KairosSwap

Prepared by: Octane Security
Date: May 22, 2026
Target: KairosSwap - AMM-Based Interest Rate Swap Platform

1. Executive Summary

Octane Security conducted an Adversarial Research Engagement on the KairosSwap interest rate swap protocol. The engagement covered the core protocol surface: swap lifecycle (creation through settlement and liquidation), LP pool accounting and share pricing, oracle integration paths (consumption, not oracle adapter implementations), adapter integrations for Morpho Vault V2, fee calculations, and the permissionless settlement and liquidation mechanics across SwapCore, Utils, SwapFormulas, RateIndexLib, and the adapter layer.

The engagement identified **20 vulnerabilities**, each validated with a dedicated proof-of-concept. Across 20 PoC files, **155 test functions** demonstrate the claimed issue end-to-end. No finding appears without runnable proof.

9 distinct vulnerability classes were identified — structural patterns that recur across the codebase, not isolated bugs. The vulnerability classes span adapter and external balance accounting, transaction-ordering share price manipulation, accounting path inconsistencies, non-binding previews with missing slippage guards, algebraic time-window mismatches, oracle integration gaps, LP dilution via share price floors, blacklistable token interactions, and inconsistent rate sources.

Severity breakdown:

Severity	Count
Critical	1
High	3
Medium	7
Low	8
Informational	1
Total	**20**

Top findings:

The KairosBuyerAdapter uses a delta-only forwarding pattern for token transfers and lacks any sweep mechanism. Because swap settlement is permissionless, any third party can call `makePayment` and settle swaps before the vault's `deallocate` path runs. Settlement transfers proceeds directly to the adapter, but the adapter cannot forward pre-existing balances — only per-call deltas. Once settled out-of-band, the vault's normal `deallocate` reverts because the swap is already marked settled, permanently stranding both settlement proceeds and buyer collateral at the adapter with no on-chain recovery path.

LP share prices are computed from uncapped obligations in the mark-to-market calculation, but settlement caps actual payouts at available pool collateral. This gap is exploitable: an attacker can deposit when the share price reflects maximum uncapped loss, trigger settlement (which only realizes the smaller capped amount), and withdraw at the recovered price — extracting value from existing LPs. The same pattern extends to liquidations, where liquidators are positioned to double-dip on the capped-vs-MtM gap.

The buyer-side liquidation path in `Utils.liquidateSwap` sets `swap.collateralBalance = collateralToPool` instead of zeroing it, while simultaneously crediting the same amount to `pool.totalCollateral`. This double-counts the tokens — they are claimed by both the pool and the settled swap record — breaking the Token Balance Solvency invariant. Normal settlement and pool-side liquidation correctly zero the swap's collateral balance; only the buyer-liquidation branch has this bug.

Grouped themes across the 20 findings:

- Oracle integration gaps (5 findings) — missing normalization, unconditional dependency, stale index on invalid oracle, same-block index locking, stale view-path reads
- Transaction-ordering share price manipulation (4 findings) — settlement ordering, JIT liquidity, immediate bucket P&L, deterministic bucket targeting
- Adapter and external balance accounting (3 findings) — delta-only forwarding strands tokens, raw `balanceOf` inflatable by donations
- Non-binding previews / missing slippage (2 findings) — early exit preview drift and buySwap rate manipulation
- Time-window mismatches (2 findings) — post-expiry rate derivation drift and bucket MtM algebraic cancellation
- Accounting path inconsistency (1 finding) — buyer liquidation double-counts collateral
- LP dilution via share price floor (1 finding) — MIN_SHARE_PRICE enables opportunistic minting
- Blacklistable token interaction (1 finding) — push-based transfer revert locks pool collateral
- Inconsistent rate source (1 finding) — spot vs tenor-based rates between early exit and MtM

---

2. Scope and Methodology

2.1 Target Scope

The engagement covered the core protocol contracts under `packages/hardhat/contracts/` at commit `1c7f9f29`.

Core contracts in scope:

Contract	Role
SwapCore	Main entry point: market creation, LP operations, swap lifecycle, settlement, liquidation
Utils (library)	Settlement pipeline, liquidation logic, bucket math, share price calculation
SwapFormulas (library)	Payment calculations, collateral formulas, rate index math
RateIndexLib (library)	Global per-oracle cumulative floating rate index

Contract	Role
Admin	Global configuration: pause flags, fee parameters, trade size limits, collateral caps
KairosBuyerAdapter	Morpho Vault V2 buyer position integration
KairosMarketAdapter	Morpho Vault V2 LP position integration
KairosBuyerAdapterFactory	Factory for buyer adapter deployment
KairosMarketAdapterFactory	Factory for LP adapter deployment
OracleFactory / SimpleOracle	Chainlink composite price oracle deployment
Views	Read-only view contract for off-chain consumers

2.2 Methodology

The engagement followed a three-phase methodology designed to maximize detection depth through structured context engineering, directed platform analysis, and exhaustive exploitation validation.

Phase 1 — Analytical Foundation. Comprehensive context engineering produced architecture documentation mapping component boundaries, data flows, and state mutation paths; threat modeling with prioritized attack categories and paths across four severity tiers; fund-flow analysis tracing every token path and collateral state dependency across the swap lifecycle from LP deposit through settlement and liquidation; invariant identification covering accounting, access control, economic, and timing properties; and historical vulnerability research analyzing prior exploits in comparable share-based pools, oracle-dependent protocols, and precision-sensitive AMM systems. Client intake responses informed detection parameters and analysis focus areas.

Phase 2 — Octane Platform Analysis. The Octane platform's analysis ran against the Phase 1 analytical foundation with expanded compute and targeted detection categories tailored to KairosSwap's specific attack surfaces: pool accounting integrity, settlement correctness, share price integrity, negative rate handling, collateral and rounding, and early exit timing. The platform's multi-stage analysis pipeline examined LP pool accounting and solvency boundaries, oracle integration and cumulative rate index manipulation paths, swap settlement and liquidation correctness under signed arithmetic, bucket system integrity and share price manipulation vectors, negative rate edge cases across the settlement and collateral calculation pipeline, collateral rounding direction and utilization fee discontinuities, early exit timing and projection direction conditions, and adapter token transfer patterns. Candidate findings were produced across all major protocol domains.

Phase 3 — Exploitation and Validation. Every candidate finding received a dedicated proof-of-concept designed to confirm or disprove the claimed vulnerability. Each PoC validates the root cause, quantifies the exact impact boundary, documents attack economics (capital required, transaction count, timing constraints, net profit or loss), and proves what the attack cannot do alongside what it can. All 20 PoCs successfully demonstrated the claimed vulnerability; candidates that could not be dynamically proven were discarded.

---

3. Findings Summary

#	Title	Severity
1	Missing sweep and delta-only forwarding in KairosBuyerAdapter enables third-party settlement to strand funds, causing permanent freezing of vault assets	Critical
2	Permissionless selective settlement in SwapCore.makePayment causes manipulated LP share price and fund extraction	High
3	Non-binding early-exit preview without bounds in early-exit settlement flow causes front-runnable and drift-prone user losses	High
4	Double-counting buyer collateral in Utils.liquidateSwap buyer-liquidation path causes Token Balance Solvency invariant break	High
5	Time-window mismatch in bucket MtM (Utils.calculateBucketUnrealizedPnL) causes LP share price mispricing and fund extraction	Medium
6	Missing decimals normalization in Utils.calculateTotalUnrealizedPnL projected-rate handling causes LP share price mispricing and value extraction	Medium
7	Per-bucket external base oracle dependency in share-price and settlement causes market DoS and frozen LP funds	Medium
8	Share-price floor at MIN_SHARE_PRICE in calculateLpSharePrice during underwater MtM causes severe LP dilution	Medium
9	Stale-index best-effort update in RateIndexLib.update on invalid reference oracles causes direct LP principal loss via mis-settlement and mispriced LP share changes	Medium
10	Direct ERC20 payouts to user-specified recipient in settlement/liquidation with blacklistable tokens cause permanent bricking and locked pool collateral	Medium
11	Inconsistent projected rate source (spot vs tenor) in early-exit settlement vs MtM causes JIT LP share-price extraction/dilution	Medium
12	Same-transaction LP deposit/withdrawal manipulating availableCollateral in SwapCore.buySwap causes permanent LP fee loss	Low
13	Post-expiry floating rate derivation window mismatch in normal settlement causes LP principal loss	Low
14	Same-block index sample locking in RateIndexLib.update causes LP withdrawal overpayment (principal loss to remaining LPs)	Low
15	Immediate bucket P&L inclusion of new swaps without min-shares-out in LP minting causes over-minting and LP principal loss	Low
16	Timestamp-based bucket assignment with tenor-dependent per-bucket MtM in LP share pricing enables JIT manipulation causing LP principal loss	Low
17	Lack of on-chain slippage guard and non-binding rate preview in SwapCore.buySwap causes buyer rate manipulation and increased payments	Low

#	Title	Severity
18	Externally manipulable token balance in KairosBuyerAdapter.realAssets() causes AUM inflation leading to share mispricing and cap-based DoS	Low
19	Stale oracle index in view-based LP share price (SwapCore.getPoolSharePrice used by KairosMarketAdapter.realAssets) causes integrator-level NAV misvaluation and potential user fund loss	Low
20	Raw-balance collateral cap enforcement in SwapCore (supplyCollateral/buySwap) causes donation-based DoS of deposits and purchases	Informational

4. Detailed Findings

4.1: Missing sweep and delta-only forwarding in KairosBuyerAdapter enables third-party settlement to strand funds, causing permanent freezing of vault assets

Severity: Critical

What happens

The `KairosBuyerAdapter` uses a balance-before/after delta pattern in `allocate` and `deallocate` — it only forwards the balance change that occurs during the call. The adapter has no sweep function and no mechanism to forward pre-existing token balances. When a vault is connected, its operator calls `allocate` on the adapter, which calls `buySwap` in the core with `onBehalfOf` set to the adapter's address. When the swap expires, the expected flow is via `deallocate`, which calls `makePayment`, receives the proceeds, and forwards the delta back to the vault.

However, `makePayment` is entirely permissionless. Anyone can call it with the swap ID and settle the swap out-of-band — a bot doing housekeeping, LPs wanting to unlock their collateral, or a depositor. This transfers the tokens directly to the adapter. Now when the vault calls `deallocate` and it calls `makePayment`, the core sees the swap has already been settled and reverts. The tokens are stuck at the adapter with no on-chain recovery path.

