feed from a fresh feed whose rate is exactly 0%.

For rate feeds, 0% can be a valid value. Pyth's local `getPriceNoOlderThan()` implementation already rejects stale data by checking `publishTime` against the requested age and `PythAdapter` separately rejects `publishTime == 0`, future timestamps, stale timestamps, low-confidence readings and unscalable values.

Consequently, a `SpotRate` market that uses `PythAdapter` as its reference oracle becomes temporarily unusable while the external feed reports exactly 0%. `RateIndexLib.update()` receives `isValid == false` and reverts with `E606`, so `updateMarketRateIndex()`, `buySwap()` and LP deposits that call `supplyCollateral()` cannot proceed even though the oracle update is fresh.

RECOMMENDATION

Accept zero as a valid signed rate when the Pyth update has a nonzero, nonfuture, fresh `publishTime`, passes the configured confidence check and can be represented at the target scale.

Remove the unconditional `pythPrice.price == 0` invalidation. If a scaled zero can also result from truncating a very small nonzero feed, handle that case separately from an exact zero rate so legitimate 0% updates remain usable.

RESOLUTION

Status: Reported during the fix review period.

INFORMATIONAL **I-01** ACKNOWLEDGED

Adapter NAV omits idle deallocated cash

CATEGORY	Reentrancy
LOCATION	`KairosMarketAdapter.sol`
REVIEW	Remediation Review 4

DESCRIPTION

During a partial LP deallocation, `KairosMarketAdapter.deallocate()` calls `SwapCore.withdrawCollateral()` with the adapter as the receiver. `SwapCore.withdrawCollateral()` debits the adapter's LP shares and reduces `pool.totalCollateral`, then transfers the withdrawn asset to the adapter.

For a callback-capable market asset, that transfer can reenter external vault logic after the withdrawn tokens have been credited to the adapter, but before `adapter.deallocate()` returns and before the vault performs its post-return `transferFrom`. At that point, `KairosMarketAdapter._realAssets()` values only the remaining LP shares:

```plain text
assets = Math.mulDiv(pos.shares, sharePrice, WAD);
```

`It does not add` asset.balanceOf(address(this)) `. The withdrawn collateral is therefore real adapter-owned value, but it is invisible to` realAssets()` during the callback window.

RECOMMENDATION

Remove or guard the callback-observable window. The safest options are to support only non-callback market assets for this adapter, or require the parent vault to block reentrant deposit, mint, allocation and deallocation flows while adapter deallocation is in progress. If callback-capable assets must be supported, add narrowly scoped in-flight withdrawal accounting or a deallocation reentrancy guard so `realAssets()` cannot be observed in the reduced-shares, pre-pull state.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-02

ACKNOWLEDGED

Split exits round away swap fees

CATEGORY	Rounding
LOCATION	`Utils.sol, SwapCore.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`calculateEarlyExitSettlement` computes `earlyExitFee = (netObligation * market.earlyExitFee) / WAD` for one swap at a time. The division floors the result in the market token's smallest unit. `exitSwapEarly` accepts an array of exits, but it marks each swap and then settles each swap independently through `_settleSwaps`. There is no aggregation of a user's exits before the fee is calculated. A trader can therefore split one aggregate exposure into many swaps whose individual fee products are below one token unit, causing every per-swap fee to round to zero.

RECOMMENDATION

Accumulate fees at higher precision or compute them over aggregated exits per user/market. Alternatively round non-zero early-exit fees up or enforce token-aware minimum notional/net-obligation thresholds so per-swap truncation is immaterial.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-03 RESOLVED

Callback credits wrapped payout early

CATEGORY	Reentrancy
LOCATION	`SwapCore.sol, Utils.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`Utils.settleSwap` sets `swap.settled = true` before it transfers the settlement token to the swap owner. For wrapped positions, that owner is `SwapPositionWrapper`. `SwapCore._settleSwaps` only writes `settlementPayouts[swapId]` after `Utils.settleSwap` returns.