Root cause

- `SwapCore.makePayment()` is permissionless at `SwapCore.sol:L952` — any address can settle any matured swap
- `Utils.settleSwap()` transfers proceeds to `swap.userAddress` (the adapter) at `Utils.sol:L359`
- `Utils.settleSwap()` marks `swap.settled = true` at `Utils.sol:L325` — subsequent settlement calls revert
- `KairosBuyerAdapter.deallocate()` calls `makePayment` internally at `KairosBuyerAdapter.sol:L273` — reverts when swap is already settled
- `KairosBuyerAdapter` uses balance-before/after delta pattern at `KairosBuyerAdapter.sol:L193,L216,L268,L276` — only forwards per-call changes, not pre-existing balance

- No sweep function exists on `KairosBuyerAdapter` — `IAdapter` interface defines no recovery mechanism

Impact boundary

What is possible:

- Permanent freezing of vault assets: settlement proceeds and buyer collateral stuck at the adapter with no on-chain recovery
- `realAssets()` still counts stuck funds at `KairosBuyerAdapter.sol:L345`, inflating reported AUM while tokens are inaccessible
- Multiple swaps compound the stranding — at realistic scale (10 swaps, 500k USDC), settlement proceeds accumulate at the adapter
- Both direct ERC20 donations and legitimate settlement proceeds become permanently stuck

What is not possible:

- Token theft — the funds are frozen, not extractable by any party
- Vault accounting corruption beyond AUM inflation — the adapter correctly tracks positions

Attack economics:

- Capital required: 0 (third-party settlement is rational behavior — LPs settling to unlock `pool.lockedCollateral`)
- Transactions: 1 (single `makePayment` call by any address)
- Timing: After swap expiry
- Net profit/loss to attacker: Not an attack per se — emerges from rational market behavior
- Who can trigger: Any address (permissionless). LPs are incentivized to settle promptly.

Proof of Concept results

- Third party can permissionlessly settle matured swaps via `makePayment()`, transferring proceeds to the adapter
- Once settled out-of-band, the vault's `deallocate()` path reverts with "already settled"
- Settlement proceeds permanently stuck at adapter; vault balance unchanged
- Stuck funds counted in `realAssets()` — inflating reported AUM while tokens are inaccessible
- Donation of 10,000 USDC to adapter also strands alongside settlement proceeds
- Multiple swaps compound the stranding — each out-of-band settlement adds to the unreachable balance
- Normal settlement path (vault calls `deallocate` first) works correctly — zero stranding

PoC file: `test-foundry/pocs/POC-FINDING01-ADAPTER-FUND-STRANDING.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING01-ADAPTER-FUND-STRANDING.t.sol -vvv`

4.2: Permissionless selective settlement in SwapCore.makePayment causes manipulated LP share price and fund extraction

Severity: High

What happens

`SwapCore.makePayment()` is permissionless and settles arbitrary subsets of matured swaps. Settlement caps payouts at available pool collateral, but LP share prices are computed using uncapped obligations in `calculateTotalUnrealizedPnL`. This creates a gap: a pool with 60 USDC backing a swap that owes 100 shows -100 in MtM but only realizes -60 at settlement. An attacker can deposit when MtM reflects the full -100 (getting cheap shares), trigger settlement (which only realizes -60), and withdraw at the recovered price — extracting the 40 USDC gap from existing LPs. The same principle works for withdrawals: exit the pool before settlement to avoid the loss. Liquidators are positioned to double-dip when the liquidation incentive creates a similar capped-vs-MtM gap.

Root cause

- `SwapCore.makePayment()` at `SwapCore.sol:L952` accepts arbitrary swap ID arrays, fully permissionless
- `Utils.settleSwap()` at `Utils.sol:L299-321` caps settlement amounts at available pool collateral
- `Utils.calculateBucketUnrealizedPnL()` uses uncapped obligations in MtM at `Utils.sol:L1011-1160`
- `Utils.calculateLpSharePrice()` at `Utils.sol:L482` sums uncapped MtM for share pricing
- `SwapCore.supplyCollateral()` at `SwapCore.sol:L427` and `withdrawCollateral()` at `SwapCore.sol:L316` mint/burn at live share price with no slippage parameter

Impact boundary

What is possible:

- Attacker deposits at MIN_SHARE_PRICE when MtM reflects full uncapped loss, mints ~99.9998% of shares
- After settlement (which only realizes capped loss), share price recovers and attacker withdraws at higher price
- Bidirectional: withdrawers can exit before settlement to avoid loss; depositors can enter during maximum depression
- Extends to liquidations: liquidation incentive creates additional gap that liquidators can exploit

What is not possible:

- Direct token theft outside normal deposit/withdraw flows
- Bypassing `nonReentrant` guards — requires sequential transactions (or atomic via smart contract)
- Exploitation without undercollateralized matured swaps existing

Attack economics:

- Capital required: Deposit amount (returned after withdrawal at profit)
- Transactions: 3 (supplyCollateral, makePayment, withdrawCollateral — can be atomic)
- Timing: Requires matured undercollateralized swaps
- Net profit/loss to attacker: Gap between uncapped MtM and capped settlement, proportional to attacker's share
- Who can trigger: Any address (permissionless settlement, permissionless LP operations)

Proof of Concept results

- Settlement caps at available pool collateral: 43k max on 10M notional exposure confirmed
- MtM uses full uncapped obligation: share price depressed to MIN_SHARE_PRICE while capped loss is much smaller
- Deposit-before-settle: 500k USDC deposit at MIN_SHARE_PRICE mints ~5e17 shares (~99.9998% ownership); post-settlement share price recovers
- Withdraw-before-settle: attacker withdraws at inflated price, avoiding the capped settlement loss
- Two-step arbitrage: deposit between first and second settlement to capture progressive recovery
- Attack cost accounting: 3 transactions, fully recoverable capital, gas-only cost
- Without manipulation (same-time settlement): zero arbitrage opportunity — confirms vulnerability is the ordering gap

PoC file: `test-foundry/pocs/POC-FINDING04-SELECTIVE-SETTLEMENT.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING04-SELECTIVE-SETTLEMENT.t.sol -vvv`

4.3: Non-binding early-exit preview without bounds in early-exit settlement flow causes front-runnable and drift-prone user losses

Severity: High

What happens

`previewEarlyExitSettlement()` simulates the early exit using cached oracle values and the current base rate. The actual `exitSwapEarly()` → `makePayment()` execution re-reads fresh oracle values and updates the rate index at execution time. The `projectFavorsBuyer` boolean can flip between preview and execution based on a rate update, changing settlement entirely: when true, the buyer forfeits projected gains but keeps accrued amounts; when false, the buyer pays discounted projected losses plus the early exit fee plus accrued payments. No slippage protection parameters exist on `exitSwapEarly()` — no `minOut`, `maxPay`, or `expectedDirection`. Combined with JIT liquidity attacks, an attacker could profit from sandwiching early exit transactions.

Root cause

- `previewEarlyExitSettlement()` reads at call time without updating oracle index at `SwapCore.sol:L659`
- `exitSwapEarly()` has no slippage parameters at `SwapCore.sol:L683-706` — no `minOut`, `maxPay`, or direction lock
- `makePayment()` re-reads fresh oracle values at `SwapCore.sol:L978-981`, updating the rate index before settlement
- `Utils.calculateEarlyExitSettlement()` uses `projectedRate` to determine `projectFavorsBuyer` at `Utils.sol:L451-536` — a rate change between preview and execution flips this boolean
- `KairosBuyerAdapter` early exit path at `KairosBuyerAdapter.sol:L296-333` also has no slippage protection

Impact boundary

What is possible:

- Settlement outcome flips from buyer-favorable (forfeiting projected gains) to buyer-unfavorable (paying discounted losses + fees)
- Natural oracle drift alone can cause material divergence — no attacker needed
- MEV sandwich attack: attacker manipulates oracle rates around the victim's `exitSwapEarly` transaction
- Early exit fee (1%) applied on top of unexpected losses when direction flips

What is not possible:

- Exploitation on standard `buySwap` — swaps don't settle immediately, so rate drift is absorbed over the term
- Impact when oracle rates are stable between preview and execution

Attack economics:

- Capital required: 0 for passive drift exploitation; MEV sandwich requires oracle manipulation capital
- Transactions: 1 (the victim's own `exitSwapEarly`)
- Timing: Rate change must occur between preview and execution
- Net profit/loss to attacker: MEV attacker captures the delta; passive drift affects the user directly
- Who can trigger: Any MEV searcher, or natural oracle volatility (non-adversarial)

Proof of Concept results

- Preview vs execution divergence: 5% → 3% base rate change between preview and execution produces different settlement amounts
- Direction flip: `projectedFavorsBuyer` flips from true to false on rate changes of 5% → 1%
- No slippage protection: `exitSwapEarly()` function signature has zero protective parameters
- Natural oracle drift: 5% → 4.5% movement (non-adversarial) causes measurable settlement divergence
- MEV sandwich: attacker manipulates oracle 5% → 8% (pre-tx) → 2% (post-tx), extracting value from victim
- Fee compounds loss: 1% early exit fee applied on top when direction flips to buyer-unfavorable
- Attack cost accounting: rate manipulation (5% → 8%) between preview and execution changes settlement outcome; buyer collateral as capital, MEV sandwich viable

PoC file: `test-foundry/pocs/POC-FINDING06-EARLY-EXIT-FRONTRUN.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING06-EARLY-EXIT-FRONTRUN.t.sol -vvv`

4.4: Double-counting buyer collateral in Utils.liquidateSwap buyer-liquidation path causes Token Balance Solvency invariant break

Severity: High

What happens

In the buyer-side liquidation path of `Utils.liquidateSwap`, the protocol sets `swap.collateralBalance = collateralToPool` (buyer collateral minus liquidator reward) and simultaneously increases `pool.totalCollateral` by the same `collateralToPool` amount. The same tokens are claimed by both `pool.totalCollateral` and `swap.collateralBalance` on the now-settled swap. Normal settlement correctly zeroes `swap.collateralBalance` at `Utils.sol:L349`, and pool-side liquidation correctly zeroes it at `Utils.sol:L843` — only the buyer-liquidation branch has this bug.

Root cause

- Buyer-liquidation branch at `Utils.sol:L818` sets `swap.collateralBalance = collateralToPool` instead of 0
- Same branch at `Utils.sol:L813-827` also increases `pool.totalCollateral += collateralToPool` — double-counting
- Pool-liquidation branch correctly zeroes at `Utils.sol:L843`: `swap.collateralBalance = 0`
- Normal settlement correctly zeroes at `Utils.sol:L349`: `swap.collateralBalance = 0`
- Token Balance Solvency invariant (`balanceOf >= sum(pool.totalCollateral) + sum(swap.collateralBalance)`) breaks

Impact boundary

What is possible:

- Token Balance Solvency invariant broken: `pool.totalCollateral + swap.collateralBalance` exceeds actual contract balance by `(C - R)` per liquidation, where C is buyer collateral and R is liquidator reward
- Multiple buyer-side liquidations compound: 3 liquidations at 500k each accumulate ~1.425M phantom USDC
- Off-chain solvency monitoring and analytics permanently desynchronized

What is not possible:

- Direct token extraction via the phantom balance — all state-changing paths check `swap.settled` and skip/revert
- Impact on pool-side liquidation or normal settlement — those paths zero correctly
- Accounting corruption beyond the `swap.collateralBalance` field

Attack economics:

- Capital required: None (liquidation is permissionless and occurs during normal market conditions)
- Transactions: 1 (standard `liquidateSwap` call)
- Timing: Triggered whenever buyer's accrued obligation exceeds liquidation threshold
- Net profit/loss to attacker: Not a profit-seeking attack — the invariant break occurs during normal liquidation
- Who can trigger: Any liquidator (permissionless)

Proof of Concept results

- Pool-side liquidation correctly zeroes `swap.collateralBalance` to 0

- Buyer-side liquidation bug: `swap.collateralBalance` remains non-zero (475k USDC on 500k collateral with 5% liquidation incentive)
- Token Balance Solvency invariant broken: `pool.totalCollateral + swap.collateralBalance > contractBalance` by 475k
- Multiple liquidations compound: 3 buyer-side liquidations accumulate ~1.425M phantom balance
- Normal settlement correctly zeroes collateral (control test)
- Side-by-side comparison: identical setup, buyer-liquidation leaves non-zero collateral, pool-liquidation zeroes it
- Phantom balance cannot be withdrawn — persistent off-chain metric corruption

PoC file: `test-foundry/pocs/POC-FINDING13-DOUBLE-COUNT-LIQUIDATION.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING13-DOUBLE-COUNT-LIQUIDATION.t.sol -vvv`

4.5: Time-window mismatch in bucket MtM (Utils.calculateBucketUnrealizedPnL) causes LP share price mispricing and fund extraction

Severity: Medium

What happens

In `Utils.calculateBucketUnrealizedPnL()`, `timeElapsed` is capped at `swapTerm` and used in both `deriveRateFromIndex()` (which divides by `timeElapsed` to annualize) and `calculateSwapPayment()` (which multiplies by `timeElapsed` to de-annualize). The cap cancels algebraically, producing an accrued floating payment of `notional × (oracleIndex/entryIndex − 1)` — capturing full post-expiry index drift regardless of the cap. Settlement uses uncapped `totalElapsed` for rate derivation but capped `swapDuration` for payment, correctly scaling down post-expiry drift. This inconsistency creates a predictable share price jump at settlement that can be exploited via deposit-settle-withdraw arbitrage.

Root cause

- `calculateBucketUnrealizedPnL()` caps `timeElapsed` at `swapTerm` at `Utils.sol:L1033`
- `deriveRateFromIndex()` divides by `timeElapsed` at `SwapFormulas.sol:L164-175` — annualization
- `calculateSwapPayment()` multiplies by `timeElapsed` at `SwapFormulas.sol:L186-206` — de-annualization
- Cap cancels algebraically: `(indexGrowth / cappedTime) × cappedTime = indexGrowth` — post-expiry drift fully captured
- Settlement at `Utils.sol:L240-280` uses uncapped time for rate derivation, capped for payment — correct scaling

Impact boundary

What is possible:

- Bucket MtM over-counts post-expiry losses, depressing share price for deposits
- Settlement only realizes term-proportional losses, creating predictable price recovery

- Deposit-settle-withdraw arbitrage: buy at depressed MtM price, settle to realize smaller loss, sell at recovered price
- Mispricing scales linearly with post-expiry delay (tested at 1, 3, 7 days)

What is not possible:

- Impact when settlement occurs at exact expiry — no post-expiry drift, no mismatch
- Impact pre-expiry — both paths use the same elapsed time

Attack economics:

- Capital required: Deposit amount (fully recoverable)
- Transactions: 3 (deposit, settle N swaps, withdraw)
- Timing: Requires expired-but-unsettled swaps with post-expiry oracle drift
- Net profit/loss to attacker: The MtM-vs-settlement gap, proportional to post-expiry delay and rate movement
- Who can trigger: Any address (permissionless settlement and LP operations)

Proof of Concept results

- Algebraic cancellation proof: `deriveRateFromIndex / timeElapsed × timeElapsed = indexGrowth` regardless of cap
- Share price mispricing post-expiry: bucket MtM reflects full index drift, settlement realizes only term-proportional amount
- Deposit-settle-withdraw attack: attacker deposits at depressed price, settles expired swaps, withdraws at recovered price
- BUY_FLOATING mismatch confirmed: same pattern with opposite rate direction
- Mispricing scales with delay: 1-day, 3-day, 7-day post-expiry delays produce proportionally larger gaps
- Attack cost accounting: minimal cost (1 deposit + N settlements + 1 withdrawal), fully recoverable capital
- Negative test: settlement at exact expiry produces zero mismatch

PoC file: `test-foundry/pocs/POC-FINDING10-BUCKET-TIME-MISMATCH.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING10-BUCKET-TIME-MISMATCH.t.sol -vvv`

4.6: Missing decimals normalization in Utils.calculateTotalUnrealizedPnL projected-rate handling causes LP share price mispricing and value extraction

Severity: Medium

What happens

`Utils.calculateTotalUnrealizedPnL()` calls `IBaseRateOracle.getBaseRate(remainingTenor)` to obtain projected rates for bucket MtM, but passes the returned rate directly to `calculateBucketUnrealizedPnL` without normalization. All other oracle rate paths in the codebase correctly normalize via `Utils.normalizeToWadSigned(rate,`

`decimals)`. Chainlink and Pyth default to 8 decimals, so an 8-decimal base rate oracle would produce a rate that is 1e10x smaller than expected — causing bucket P&L calculations to be 1e10x underestimated, share prices to barely change after rate movements, and liquidation thresholds to be effectively unreachable.

Root cause

- `Utils.calculateTotalUnrealizedPnL()` at `Utils.sol:L1158` calls `getBaseRate(remainingTenor)` without normalization
- `IBaseRateOracle` extends `IKairosRateOracle` which exposes `decimals()` at `External.sol:L52,L56` — the data is available but unused
- `Utils.getOracleRate()` at `Utils.sol:L643-650` correctly normalizes via `normalizeToWadSigned(rate, decimals)` — correct pattern exists
- With 8-decimal oracle: rate = 5,000,000 (representing 5%) vs WAD = 0.05e18 — a 1e10x scaling error

Impact boundary

What is possible:

- LP share prices barely change after rate movements with 8-decimal oracles — P&L calculations underestimated by 1e10x
- Deposits at effectively frozen share prices mint too many or too few shares
- Liquidation thresholds become unreachable — 1e10x too high for 8-decimal oracles
- Attackers can mint excess shares at understated prices, wait for tenor decay or settlement, then redeem at corrected prices

What is not possible:

- Impact with current 18-decimal oracles — the bug is latent until a non-18-decimal oracle is deployed
- Impact on settlement path — settlement uses `getOracleRate()` which normalizes correctly

Attack economics:

- Capital required: Deposit amount (standard LP operation)
- Transactions: Standard deposit/withdraw
- Timing: Requires non-18-decimal oracle deployment (within owner's authority)
- Net profit/loss to attacker: Excess shares worth the scaling error, bounded by pool size
- Who can trigger: Any LP, once a non-18-decimal oracle is deployed

Proof of Concept results

- 8-decimal vs 18-decimal: same 5% rate represented as 5,000,000 vs 0.05e18 — 1e10x difference confirmed
- Share price barely changes with 8-decimal oracle after rate movements — P&L calculations 1e10x too small
- Deposits at wrong price: LP2 receives shares as if price never changed despite significant rate movement
- Liquidation threshold unreachable: 1e10x scaling error makes liquidation conditions impossible to trigger

- Extreme rates (50%) produce divergent share prices between 8-decimal and 18-decimal markets
- Negative test: applying `normalizeToWadSigned(rate, oracle.decimals())` fixes the scaling entirely
- Attack cost accounting: standard LP deposit/withdraw (2 transactions) exploits 1e10x scaling mismatch when 8-decimal oracle is deployed; capital fully recoverable

PoC file: `test-foundry/pocs/POC-FINDING12-DECIMALS-NORMALIZATION.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING12-DECIMALS-NORMALIZATION.t.sol -vvv`

4.7: Per-bucket external base oracle dependency in share-price and settlement causes market DoS and frozen LP funds

Severity: Medium

What happens

`Utils.calculateTotalUnrealizedPnL()` calls `IBaseRateOracle.getBaseRate(remainingTenor)` unconditionally for all non-empty buckets with a hard `require(isValid)`. This function is called by `getPoolSharePrice()`, which is called by `supplyCollateral()`, `withdrawCollateral()`, and during settlement. Additionally, `SwapCore.makePayment()` unconditionally calls `Utils.getOracleRate(market.baseSwapRateOracle)` before settlement — even for normal post-expiry flows where the returned rate is never used (it is only consumed in the early exit path). These two independent oracle call sites mean any base oracle failure (revert or invalid response) blocks all LP operations and all settlements for affected markets, freezing LP funds indefinitely.