If the market token executes recipient code during `transfer`, a third-party direct call to `SwapCore.makePayment` can enter `SwapPositionWrapper.settle(tokenId)` from inside the settlement transfer. At that point the wrapper observes `settled == true`, skips its own `makePayment` call, reads the still-zero `settlementPayouts[swapId]` and sets `_credited[tokenId] = true` without increasing `redeemable[tokenId]`.

The outer settlement then finishes and records the real positive payout in `settlementPayouts[swapId]`, but the wrapper has already marked the NFT as credited. Later `settle` or `redeem` calls are idempotent no-ops for crediting, so the NFT remains redeemable for zero.

RECOMMENDATION

Write `settlementPayouts[swapId]` before any external token transfer can reenter wrapper crediting logic or make wrapper crediting depend on a finalized payout flag that is distinct from `swap.settled`. Alternatively, explicitly reject callback-capable settlement tokens at market creation.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/850`

INFORMATIONAL I-04 RESOLVED

Split swaps bypass aggregate collateralization

CATEGORY	Rounding
LOCATION	SwapFormulas.sol
REVIEW	Remediation Review 4

DESCRIPTION

`calculateBuyerCollateral()` and `calculateLPCollateral()` calculate opening margin for one swap at a time and round the token amount down. `SwapCore.buySwap()` then treats each rounded value as fully collateralized as long as it is at least `minCollateral`.

This lets a buyer capture the discarded fractional collateral by splitting one exposure into many small swaps. The aggregate position can require materially more collateral under the same formula than the sum of the independently rounded split positions.

For example, in a zero-decimal market with a 5% base rate, one-year term, 1x leverage, no utilization fee, no risk premium and `minCollateral = 1`, a `39` notional swap has an exact buyer and LP collateral requirement of `1.95` tokens. Both functions return `1`. Repeating that opening 99 times locks only `99` pool tokens while creating `3,861` notional. The same aggregate notional would require `193` LP collateral tokens under the unsplit formula.

Consequently, low-decimal markets or markets with an economically small `minCollateral` can allow more notional per unit of collateral than intended. If rates later move in the buyer's favor, the buyer can extract more LP backing than the aggregate collateral model should have allowed. The effect is usually negligible for conventional 6- or 18-decimal assets when `minCollateral` is set well above one atomic unit.

RECOMMENDATION

Round buyer collateral and LP collateral up whenever a nonzero exposure requires a fractional token amount. Prefer a single full-precision calculation with ceiling rounding for the whole expression:

```plain text
ceil(notional * rate * term / (leverage * SECONDS_IN_YEAR))
```

`Keep the inverse available-liquidity calculation conservative by rounding down. minCollateral` should remain a token-specific safety floor, but it should not be the only protection against repeated per-swap rounding loss.

I-04 Split swaps bypass aggregate collateralization CONTINUED

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/884

INFORMATIONAL I-05 RESOLVED

Fee receiver can block market actions

CATEGORY	Unexpected Behavior
LOCATION	`SwapCore.sol, Utils.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`SwapCore` delivers protocol fees synchronously while executing user operations. During `createMarket()`, if a native market-creation fee is active, the function sends ETH to `admin.getProtocolMultisig()` and reverts with `E151` when the receiver call fails. During `buySwap()`, the buyer's protocol fee is first pulled into `SwapCore`, then `Utils.distributeProtocolFee()` pushes the protocol share to `protocolMultisig` with `safeTransfer`. Any token revert from that transfer reverts the entire swap opening.

This makes the configured fee receiver a hard liveness dependency for unrelated users. A multisig contract can be valid when configured and later become unable to receive ETH, for example because its fallback changes or rejects direct transfers. For ERC20 protocol fees, a market token can later reject or blacklist transfers to the multisig. In both cases, the caller cannot work around the receiver failure from the affected operation.

Consequently, fee-bearing market creation can be disabled by a rejecting native-fee receiver and fee-bearing swap opening can be disabled for any market token that refuses protocol-fee delivery to the multisig. Recovery requires admin action to disable the fee or rotate to a receiver that accepts the asset, even though market creators and buyers supplied valid inputs.