Root cause

- `Utils.calculateTotalUnrealizedPnL()` at `Utils.sol:L1158-1159` calls `getBaseRate(remainingTenor)` with hard `require(isValid)` — no try/catch
- `SwapCore.makePayment()` at `SwapCore.sol:L978` unconditionally calls `Utils.getOracleRate(market.baseSwapRateOracle)` — blocks normal post-expiry settlement even though the returned rate is only consumed in the early exit path
- `SwapCore.supplyCollateral()` at `SwapCore.sol:L427` and `withdrawCollateral()` at `SwapCore.sol:L316` both depend on `getPoolSharePrice()` which calls the oracle
- Moving the `getOracleRate` call into the `isEarlyExit` block would eliminate the settlement DoS while preserving early exit functionality

Impact boundary

What is possible:

- Complete market DoS: settlements, LP deposits, and LP withdrawals all blocked simultaneously
- Multiple LPs frozen: 450k+ USDC across 3 LPs frozen in test scenario with 100k unsettled notional
- Duration: indefinite until oracle recovers or governance intervenes

What is not possible:

- Cross-market contagion — the oracle failure is per-oracle, affecting only markets using that oracle
- Fund loss — the DoS is temporary, resolving when the oracle recovers
- Exploitation for profit — pure availability degradation

Attack economics:

- Capital required: 0 (external oracle failure, not attacker-triggered)
- Transactions: 0
- Timing: Oracle failure duration
- Net profit/loss to attacker: No profit — pure DoS
- Who can trigger: External event (oracle downtime, contract bug, staleness)

Proof of Concept results

- Normal settlement works with healthy oracle (baseline)
- `makePayment` reverts when base oracle reverts or returns `isValid = false`, even for normal post-expiry settlements
- LP withdrawals blocked: `getPoolSharePrice → calculateTotalUnrealizedPnL → getBaseRate` chain reverts
- LP deposits blocked when `totalShares > 0`, triggering share price calculation
- Complete market DoS: all operations (settle, deposit, withdraw) blocked simultaneously
- Root cause confirmed: projected rate only used in early exit path, never for normal settlement
- 450k USDC frozen across 3 LPs with 100k unsettled notional
- Zero capital, indefinite duration, no admin workaround
- Oracle recovery restores all operations immediately
- Conditional oracle call (only for early exit) would eliminate the DoS while preserving early exit functionality

PoC file: `test-foundry/pocs/POC-FINDING02-ORACLE-MARKET-DOS.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING02-ORACLE-MARKET-DOS.t.sol -vvv`

4.8: Share-price floor at MIN_SHARE_PRICE in calculateLpSharePrice during underwater MtM causes severe LP dilution

Severity: Medium

What happens

When pool MtM becomes deeply negative (unrealized losses >= totalCollateral), `Utils.calculateLpSharePrice()` hard-floors the share price at `MIN_SHARE_PRICE` (1e12 WAD). With a 6-decimal token like USDC, this floor amplifies share minting by 1e6x: 1 USDC mints 1,000,000 shares instead of 1 share at WAD price. An attacker deposits minimal collateral at the floor, mints an enormous share count, then captures a disproportionate fraction of any subsequent recovery — settlement inflows where buyers owe, rate reversions, or new LP deposits at recovered prices. Existing LPs are diluted nearly to zero.

Root cause

- `Utils.calculateLpSharePrice()` hard-floors at `MIN_SHARE_PRICE = 1e12` at `Utils.sol:L1195-1197`
- `SwapCore.supplyCollateral()` at `SwapCore.sol:L427-431` mints shares as `collateralAmount * WAD / sharePrice` with no underwater check
- No deposit gate, minimum deposit, or anti-dilution protection when pool is at the floor
- 1e6x amplification for 6-decimal tokens: `1e6 * 1e18 / 1e12 = 1e12` shares per USDC

Impact boundary

What is possible:

- 100 USDC deposit at floor mints shares comparable to or exceeding LP1's 1,000,000 USDC deposit
- Attacker captures majority of post-floor recovery (settlement inflows, rate reversions)
- Existing LPs diluted nearly to zero — 1M USDC principal effectively wiped

What is not possible:

- Profitable attack without subsequent pool recovery — if rates stay underwater, attacker's shares are worthless too
- Forcing the pool underwater — requires genuine market conditions (large notional, extreme rates)
- Exploitation when `lpWhitelistEnabled = true` — only whitelisted addresses can deposit

Attack economics:

- Capital required: 100 USDC or less (trivial)
- Transactions: 1-2 (deposit, optional later withdrawal)
- Timing: Pool must be at underwater floor state
- Net profit/loss to attacker: Proportional to post-deposit recovery; 100 USDC investment captures majority of recovery
- Who can trigger: Any address (when LP whitelist disabled)

Proof of Concept results

- MIN_SHARE_PRICE amplifies minting by 1e6x for 6-decimal tokens
- 10M notional, 150% rate over 25 days causes 1M pool to floor at MIN_SHARE_PRICE
- 100 USDC at floor captures disproportionate recovery value, diluting LP1's 1M USDC deposit
- 50 USDC deposit at floor yields more shares than LP1's 1M deposit
- 10 USDC at extreme floor owns significant pool fraction, captures recovery profits
- Attack profitable only when pool recovers — no profit if rates stay bad (negative test confirmed)
- 100 USDC cost, fully recoverable, 1-2 transactions, no MEV required

PoC file: `test-foundry/pocs/POC-FINDING03-SHARE-PRICE-FLOOR.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING03-SHARE-PRICE-FLOOR.t.sol -vvv`

4.9: Stale-index best-effort update in RateIndexLib.update on invalid reference oracles causes direct LP principal loss via mis-settlement and mispriced LP share changes

Severity: Medium

What happens

`RateIndexLib.update()` returns the current (stale) index without reverting when the reference oracle returns `isValid = false` or reverts — a best-effort design. Callers (`makePayment`, `withdrawCollateral`, `supplyCollateral`, `liquidateSwap`) proceed using the stale index with no validity check. The base rate oracle fails closed (hard `require`), but the reference rate oracle fails open (silent stale return). An attacker can exploit the delta: monitor for windows where the reference oracle is invalid, settle swaps during the stale window to zero out floating rate accrual, and extract value from the asymmetric failure behavior.

Root cause

- `RateIndexLib.update()` at `RateIndexLib.sol:L38-49` returns stale index on oracle failure — no revert
- `SwapFormulas.deriveRateFromIndex()` yields 0 derived rate when entry index == closing index (stale)
- `SwapCore.makePayment()` at `SwapCore.sol:L981` calls `rateIndex.update()` before settlement — stale index used
- Base oracle fails closed (hard `require`), reference oracle fails open (silent stale) — asymmetric behavior
- All state-changing paths (`makePayment`, `supplyCollateral`, `withdrawCollateral`, `liquidateSwap`) use the stale index without validity checking

Impact boundary

What is possible:

- BUY_FLOATING buyer pays zero floating rate during stale window — LP loses accrued revenue
- LP share price mispriced during stale window — unrealized P&L uses frozen index
- Liquidation checks suppressed — accrued obligations computed with stale data
- Multiple swaps settleable in single stale window — cumulative LP loss

What is not possible:

- Impact when oracle is healthy — index updates correctly
- Impact on base oracle path — it fails closed (reverts)

Attack economics:

- Capital required: 0 (permissionless settlement during stale window)
- Transactions: N (settle multiple swaps in one window)
- Timing: Monitor reference oracle for invalid windows
- Net profit/loss to attacker: Floating rate accrual saved, proportional to notional × rate × stale period
- Who can trigger: Any address that monitors oracle validity

Proof of Concept results

- `RateIndexLib.update()` silently returns stale index when oracle is invalid
- `deriveRateFromIndex` yields exactly 0 when index is stale (entry == closing)

- Asymmetric validation: base oracle fails closed (reverts), reference oracle fails open (silent)
- End-to-end: BUY_FLOATING buyer underpays LP when settling during stale window
- 50k notional at 5% annual, 18-hour stale period: ~75% of floating payment lost
- LP share price mispriced during stale window — unrealized P&L uses frozen index
- Attacker monitors oracle for invalid windows, settles during stale period — zero capital, permissionless
- Strict revert on invalid oracle would prevent attack — settlement frozen until recovery
- 3 swaps × 50k notional, 20-hour stale period: all settleable, cumulative LP loss confirmed
- Multiple swaps in single window amplify loss

PoC file: `test-foundry/pocs/POC-FINDING11-STALE-INDEX-SETTLEMENT.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING11-STALE-INDEX-SETTLEMENT.t.sol -vvv`

4.10: Direct ERC20 payouts to user-specified recipient in settlement/liquidation with blacklistable tokens cause permanent bricking and locked pool collateral

Severity: Medium

What happens

Settlement and liquidation transfer tokens directly to `swap.userAddress` via `safeTransfer`. A buyer can set `userAddress` to a blacklisted address (e.g., USDC/USDT OFAC-sanctioned address) at swap creation. When the token's transfer function reverts for that recipient, the entire `makePayment` or `liquidateSwap` call reverts — leaving `pool.lockedCollateral` permanently unreleased. Liquidation is disallowed after expiry, so a bricked settlement cannot be bypassed via the liquidation path either. This is worse than just locked collateral: the unreleased locked collateral contaminates LP share price calculations over time, mispricing all LP deposits and withdrawals. It affects all LPs in the pool, not just the parties directly involved.

Root cause

- `Utils.settleSwap()` at `Utils.sol:L358-360` performs `safeTransfer` to `swap.userAddress` — revert blocks entire settlement
- `Utils.liquidateSwap()` at `Utils.sol:L860-861` performs `safeTransfer` to `swap.userAddress` in pool-liquidation branch
- `SwapCore.buySwap()` at `SwapCore.sol:L580` sets `swap.userAddress = onBehalfOf` with no address validation
- No pull-based fallback, escrow mechanism, or try/catch around the transfer

Impact boundary

What is possible:

- Settlement and liquidation permanently bricked for affected swaps
- `pool.lockedCollateral` permanently unreleased — reduces available collateral for all LPs
- Batch contamination: blacklisted swap in `makePayment` array blocks all swaps in the batch
- LP share price contaminated: locked collateral affects `totalLPAvailableCollateral` and downstream pricing

- Mid-term blacklisting (OFAC event) affects previously legitimate swaps — timing-independent
- New LPs joining after the attack are also harmed by pre-existing locked collateral

What is not possible:

- Profit from the attack — pure griefing, attacker sacrifices their own collateral
- Cross-market contamination — locked collateral is per-market

Attack economics:

- Capital required: `minTradeSize` per swap (attacker sacrifices buyer collateral)
- Transactions: 1 per malicious swap
- Timing: Damage manifests at settlement/liquidation
- Net profit/loss to attacker: Net loss (sacrificed collateral) — pure griefing
- Who can trigger: Any buyer with a blacklisted recipient address

Proof of Concept results

- `makePayment` reverts when `swap.userAddress` is blacklisted — swap remains unsettled, collateral locked
- 5 swaps with different blacklisted recipients: locked collateral accumulates, choking the pool
- Batch contamination: single blacklisted swap blocks entire `makePayment` array
- LP share price contaminated by locked collateral — forces partial withdrawals
- Mid-term blacklisting: previously legitimate buyer gets OFAC-blacklisted, settlement bricked at expiry
- Attack cost: buyer collateral per swap; locked LP collateral is 0.5x+ attacker cost; one-time cost, permanent damage
- Systemic impact: new LP2 joining after attack still harmed by pre-existing locked collateral
- Normal (non-blacklisted) settlement succeeds — push-based transfer is the root cause
- Both `settleSwap` (Utils.sol:L359) and `liquidateSwap` (Utils.sol:L861) use `safeTransfer` — both bricked

PoC file: `test-foundry/pocs/POC-FINDING17-BLACKLIST-TOKEN-GRIEFING.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING17-BLACKLIST-TOKEN-GRIEFING.t.sol -vvv`

4.11: Inconsistent projected rate source (spot vs tenor) in early-exit settlement vs MtM causes JIT LP share-price extraction/dilution

Severity: Medium

What happens

Early-exit settlement prices remaining payments using the spot base rate (`IKairosRateOracle.getRate()`), while LP share-price MtM uses tenor-specific rates (`IBaseRateOracle.getBaseRate(remainingTenor)`). When yield curves are non-flat, the MtM share price (tenor-based) diverges from the realized settlement value (spot-based), creating a predictable share price jump at settlement. JIT LPs can deposit before an early-exit settlement (at the tenor-based MtM

price) and withdraw after (at the spot-based settlement price), capturing the rate differential from passive LPs.