The creator-fee portion already treats receiver failure as non-fatal: if the market owner cannot receive its share, that share stays with the protocol amount. The protocol-fee portion does not have the same protection, so the passive protocol receiver can make user-critical operations unavailable.

RECOMMENDATION

Do not require fee delivery to the protocol multisig to succeed inside `createMarket()` or `buySwap()`. Account for the fee during the user operation, store claimable balances by asset and recipient and let the current multisig claim fees separately.

A failed claim should leave the accrued balance intact and should not revert market creation or swap opening. This keeps fee accounting complete while removing the receiver as a synchronous liveness dependency.

I-05 Fee receiver can block market actions CONTINUED

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/commit/2aaa6c62

INFORMATIONAL I-06 RESOLVED

Unset Morpho ids report zero rates

CATEGORY	Validation
LOCATION	ReferenceRateMorpho.sol
REVIEW	Remediation Review 4

DESCRIPTION

`ReferenceRateMorpho` does not verify that `marketId` belongs to a created Morpho market. `_getRate()` reads `idToMarketParams(marketId)` and `market(marketId)`, but it treats the default zero values as a valid zero-rate market. When `marketParams.irm == address(0)`, the function returns `0`. `AbstractReferenceRate.getRate()` then wraps that answer as `(0, true)`. During `SpotRate` or `SpotCompoundRate` market creation, `RateIndexLib.initialize()` only checks the returned validity flag. It does not know whether the Morpho market exists, has a nonzero IRM or has ever been initialized. Consequently, a typo, stale deployment config or permissionless market creator can publish a Kairos market whose reference oracle points at an uncreated Morpho market. The reference index then starts from a fresh-looking 0% rate instead of failing closed. If LPs later fund the market, buyers can trade against a flat reference index that does not represent any real Morpho source market.

RECOMMENDATION

Reject uncreated Morpho markets during construction or during every read. At minimum, check `morpho.market(marketId).lastUpdate != 0` before returning a valid rate.
Also propagate invalidity through the reference-rate interface. If the Morpho market is missing, `getRate()` should return `(0, false)` or the constructor should revert, instead of returning a valid zero rate. A genuinely created zero-IRM market can remain valid if it has a nonzero created-market signal.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/885

INFORMATIONAL **I-07** ACKNOWLEDGED

BaseSwapRate cannot serve SwapCore

CATEGORY	Warning
LOCATION	`BaseSwapRate.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`BaseSwapRate` presents itself as a base-rate oracle, but it only implements the single-rate `IRateOracle` interface. It exposes `getRate()`, `latestRoundData()`, `decimals()` and `description()`. It does not implement `ITenorRateOracle.getRateByTenor()` or `ITenorRateOracle.getRates()` and it does not advertise `ITenorRateOracle` through ERC-165.

`SwapCore` requires the base-rate oracle to support those tenor-dependent reads during market creation. `_initializeMarket()` calls `Utils.probeOraclesForMarketCreation()`, which casts `baseSwapRateOracle` to `ITenorRateOracle` and calls both `getRateByTenor(swapTerm)` and `getRates([swapTerm])`. Because `BaseSwapRate` has neither selector and no fallback, a market that uses it as `baseSwapRateOracle` reverts during creation.

Consequently, operators cannot use `BaseSwapRate` in the base-rate slot despite the contract name and its otherwise valid single-rate oracle surface. Any deployment or integration that selects it for `baseSwapRateOracle` will fail before the market is created.

RECOMMENDATION

If `BaseSwapRate` is intended to be usable as a `SwapCore` base-rate oracle, implement the full `ITenorRateOracle` interface. `getRateByTenor()` should return the configured base rate for the requested tenor and `getRates()` should return one value per requested tenor in the same order.

If the contract is only intended for single-rate reference or risk-premium use, rename it or document that it must not be supplied as `baseSwapRateOracle`.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-08 RESOLVED

Cached oracle health is skipped

CATEGORY	Validation
LOCATION	RateIndexLib.sol
REVIEW	Remediation Review 4

DESCRIPTION

`RateIndexLib.initialize()` validates a reference oracle only the first time that oracle address is used. When `lastUpdates[oracle]` is already nonzero, the function only checks that the stored `RateConvention` matches the requested convention. It does not read the oracle again or verify that the oracle is currently healthy.