Root cause

- `SwapCore.makePayment()` at `SwapCore.sol:L978` uses `Utils.getOracleRate()` (spot) for early-exit settlement
- `Utils.calculateTotalUnrealizedPnL()` at `Utils.sol:L1158` uses `IBaseRateOracle.getBaseRate(remainingTenor)` for MtM
- With non-flat yield curve: tenor rate != spot rate → MtM price != settlement price
- `supplyCollateral` and `withdrawCollateral` use spot share price with no TWAP, cooldown, or epoching

Impact boundary

What is possible:

- JIT LPs deposit before early-exit settlement, withdraw after — capture share-price jump
- Transfers value from passive LPs to JIT LPs (dilution)
- Scales with yield curve slope and notional size
- Timing-dependent: requires observable early-exit events or coordination

What is not possible:

- Impact when yield curves are flat — spot equals tenor, no divergence
- Pool insolvency — token balances remain correct, only LP value redistribution
- In current production: oracles that don't implement `IBaseRateOracle.getBaseRate()` would revert on the tenor path

Attack economics:

- Capital required: JIT deposit amount (fully recoverable)
- Transactions: 3 (deposit, early-exit settlement, withdraw)
- Timing: Must coincide with early-exit event
- Net profit/loss to attacker: Proportional to curve slope × notional × attacker share
- Who can trigger: Any LP (when whitelist disabled)

Proof of Concept results

- Tenor curve configured: spot 3% vs 1-day tenor 5% (2% premium)
- MtM uses tenor-based rates while settlement uses spot — divergence confirmed
- JIT extraction with upward-sloping curve: MtM overestimates obligation, settlement at lower spot increases pool value
- Explicit code path verification: MtM reads `getBaseRate(tenor)`, settlement reads `getOracleRate()` (spot)
- Multi-bucket scenario: 2 swaps across buckets show aggregate mismatch from multiple tenors
- Quantified extraction: measured as percentage of JIT deposit, scales with curve slope
- Attack cost accounting: JIT LP deposit before early exit on steep yield curve (2% spot, 8% tenor), 3 transactions, profit proportional to curve slope × notional × attacker share

PoC file: `test-foundry/pocs/POC-FINDING20-INCONSISTENT-PROJECTED-RATE.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING20-INCONSISTENT-PROJECTED-RATE.t.sol -vvv`

4.12: Same-transaction LP deposit/withdrawal manipulating availableCollateral in SwapCore.buySwap causes permanent LP fee loss

Severity: Low

What happens

A buyer can in one transaction: (1) supply temporary LP collateral via `supplyCollateral` to inflate `availableCollateral`, (2) call `buySwap` which locks in a lower utilization fee based on the inflated snapshot, and (3) withdraw the temporary deposit via `withdrawCollateral` in the same block. The utilization fee is computed from an instantaneous free-collateral snapshot and baked into the swap for its entire term. The attack becomes most relevant when crossing the kink: pushing utilization from above the kink (quadratic premium region) to below it eliminates the quadratic component entirely — approximately 0.9% at 90% utilization versus 5.9% at 100%.

Root cause

- `SwapCore.buySwap()` at `SwapCore.sol:L490` reads `availableCollateral = totalLPAvailableCollateral(marketId)` as instant snapshot
- `totalLPAvailableCollateral()` at `SwapCore.sol:L860` computes `pool.totalCollateral - pool.lockedCollateral`
- `SwapFormulas.calculateUtilizationFee()` at `SwapFormulas.sol:L258` uses instant `availableLiquidity`
- `SwapPosition.utilFee` baked into swap at `Types.sol:L93` — persists for entire term
- `supplyCollateral`, `buySwap`, `withdrawCollateral` all `nonReentrant` but sequential calls in one tx allowed

Impact boundary

What is possible:

- Crossing from 95% to 89% utilization saves ~3k on a 90-day 1M notional swap
- Crossing the kink eliminates the quadratic fee component (0.9% at 90% vs 5.9% at 100%)
- Multiple swaps benefit from a single JIT deposit, compounding fee reduction
- Fee loss persists for the entire swap term — cannot be recovered by LPs

What is not possible:

- Exploitation when `lpWhitelistEnabled = true` — JIT deposit blocked
- Flash loan cost (if used) may offset savings on small swaps

Attack economics:

- Capital required: 0 if using flash loan; deposit amount otherwise (fully returned same block)
- Transactions: 3 in one tx (supply, buy, withdraw)

- Timing: Immediate — no waiting period
- Net profit/loss to attacker: Fee savings for swap lifetime; ~3k on 1M notional 90-day swap at kink crossing
- Who can trigger: Any buyer (when LP whitelist disabled)

Proof of Concept results

- Transient deposit reduces utilization fraction
- Full JIT cycle (deposit → buySwap → withdraw) completes in one block with ~1-unit rounding tolerance
- Fee comparison: identical notional swaps have lower utilization fees with JIT manipulation
- Kink crossing: JIT deposit pushing below 70% kink eliminates quadratic premium entirely
- LP revenue loss: significant over 90-day term (exact amount varies by scenario)
- Attacker's JIT deposit fully recovered same block; buyer fee savings persist for entire swap term
- Multiple swaps: 3 swaps benefit from single JIT deposit, compounding fee reduction
- LP whitelist blocks non-whitelisted JIT deposits (mitigation path)
- Without JIT: identical utilization produces identical fees (control test)

PoC file: `test-foundry/pocs/POC-FINDING05-JIT-LP-FEE-LOSS.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING05-JIT-LP-FEE-LOSS.t.sol -vvv`

4.13: Post-expiry floating rate derivation window mismatch in normal settlement causes LP principal loss

Severity: Low

What happens

In normal post-expiry settlement, the floating rate is derived using index growth from entry to now (uncapped), but the floating payment is calculated over duration = `min(now - entry, swapTerm)` (capped at term). This allows post-expiry index movement to be baked into the derived rate even though accrual should stop at expiry. Settlement is permissionless with no deadline or keeper incentive, so an attacker can delay settlement and time it when post-expiry rates move favorably — inflating floating payment for BUY_FIXED swaps when rates rise, or reducing obligations for BUY_FLOATING when rates fall.

Root cause

- `Utils.settleSwap()` at `Utils.sol:L203` uses `timeElapsedSinceSwapOpened = block.timestamp - swap.entryTimestamp` (uncapped) for rate derivation
- Payment calculation at `Utils.sol:L240-280` uses `swapDuration = min(timeElapsed, swapTerm)` (capped) for de-annualization
- `SwapCore.makePayment()` updates rate index to current timestamp via `rateIndex.update()` before settlement — capturing post-expiry drift
- No settlement deadline, keeper incentive, or freshness guard

Impact boundary

What is possible:

- BUY_FIXED: delayed settlement when post-expiry rates rise inflates floating payment — LP pays more
- BUY_FLOATING: delayed settlement when post-expiry rates fall reduces buyer obligation — LP receives less
- Impact scales linearly with post-expiry delay duration and rate movement magnitude
- On 1M notional with 5% → 12% rate spike over 15 days post-expiry, measurable LP dollar loss

What is not possible:

- Impact when settlement occurs at exact expiry — derivation and payment windows are identical
- Impact pre-expiry — both windows equal elapsed time
- Impact without post-expiry rate movement

Attack economics:

- Capital required: 0 (permissionless settlement timing, no capital needed)
- Transactions: 1 (`makePayment` at chosen time)
- Timing: Patience — wait for favorable post-expiry rate movement
- Net profit/loss to attacker: Rate-proportional, bounded by collateral caps
- Who can trigger: Any address (permissionless settlement)

Proof of Concept results

- Rate derivation window extends past expiry: entry → now includes post-expiry period, confirmed
- BUY_FIXED delayed settlement with rate rise: LP pays more than term-only accrual
- BUY_FLOATING delayed settlement with rate fall: LP receives less than term-only accrual
- 1M notional, 5% → 12% rate spike over 15 days: quantified LP dollar loss
- Math proof: `deriveRateFromIndex` uses 45-day window while payment uses 30-day term
- Selective settlement: attacker settles swaps at favorable post-expiry times, progressively worsening LP outcomes
- Zero capital, patience only — permissionless settlement timing
- Settlement at exact expiry: correct windows, zero mismatch (negative test)
- Pre-expiry: no vulnerability — both windows equal elapsed time (negative test)

PoC file: `test-foundry/pocs/POC-FINDING09-POSTEXPIRY-RATE-WINDOW.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING09-POSTEXPIRY-RATE-WINDOW.t.sol -vvv`

4.14: Same-block index sample locking in RateIndexLib.update causes LP withdrawal overpayment (principal loss to remaining LPs)

Severity: Low

What happens

`RateIndexLib.update()` uses an elapsed time check: if `elapsed == 0` (same block), it returns the cached index without re-sampling the oracle. The index is shared by oracle across markets — a single

`updateMarketRateIndex()` call on any market locks the sample for all markets using that oracle. `updateMarketRateIndex()` is permissionless, so an attacker can front-run an oracle update by calling it first, locking the old rate sample for the entire block. All subsequent in-block calls see `elapsed == 0` and cannot access the fresher oracle data. An LP can withdraw at a share price based on the stale (favorable) index.