`SwapCore._initializeMarket()` relies on this function as the reference-oracle health check during market creation. Therefore, once one market has initialized a reference oracle, later markets can reuse the same oracle address even if it is now paused, stale, under an emergency cutoff or otherwise returning `isValid = false`.

Consequently, a new market pair can be recorded and its creation fee can be collected while its reference oracle is unusable. The market then appears valid to indexers, adapters and users, but rate-dependent actions such as LP deposits, swap entry, liquidation, settlement refreshes and active-pool valuation can revert until the oracle recovers.

RECOMMENDATION

Probe the reference oracle during every market creation, even when its rate index has already been initialized.

Keep `RateIndexLib.initialize()` idempotent for stored index setup, but add a separate current-health check in `SwapCore._initializeMarket()` after any configured oracle poke and before the market creation completes. For `Cumulative` markets, require `getCumulativeIndex()` to return a nonzero valid index. For `SpotRate` and `SpotCompoundRate` markets, require `getRate()` to return `isValid = true`.

RESOLUTION

Status: Resolved.

https://github.com/kairosswap/protocol/pull/886

INFORMATIONAL I-09 RESOLVED

Redeem callback can sell live NFT

CATEGORY	Reentrancy
LOCATION	`SwapPositionWrapper.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`SwapPositionWrapper.redeem()` transfers the redeemed settlement tokens before it burns the ERC721 position token. The wrapper deletes its accounting first, then calls `safeTransfer(msg.sender, amount)` and only after that calls `_burn(tokenId)`.

If the settlement token can execute recipient code during transfer, the redeeming holder can run code while the NFT still exists. The wrapper's `nonReentrant` modifier protects `redeem()` itself, but it does not protect the inherited ERC721 `transferFrom()` and `safeTransferFrom()` functions. A contract holder can therefore transfer the still-live NFT during the token callback.

Consequently, a malicious holder can redeem the position, receive the payout and then sell or transfer the NFT during the payout callback. When control returns to `redeem()`, `_burn(tokenId)` burns the token from its new owner. A buyer that accepted the NFT in the same transaction can pay for an asset that is destroyed before the transaction completes.

This requires a callback-capable settlement token and an atomic sale or standing bid that can be filled from the callback. The issue is therefore low severity unless those token and marketplace assumptions are part of an intended deployment.

RECOMMENDATION

Burn the NFT before transferring settlement tokens or otherwise make the token non-transferable once redemption starts. A minimal fix is to call `_burn(tokenId)` immediately after the ownership and settlement checks, before any external token transfer is attempted.

If the contract must keep the burn after payout for compatibility reasons, override ERC721 transfer and approval functions so a token marked as redeeming cannot be transferred or approved during redemption.

RESOLUTION

Status: Resolved.

`https://github.com/kairosswap/protocol/pull/887`

INFORMATIONAL I-10 RESOLVED

Tenor cardinality can exhaust gas

CATEGORY	Gas Optimization
LOCATION	`BaseTimelockedOracle.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`BaseTimelockedOracle` accepts any non-empty sorted `tenorBuckets` array. The constructor and `setTenorBuckets()` do not enforce a maximum number of buckets.

That matters because every successful oracle update must include the complete active tenor set. `_validateTenorsAndCadence()` requires the posted `tenors` array to have exactly the same length and order as `tenorBuckets`. `FixedRatePricingOracle.postForecasts()` and `RiskRateOracle.postRiskPremiums()` then validate and store each entry in that full vector.

The read side also scales poorly with the configured curve size. `_getRatesInternalWith()` first scans every configured bucket to find the earliest timestamp and then scans every bucket again to copy rates into memory. After that, each requested tenor calls `_interpolateFromMemory()`, which scans the full bucket list once more. This makes batch reads grow with `bucketCount * requestedTenors`.

Consequently, a syntactically valid tenor expansion can make the oracle impractical to reseed after activation because a complete signed post may no longer fit within the available gas budget. Smaller but still oversized curves can also make multi-tenor base-rate reads expensive enough to break LP share pricing. `Utils.calculateTotalUnrealizedPnL()` can request one tenor for each active market bucket, so active-pool deposits, withdrawals, share-price reads and `KairosMarketAdapter.realAssets()` inherit the inflated oracle cost.

RECOMMENDATION

Add a maximum tenor-bucket count and enforce it in both the constructor and `setTenorBuckets()`. Size the cap against the worst-case first post for `FixedRatePricingOracle` and `RiskRateOracle`, plus the largest expected production batch read from LP valuation.

Also consider replacing `_interpolateFromMemory()` with binary search or with a sorted one-pass merge for batch reads. That keeps multi-tenor reads from multiplying the full curve length by the number of requested tenors.

RESOLUTION

Status: Resolved.

I-10 Tenor cardinality can exhaust gas CONTINUED

https://github.com/kairosswap/protocol/pull/888

INFORMATIONAL I-11 ACKNOWLEDGED

Pool-side self-liquidation captures reward

CATEGORY	Warning
LOCATION	`Utils.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`Utils.liquidateSwap` pays the pool-side liquidation reward to the unrestricted `liquidator` address and pays the remaining pool backing to `swap.userAddress`. `SwapCore.liquidateSwap` passes `msg.sender` as `liquidator`. Neither function rejects `msg.sender == swap.userAddress`.
Consequently, a buyer can wait until their in-the-money swap crosses the pool-side liquidation threshold, then call `liquidateSwap` on their own position. The buyer receives the normal buyer payout and the keeper reward. This gives the buyer more than the live accrued claim and more than the liquidation-aware buyer value that the protocol reports before liquidation.

RECOMMENDATION

Reject self-liquidation when the pool side is liquidatable or do not pay the keeper reward when `liquidator == swap.userAddress`. The simplest fix is to require a third-party liquidator for pool-side liquidations before any state is mutated. If self-liquidation should remain possible, settle the buyer at `poolBacking - liquidatorReward` and leave the reward in the pool instead of transferring it to the buyer.

RESOLUTION

Status: Acknowledged.

INFORMATIONAL I-12

Unbounded poke spacing disables refresh

CATEGORY	Validation
LOCATION	`Admin.sol`
REVIEW	Remediation Review 4

DESCRIPTION

`Admin.setPokeMinSpacingSec()` only enforces a 60 second lower bound. It does not enforce any upper bound, so the owner can set `pokeMinSpacingSec` to a value such as `type(uint32).max`.
`SwapCore` reads this value live and passes it to `OraclePokeLib`. That library skips a poke whenever `block.timestamp - lastPokes[oracle] < minSpacingSec`. Therefore, an accidentally oversized value can suppress protocol-triggered pokes for years after an oracle has been poked once.
This is an owner misconfiguration issue which impact is still relevant for poke-dependent base or risk oracles because automatic refreshes can stop working. Affected markets can then reject swaps, exits, settlements or NAV reads once those oracles become stale or invalid unless an out-of-band refresh recovers them.

RECOMMENDATION

Add a maximum value for `pokeMinSpacingSec` and reject larger updates. Size the ceiling to the slowest supported oracle cadence, for example a few hours or one day. Consider timelocking large increases while still allowing decreases immediately.

RESOLUTION

Status: Reported during the fix review period.

INFORMATIONAL I-13

Split buys avoid fees via rounding

CATEGORY	Rounding
LOCATION	SwapCore.sol
REVIEW	Remediation Review 4

DESCRIPTION

`buySwap` computes the upfront protocol fee independently for each swap and rounds the result down:

```plain text
protocolFee = Math.mulDiv(notionalAmount, feeRate * swapTerm, SwapFormulas.SECONDS_IN_YEAR * SwapFormulas.WAD);
```

A buyer can split one intended exposure into smaller buys whose individual fees are each below one raw token unit. Each buy then records `protocolFee == 0`, even though the same aggregate notional would pay a nonzero fee.