Root cause

- `RateIndexLib.update()` at `RateIndexLib.sol:L28-58`: `if (elapsed == 0) return cached_index`
- `lastUpdates[oracle]` set to `block.timestamp` on first update at `RateIndexLib.sol:L56`
- `SwapCore.updateMarketRateIndex()` at `SwapCore.sol:L834-837` is permissionless
- LP operations call `rateIndex.update()` before pricing — locked into the first-in-block sample

Impact boundary

What is possible:

- Attacker locks stale oracle sample, LPs deposit/withdraw at mispriced shares
- Liquidation deferred by one block
- Profitable only on dormant markets with hours-stale indices; active markets have sub-dollar extractable value

What is not possible:

- Multi-block exploitation — each block can be independently re-sampled
- Impact on active markets with frequent oracle updates

Attack economics:

- Capital required: LP share balance for withdrawal; MEV ordering capability on target L2
- Transactions: 2 (updateMarketRateIndex + withdraw, same block)
- Timing: Requires oracle rate change within the block
- Net profit/loss to attacker: Small — realistic magnitude is sub-dollar on active markets
- Who can trigger: MEV-aware participant with block ordering control

Proof of Concept results

- Index locked in same block after first `updateMarketRateIndex()`
- `updateMarketRateIndex()` is permissionless
- LP withdrawal at stale price: attacker withdraws more than fair value
- Front-run deposit: depositor overcharged when stale index understates float accrual
- Quantified advantage: measurable but small on typical market parameters
- Fresh index (next block) eliminates advantage — negative test confirmed

PoC file: `test-foundry/pocs/POC-FINDING07-SAME-BLOCK-INDEX.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING07-SAME-BLOCK-INDEX.t.sol -vvv`

4.15: Immediate bucket P&L inclusion of new swaps without min-shares-out in LP minting causes over-minting and LP principal loss

Severity: Low

What happens

When `buySwap()` opens a swap, `Utils.addSwapToBucket()` immediately updates bucket aggregates. The newly added swap contributes its full `timeRemaining = swapTerm` of projected P&L to the share price instantly. An attacker can open a large swap that depresses the share price (via unfavorable projected P&L), deposit at the depressed price (over-minting shares due to no `minSharesOut` parameter), then exit the swap early in the next block when early-exit fee is minimal. This restores the share price and the attacker can withdraw at a profit, diluting existing LPs.

Root cause

- `SwapCore.buySwap()` at `SwapCore.sol:L598` immediately calls `addSwapToBucket()` — no ramp-in period
- `Utils.calculateBucketUnrealizedPnL()` at `Utils.sol:L1011-1160` includes projected remaining for all swaps
- `SwapCore.supplyCollateral()` at `SwapCore.sol:L386-447` has no `minSharesOut` parameter
- Early exit fee at `SwapCore.sol:L692-706` is minimal for 1-block duration (fee based on accrued, not notional)

Impact boundary

What is possible:

- Share price depression by opening large swaps with unfavorable projected P&L
- Over-minting at depressed price; early exit to restore price; withdraw at profit
- Only viable in turbulent and/or low fee markets — fees keep it in check otherwise

What is not possible:

- Profitable attack when fees exceed the projected P&L impact
- Impact without favorable oracle rate curve configuration

Attack economics:

- Capital required: Collateral for the manipulative swap + deposit amount
- Transactions: 4 (buySwap, supply, exitSwapEarly, withdraw)
- Timing: Requires favorable rate curve; must cross oracle rate vs fixed rate boundary
- Net profit/loss to attacker: Marginal — early exit fee and swap fees often exceed the extraction
- Who can trigger: Any buyer/LP

Proof of Concept results

- New swap immediately changes share price (before/after comparison)
- Over-minting at depressed price: depositor receives more shares than fair value

- No `minSharesOut` protection — `supplyCollateral` has no parameter
- Complete attack flow: open swap → deposit → exit swap → withdraw — price restores, attacker profits
- Impact scales with swap size relative to pool
- Negative test: ramp-in mechanism (gradual P&L inclusion) would eliminate the vulnerability

PoC file: `test-foundry/pocs/POC-FINDING08-IMMEDIATE-BUCKET-PNL.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING08-IMMEDIATE-BUCKET-PNL.t.sol -vvv`

4.16: Timestamp-based bucket assignment with tenor-dependent per-bucket MtM in LP share pricing enables JIT manipulation causing LP principal loss

Severity: Low

What happens

Bucket assignment is deterministic via `getBucketId(block.timestamp)`. Each bucket's unrealized P&L uses a tenor-dependent projected rate from `getBaseRate(remainingTenor)`, where `remainingTenor = swapTerm - (now - avgEntryTime)`. A large swap landing in a specific bucket shifts that bucket's `avgEntryTime` and thus its `remainingTenor`, changing the projected rate queried. Total unrealized P&L is the non-linear sum of per-bucket P&Ls with different projected rates. An attacker can time a swap to distort a specific bucket's share of total P&L, deposit at the distorted share price, exit the swap, and withdraw at the corrected price.

Root cause

- `Utils.getBucketId(timestamp)` at `Utils.sol:L893` — deterministic bucket assignment
- `Utils.calculateTotalUnrealizedPnL()` at `Utils.sol:L1133-1162` — each bucket calls `getBaseRate(remainingTenor)` independently
- `SwapCore.supplyCollateral/withdrawCollateral` — spot share pricing with no TWAP, cooldown, or epoching
- Large swap shifts bucket `avgEntryTime` → changes `remainingTenor` → changes projected rate

Impact boundary

What is possible:

- Transient share price distortion by targeting specific buckets
- JIT deposit/withdraw to capture distortion — less profitable than immediate bucket P&L variant (4.15)
- Can be combined with 4.15 for compounded effect

What is not possible:

- Profitable attack with flat rate oracle — requires non-flat tenor curve
- Impact when LP whitelist restricts deposits

Attack economics:

- Capital required: Swap collateral + deposit amount
- Transactions: 4 (buySwap, supply, exitSwapEarly, withdraw)
- Timing: Bucket-specific; attacker controls entry timestamp on L2 with centralized sequencer
- Net profit/loss to attacker: Less profitable than 4.15; depends on curve steepness and pool size
- Who can trigger: Any buyer/LP on L2

Proof of Concept results

- Bucket assignment is deterministic via timestamp
- Tenor-dependent rate curve: shorter tenor = different rate
- Large swap shifts bucket weighted-average entry time and remaining tenor
- Bucket targeting changes share price: transient distortion from manipulating single bucket's P&L
- JIT mint attack: deposit at distorted price, exit swap, withdraw at corrected price
- Negative test: flat rate oracle produces zero manipulation opportunity

PoC file: `test-foundry/pocs/POC-FINDING14-JIT-BUCKET-MANIPULATION.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING14-JIT-BUCKET-MANIPULATION.t.sol -vvv`

4.17: Lack of on-chain slippage guard and non-binding rate preview in SwapCore.buySwap causes buyer rate manipulation and increased payments

Severity: Low

What happens

`Views.calculateSwapRate` provides a non-binding preview; `SwapCore.buySwap()` re-computes all dynamic inputs at execution time — oracle rates, utilization fee, risk premium. The swap's rate parameters are locked in at execution and persist for the entire term with no on-chain slippage guard. `KairosBuyerAdapter` provides an optional `maxSwapRate` parameter, but `SwapCore.buySwap()` itself has no protection. An LP withdrawal between preview and execution can inflate the utilization fee; an oracle update can shift the base rate.

Root cause

- `SwapCore.buySwap()` at `SwapCore.sol:L410-475` re-computes oracle rates, utilization fees, risk premium at execution
- No `maxSwapRate` parameter in `buySwap()` function signature
- `KairosBuyerAdapter.sol:L177-209` already implements `maxSwapRate` check when provided — pattern exists but not enforced at core

Impact boundary

What is possible:

- Buyer locked into unfavorable rate for entire swap term
- LP front-run: withdraw before victim's buy, inflate utilization fee, re-deposit after
- Oracle reorder: block builder shifts rate between preview and execution

- Impact magnitude: ~$2,466 additional cost on 1M notional 90-day swap per 1% rate drift

What is not possible:

- Swap entry slippage/sandwich attacks are mitigated by the protocol's time-delayed settlement mechanism.

Attack economics:

- Capital required: LP front-run variant requires existing LP position
- Transactions: Standard trade
- Timing: Rate change between preview and execution
- Net profit/loss to attacker: Marginal — LP front-run requires capital lock-up
- Who can trigger: LP front-runner or block builder

Proof of Concept results

- Baseline swap rate confirmed at oracle rate + utilization fee + risk premium
- Oracle rate drift: +1% rate change produces measurably worse swap rate locked for entire term
- Utilization increase: LP withdrawal reduces available collateral, inflates utilization fee
- Combined scenario: oracle drift + utilization increase compound the rate degradation
- No slippage guard in `buySwap()` interface
- Impact magnitude over 90 days: 1M × 1% × (90/365) ≈ $2,466 additional cost per 1% drift
- Attack cost accounting: frontrunner manipulates utilization or oracle rate before victim's buySwap; ~$2,466 cost to victim per 1% slippage on 1M 90-day swap

PoC file: `test-foundry/pocs/POC-FINDING15-BUYSWAP-SLIPPAGE.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING15-BUYSWAP-SLIPPAGE.t.sol -vvv`

4.18: Externally manipulable token balance in KairosBuyerAdapter.realAssets() causes AUM inflation leading to share mispricing and cap-based DoS

Severity: Low

What happens

`KairosBuyerAdapter.realAssets()` includes the adapter's raw ERC20 balance (`asset.balanceOf(address(this))`). Any third party can transfer tokens to inflate this value. The adapter lacks a sweep mechanism and only forwards per-call deltas — donated tokens are permanently stranded. If a Morpho Vault V2 uses `realAssets()` for cap enforcement or ERC4626 share pricing, the inflation causes cap-based DoS (blocking further allocations) and share mispricing (new depositors receive fewer shares).