The dust guard does not prevent this in all markets. `createMarket` only requires `minCollateral > 0` and `buySwap` checks `minCollateral` against the rounded-up buyer and LP collateral, not against the protocol fee. Therefore low-decimal markets or markets with small raw token units, can accept trades that satisfy the collateral floor while still rounding the protocol fee to zero.

RECOMMENDATION

Do not allow nonzero fee configurations to round to zero on accepted buys. A minimal fix is to round the protocol fee up when `feeRate > 0` or to require `protocolFee > 0` for fee-enabled markets unless the protocol explicitly supports fee-free dust trades.

If rounding up by one raw unit is too punitive for high-decimal tokens, enforce a market-level minimum notional or minimum protocol fee that is derived from `feeRate`, `swapTerm` and token decimals. Add a regression test that compares an aggregate buy with split buys and asserts the split sequence cannot pay less total protocol fee for the same total notional.

RESOLUTION

Status: Reported during the fix review period.

INFORMATIONAL I-14

SpotCompound snapshots truncate extreme indexes

CATEGORY	Warning
LOCATION	RateIndexLib.sol
REVIEW	Remediation Review 4

DESCRIPTION

`RateIndexLib._appendSnapshot()` stores each historical index as `uint208(val)` without checking whether the live `uint256` index fits. This is unsafe for `SpotCompoundRate` markets.

`indexFromCumulativeRateTimeCompound()` can produce indexes above `type(uint208).max` before its explicit exponent guard reverts, so the live index remains correct while the historical snapshot silently keeps only the low 208 bits.

Normal expiry settlement uses the historical snapshot, not the live `uint256` index. `_settleSwaps()` resolves `expiryIndex` with `getIndexAt()` and `Utils.settleSwap()` passes that value into `SwapFormulas.deriveRateFromIndex()`. Therefore a truncated expiry snapshot understates the realized floating rate. A `BUY_FLOATING` buyer who should approximately net to zero against the fixed leg can instead receive LP collateral, capped only by `swap.poolCollateralBacking`.

The trigger is an extreme-rate or long-duration prerequisite, not an ordinary market condition. The snapshot width is crossed only after roughly 82 WAD-years of accumulated compound rate-time. For example, that means about 100% APR for 82 years, 500% APR for 16.4 years or 10,000% APR for 0.82 years. This makes the likelihood low for normal Aave, Compound and similar rate markets, but the code does not enforce a bound that keeps `SpotCompoundRate` snapshots inside `uint208`.

A local test used the high-rate case to cross the boundary quickly: a one-year term and a 100 WAD reference/base rate. This rate is accepted by the market and oracle bounds and remains below the exponent overflow guard. At maturity, the live index is `2.688e70`, but the stored expiry snapshot is the truncated `uint208` value `4.020e62`. Settlement derives an `81.98` WAD floating rate instead of the correct `100` WAD rate and transfers all `5,555,555.555556` USDC of locked LP backing to the buyer.

RECOMMENDATION

I-14 SpotCompound snapshots truncate extreme indexes CONTINUED

Do not silently narrow settlement-critical index history. Store snapshot indexes as `uint256` or fail closed before appending when the computed index does not fit the packed snapshot type. If packed storage must be retained, cap `SpotCompoundRate` market parameters and oracle updates so every possible settlement index is guaranteed to fit in `uint208`.

RESOLUTION

Status: Reported during the fix review period.

§ 05

Disclaimer.

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

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

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

The assessment services provided by Guardian is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis.

Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

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

§ 06

About Guardian.

Guardian is a smart contract security firm. Every engagement is worked by two independent teams competing on the same scope, each with its own tooling and methods, so no single reviewer's blind spot survives.

$45B+
DIGITAL ASSETS UNDER REVIEW

200+
ENGAGEMENTS COMPLETED

500+
CRITICAL & HIGH FINDINGS REPORTED

Reach us at audits@guardianaudits.com.