Root cause

- `KairosBuyerAdapter.realAssets()` at `KairosBuyerAdapter.sol:L346` includes `asset.balanceOf(address(this))`

- `KairosMarketAdapter.realAssets()` at `KairosMarketAdapter.sol:L249-261` does NOT include `balanceOf` — inconsistent design
- No sweep function on adapter — donated tokens permanently stranded
- `allocate/deallocate` use balance-before/after delta pattern — only forward per-call changes

Impact boundary

What is possible:

- AUM inflation: donated tokens inflate reported `realAssets()` while being inaccessible
- Cap-based DoS: further allocations blocked if vault uses `realAssets()` for cap enforcement
- Share mispricing: new depositors receive fewer shares due to overstated total assets

What is not possible:

- Profitable attack — pure griefing (attacker loses donated tokens)
- Impact without Morpho Vault V2 integration — standalone adapter operation unaffected

Attack economics:

- Capital required: Donation amount (lost)
- Transactions: 1 (token transfer)
- Timing: Immediate
- Net profit/loss to attacker: Net loss — pure griefing
- Who can trigger: Any address

Proof of Concept results

- Baseline `realAssets()` returns expected value
- Donation inflates `realAssets()` by exact donated amount
- No sweep mechanism: donated tokens cannot be recovered
- Cap-based DoS: donations block further allocations when vault has finite cap
- Share price misvaluation: inflated AUM causes depositors to receive fewer shares
- `realAssets()` includes raw balance
- Attack cost accounting: single ERC20 transfer inflates AUM by exact donation amount (1:1 ratio); donated tokens permanently stranded, pure griefing

PoC file: `test-foundry/pocs/POC-FINDING16-ADAPTER-AUM-INFLATION.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING16-ADAPTER-AUM-INFLATION.t.sol -vvv`

4.19: Stale oracle index in view-based LP share price (SwapCore.getPoolSharePrice used by KairosMarketAdapter.realAssets) causes integrator-level NAV misvaluation and potential user fund loss

Severity: Low

What happens

`KairosMarketAdapter.realAssets()` calls `SwapCore.getPoolSharePrice()`, which is a view function that reads the stored oracle index without calling `rateIndex.update()`. The stored index can be stale — the latest time slice from `lastUpdate` to `now` is omitted from the floating leg while the fixed leg accrues to `block.timestamp`. State-changing paths (`supplyCollateral`, `withdrawCollateral`, `makePayment`) correctly call `rateIndex.update()` first, so core protocol operations are unaffected. The vulnerability is isolated to integrators (Morpho Vault V2) that rely on the view-based `realAssets()` for ERC4626 total assets.

Root cause

- `SwapCore.getPoolSharePrice()` at `SwapCore.sol:L847–851` reads stored index without calling `update()`
- `KairosMarketAdapter.realAssets()` at `KairosMarketAdapter.sol:L249–261` calls `getPoolSharePrice()`
- State-changing paths correctly update: `supplyCollateral` (L414), `withdrawCollateral` (L314), `makePayment` (L981), `liquidateSwap` (L798)
- Stale window: from `lastUpdate` to `block.timestamp`, floating accrual is omitted

Impact boundary

What is possible:

- Integrator-level NAV misvaluation during stale windows
- Vault depositors receive too many shares at understated NAV; vault redeemers extract excess at overstated NAV
- Rate shock amplifies divergence between view and state-changing paths

What is not possible:

- Impact on core protocol operations — state-changing paths update first
- Impact without Morpho Vault V2 integration

Attack economics:

- Capital required: Vault deposit/redeem amount
- Transactions: Standard vault operation
- Timing: Requires stale oracle index window + significant rate movement
- Net profit/loss to attacker: Proportional to staleness × rate change × attacker share
- Who can trigger: Any vault participant during stale window

Proof of Concept results

- Stale index causes share price mismatch: view returns stale, state change returns fresh
- Adapter NAV diverges with stale index
- Floating accrual omitted in stale index: latest time slice missing from view-based pricing
- Rate shock amplifies mismatch — larger rate changes produce larger divergence
- Index fresh after state change produces correct pricing
- Integrator-level staleness impact: view functions return stale values while core protocol is protected by state-changing index updates; adapter NAV diverges from true value during staleness windows

PoC file: `test-foundry/pocs/POC-FINDING19-STALE-ADAPTER-NAV.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING19-STALE-ADAPTER-NAV.t.sol -vvv`

4.20: Raw-balance collateral cap enforcement in SwapCore (supplyCollateral/buySwap) causes donation-based DoS of deposits and purchases

Severity: Informational

What happens

SwapCore enforces the global collateral cap using `IERC20.balanceOf(address(this)) + incoming > cap`. The check uses raw ERC20 balance, not internal accounting. Any attacker can donate tokens to SwapCore to inflate `balanceOf` and trigger E409 reverts on legitimate deposits and swap purchases. No sweep function exists to recover donated tokens. The cap defaults to `type(uint256).max` (effectively disabled), so exploitation requires admin to explicitly set a finite cap. An attacker could alternatively use `buySwap` instead of donating (and potentially profit), though donated tokens are persistent while a swap eventually settles.

Root cause

- `SwapCore.supplyCollateral()` at `SwapCore.sol:L409-411` uses `balanceOf(address(this))` for cap check
- `SwapCore.buySwap()` at `SwapCore.sol:L567-569` uses `balanceOf(address(this))` for cap check (before protocol fee transfer)
- `Admin.GLOBAL_maxCollateralBalance` defaults to `type(uint256).max` at `Admin.sol:L11`
- No sweep or recovery function in SwapCore

Impact boundary

What is possible:

- Cap check fails for subsequent LP deposits and buyer swaps when finite cap is set
- Fresh markets with zero natural outflows can be completely bricked by single donation
- DoS persists until natural outflows or admin raises cap

What is not possible:

- Impact without admin-configured finite cap — default `type(uint256).max` makes the check trivially pass
- Fund theft — pure availability degradation
- Persistent impact — admin can raise cap; natural outflows (settlements) reduce balance

Attack economics:

- Capital required: Donation amount (lost) — must be significant relative to remaining cap space
- Transactions: 1 (token transfer)
- Timing: Must target markets with finite cap
- Net profit/loss to attacker: Net loss — pure griefing

- Who can trigger: Any address (when finite cap is configured)

Proof of Concept results

- Donation inflates raw balance
- Donation blocks `supplyCollateral` with E409 when finite cap is set
- Donation blocks `buySwap` with E409
- Fresh market bricked by single donation: no outflows to reduce balance
- Donated tokens cannot be recovered — no sweep function
- Intermittent DoS with small donations: periodic top-ups maintain the block
- Negative test: no donation, all operations succeed at near-cap
- Negative test: no cap (default `type(uint256).max`), donation has no effect

PoC file: `test-foundry/pocs/POC-FINDING18-DONATION-CAP-DOS.t.sol`
Run: `forge test --match-path test-foundry/pocs/POC-FINDING18-DONATION-CAP-DOS.t.sol -vvv`

---

5. Vulnerability Classes

The 20 findings map to 9 structural vulnerability classes. These are recurring patterns, not isolated bugs — addressing the class eliminates the vulnerability for future code paths sharing the same structure.

5.1 Adapter and External Balance Accounting

Findings: 4.1, 4.18, 4.20

The KairosBuyerAdapter uses a delta-only forwarding pattern with no sweep mechanism — tokens arriving outside the allocate/deallocate flow (via permissionless settlement, donations, or other transfers) are permanently stranded. Both the adapter and SwapCore also rely on raw `ERC20.balanceOf(address(this))` for accounting-critical computations: the adapter's `realAssets()` for AUM/NAV reporting, and SwapCore's cap enforcement in `supplyCollateral`/`buySwap`. Both are inflatable by external token donations.

Recommended mitigation: Add a sweep function to the adapter or use pull-based token accounting. Use internal accounting variables instead of raw balance for cap enforcement and AUM reporting.

5.2 Transaction-Ordering Share Price Manipulation

Findings: 4.2, 4.12, 4.15, 4.16

LP share prices are computed from instantaneous state with no TWAP, cooldown, or epoching. Permissionless operations (settlement, LP deposit/withdraw, swap open) can be sequenced within a block or transaction to manipulate the state that feeds share pricing. The pattern recurs across settlement ordering, JIT liquidity, immediate bucket P&L, and deterministic bucket targeting.

Recommended mitigation: Add `minSharesOut` / `minAmountOut` slippage parameters to `supplyCollateral` and `withdrawCollateral`. Consider TWAP-based share pricing or epoch-based settlement for deposits and withdrawals.

5.3 Accounting Path Inconsistency

Findings: 4.4

Three settlement paths (normal, pool-liquidation, buyer-liquidation) should all zero `swap.collateralBalance` after crediting the pool. Two paths do this correctly; one does not. The inconsistency is a single missing assignment — not a design disagreement.

Recommended mitigation: Set `swap.collateralBalance = 0` in the buyer-liquidation branch, matching the pool-liquidation and normal settlement paths.

5.4 Non-Binding Preview / Missing Slippage Guard

Findings: 4.3, 4.17

Preview functions read cached state while execution functions re-compute with fresh inputs. No slippage protection parameters exist on the core contract entry points. The `KairosBuyerAdapter` implements `maxSwapRate` for buySwap, demonstrating the architectural intent — but `exitSwapEarly` has no equivalent protection anywhere, and `SwapCore.buySwap` lacks it at the core level.

Recommended mitigation: Add slippage parameters to `exitSwapEarly` (min payout / max payment / expected direction). Enforce `maxSwapRate` at the `SwapCore.buySwap` level.

5.5 Algebraic Time-Window Mismatch

Findings: 4.5, 4.13

Rate derivation and payment calculation use inconsistent time windows in two contexts: bucket MtM (cap cancellation produces full post-expiry drift) and settlement (uncapped derivation with capped payment). The first is a mathematical identity that makes the time cap ineffective; the second is a window mismatch that captures post-expiry drift in the derived rate.

Recommended mitigation: Use the closing index at expiry (not at current time) for expired swaps in both bucket MtM and settlement rate derivation.

5.6 Oracle Integration Gaps

Findings: 4.6, 4.7, 4.9, 4.14, 4.19

Oracle integration exhibits five distinct gap patterns across the codebase. Decimal normalization is missing for one code path — `getBaseRate(remainingTenor)` in `calculateTotalUnrealizedPnL` skips the `normalizeToWadSigned` call that all other oracle paths use. The base oracle call is unconditional even when the returned value isn't needed, blocking settlement and liquidation on oracle failure. The reference oracle fails open via try/catch, returning stale index data during invalid oracle windows — enabling mis-settlement and mispriced share changes. The `elapsed == 0` short-circuit in `RateIndexLib.update()` combined with permissionless `updateMarketRateIndex()` allows MEV-aware participants to lock a stale index for the block. The view-based share price path used by `KairosMarketAdapter.realAssets()` reads the cached oracle index without refreshing it, producing stale NAV valuations for integrators.

Recommended mitigation: Apply decimal normalization consistently to all oracle return values. Make the `getBaseRate` call conditional on whether the value is needed. Make the reference oracle fail closed for

state-changing operations or add index staleness validation. Allow oracle re-sampling when the value has changed. Add an index-refresh call to the view path or document the staleness risk for integrators.

5.7 LP Dilution via Share Price Floor

Findings: 4.8

The MIN_SHARE_PRICE floor enables rescue deposits but provides no guardrails against opportunistic exploitation. The floor creates a 1e6x minting amplification for 6-decimal tokens with no deposit gate, minimum deposit size, or anti-dilution protection.

Recommended mitigation: Consider blocking deposits when pool is at the floor, requiring governance approval for rescue deposits, or implementing pro-rata dilution protection.

5.8 Blacklistable Token Interaction

Findings: 4.10

Push-based `safeTransfer` to `swap.userAddress` has no fallback for reverting transfers. This blocks settlement and liquidation for the affected swap, permanently locking pool collateral and contaminating LP P&L calculations.

Recommended mitigation: Use a pull-based withdrawal pattern or wrap the transfer in try/catch with escrow fallback.

5.9 Inconsistent Rate Source

Findings: 4.11

Early-exit settlement and MtM use different rate sources (spot vs tenor) for the same remaining-payment calculation. When yield curves are non-flat, this creates predictable share price jumps at settlement.

Recommended mitigation: Use the same rate source (either spot or tenor) for both MtM and early-exit settlement of the remaining payment.

---

6. Hardening Recommendations

These are not vulnerabilities. They are proactive improvements that would reduce blast radius if operational assumptions fail.

6.1: Add minSharesOut / minAmountOut Parameters to LP Operations

`supplyCollateral` and `withdrawCollateral` mint/burn shares at the spot share price with no slippage protection. Adding `minSharesOut` to `supplyCollateral` and `minAmountOut` to `withdrawCollateral` would protect LPs against transaction-ordering manipulation across multiple vulnerability classes (4.2, 4.5, 4.11, 4.12, 4.14, 4.15, 4.16). This is a low-effort, high-value change that mitigates the entire transaction-ordering class.

6.2: Implement Pull-Based Settlement Payouts

All settlement and liquidation paths use push-based `safeTransfer` to `swap.userAddress`. Converting to a pull-based withdrawal pattern (credit an internal balance, let the recipient withdraw) would eliminate the blacklist griefing vector (4.10) and provide a general-purpose fallback for any reverting transfer, including future token-level changes.

6.3: Add Sweep Mechanisms to Adapters

Both `KairosBuyerAdapter` and `KairosMarketAdapter` can accumulate stranded tokens with no recovery mechanism. Adding an owner-callable sweep function for tokens not accounted for in active positions would prevent the permanent stranding described in 4.1 and 4.18.

---

7. Appendix

A: Proof of Concept File Index

PoC files are delivered separately. Place them in `test-foundry/pocs/` alongside the existing Foundry test suite. Run from `packages/hardhat`:

Finding	File	Tests
4.1	`POC-FINDING01-ADAPTER-FUND-STRANDING.t.sol`	7/7
4.2	`POC-FINDING04-SELECTIVE-SETTLEMENT.t.sol`	7/7
4.3	`POC-FINDING06-EARLY-EXIT-FRONTRUN.t.sol`	7/7
4.4	`POC-FINDING13-DOUBLE-COUNT-LIQUIDATION.t.sol`	6/6
4.5	`POC-FINDING10-BUCKET-TIME-MISMATCH.t.sol`	7/7
4.6	`POC-FINDING12-DECIMALS-NORMALIZATION.t.sol`	7/7
4.7	`POC-FINDING02-ORACLE-MARKET-DOS.t.sol`	11/11
4.8	`POC-FINDING03-SHARE-PRICE-FLOOR.t.sol`	7/7
4.9	`POC-FINDING11-STALE-INDEX-SETTLEMENT.t.sol`	10/10
4.10	`POC-FINDING17-BLACKLIST-TOKEN-GRIEFING.t.sol`	9/9
4.11	`POC-FINDING20-INCONSISTENT-PROJECTED-RATE.t.sol`	7/7
4.12	`POC-FINDING05-JIT-LP-FEE-LOSS.t.sol`	9/9
4.13	`POC-FINDING09-POSTEXPIRY-RATE-WINDOW.t.sol`	9/9
4.14	`POC-FINDING07-SAME-BLOCK-INDEX.t.sol`	7/7
4.15	`POC-FINDING08-IMMEDIATE-BUCKET-PNL.t.sol`	7/7
4.16	`POC-FINDING14-JIT-BUCKET-MANIPULATION.t.sol`	7/7
4.17	`POC-FINDING15-BUYSWAP-SLIPPAGE.t.sol`	7/7
4.18	`POC-FINDING16-ADAPTER-AUM-INFLATION.t.sol`	7/7

Finding	File	Tests
4.19	`POC-FINDING19-STALE-ADAPTER-NAV.t.sol`	6/6
4.20	`POC-FINDING18-DONATION-CAP-DOS.t.sol`	9/9

Run all PoCs:

```
forge test --match-path 'test-foundry/pocs/POC-*.t.sol' -vvv
```

B: Phase 1 Artifacts (Context Engineering Outputs)

These artifacts were produced during Phase 1 and formed the analytical foundation for the Octane platform's analysis:

Artifact	Description
Architecture	Component boundaries, data flows, contract interactions
Threat Model	Prioritized threat categories with attack paths
Scope Assumptions	Trust boundaries, known issues, design intent items
Invariant Map	Protocol invariants (accounting, access control, economic)
Fund-Flow Analysis	Token paths, state dependencies, circuit breaker coverage
Historical Research	Vulnerability patterns from comparable interest rate swap systems
Contract Notes	Per-contract complexity notes and targets

C: Remediation Status

This appendix records the post-audit remediation status of each Section 4 finding, verified on 2026-05-19. The audit was conducted at commit `1c7f9f29`, and these statuses are assessed against `b0e76c0d`, the latest commit. Because `packages/hardhat/contracts/` was substantially refactored in between, the code references in the original findings no longer map directly to the current state of the repo; each issue was re-located by its code semantics and confirmed not to be present at the new locations.

#	Severity	Finding	Status
4.1	Critical	Missing sweep and delta-only forwarding in KairosBuyerAdapter	Fixed
4.2	High	Permissionless selective settlement / share-price gap	Fixed
4.3	High	Non-binding early-exit preview without bounds	Fixed
4.4	High	Double-counting buyer collateral in liquidation	Fixed
4.5	Medium	Time-window mismatch in bucket MtM	Fixed
4.6	Medium	Missing decimals normalization on projected rate	Fixed

#	Severity	Finding	Status
4.7	Medium	Per-bucket external base oracle dependency / DoS	Fixed
4.8	Medium	Share-price floor at MIN_SHARE_PRICE / LP dilution	Fixed
4.9	Medium	Stale-index best-effort update in RateIndexLib	Fixed
4.10	Medium	Blacklistable-token payout bricking	Fixed
4.11	Medium	Inconsistent projected rate source (spot vs tenor)	Fixed
4.12	Low	Same-tx LP deposit/withdraw fee manipulation	Fixed
4.13	Low	Post-expiry rate-window mismatch in settlement	Fixed
4.14	Low	Same-block index sample locking	Fixed
4.15	Low	Immediate bucket P&L inclusion / over-minting	Fixed
4.16	Low	Timestamp-based bucket targeting / JIT manipulation	Fixed
4.17	Low	Lack of on-chain slippage guard in buySwap	Fixed
4.18	Low	Manipulable token balance in realAssets() / AUM inflation	Fixed
4.19	Low	Stale oracle index in view-based LP share price	Fixed
4.20	Informational	Raw-balance collateral cap / donation DoS	Fixed

Remediation detail: What each fix changed, with any bounded or by-design residual noted.

- **4.1** — A sweep action and an escrow-claim path were added to KairosBuyerAdapter, and `deallocate` now detects an already-settled swap and forwards the recorded settlement proceeds instead of attempting a second `makePayment` that would revert. Funds reaching the adapter out-of-band are no longer stranded.
- **4.2** — Each bucket's mark-to-market is now capped to the actual posted collateral (`totalPoolBacking` / `totalBuyerCollateral`), eliminating the uncapped-vs-capped arithmetic gap, and `supplyCollateral` / `withdrawCollateral` gained `minSharesOut` / `minCollateralOut` guards. Permissionless settlement and opt-in slippage parameters are retained by design; because bucket MtM is now bounded by the same posted collateral that settlement pays out, the deposit-settle-withdraw price discontinuity the extraction relied on is gone.
- **4.3** — The non-binding `previewEarlyExitSettlement` was removed, and `exitSwapEarly` now takes a per-swap `minExitAmount` enforced against the actual payout after settlement. A drift or front-run that worsens the outcome past the caller's bound now reverts.
- **4.4** — The buyer-liquidation branch now sets `swap.collateralBalance = 0` before crediting `pool.totalCollateral`, so the collateral is counted once and the Token Balance Solvency invariant holds. This matches the normal-settlement and pool-liquidation paths.
- **4.5** — Bucket MtM now derives the floating rate over uncapped `totalElapsed` while scaling the payment by capped `timeElapsed`, so the time cap no longer cancels algebraically, and expired buckets use the index resolved at expiry. Post-expiry index drift is scaled down consistently with settlement.
- **4.6** — Projected-rate reads were moved onto the `ITenorRateOracle` batch interface (`getBaseRateOracleBatch`), which applies `normalizeToWadSigned` using the oracle's

`decimals()`. A non-18-decimal oracle no longer produces a scaling error in bucket P&L.

- **4.7** — `makePayment` now reads the base oracle only inside the early-exit branch, and `withdrawCollateral` bypasses the oracle when the pool is idle, removing the settlement and idle-pool DoS. LP operations on a pool with active swaps fail closed on an invalid base or reference oracle — pricing LP shares against open MtM exposure without a valid rate would let LPs enter or exit at the wrong NAV, so this is the intended behavior rather than a residual DoS.
- **4.8** — `supplyCollateral` now reverts when the share price is at `MIN_SHARE_PRICE` and the pool has active swaps, blocking opportunistic deposits at the floor. Rescue deposits remain possible only when no active swaps exist to capture recovery from.
- **4.9** — `RateIndexLib.update()` no longer returns a stale index on oracle failure; it reverts for every convention. The base/reference fail-open asymmetry is gone, so swaps can no longer be settled through an invalid-oracle window.
- **4.10** — Settlement and liquidation payouts now route through `_transferOrEscrow`, which escrows the amount for later claim if the transfer to the buyer fails instead of reverting the whole call. A blacklisted recipient can no longer brick settlement or lock pool collateral.
- **4.11** — Early-exit settlement and MtM now both read the tenor-aware `ITenorRateOracle` for the same market oracle, eliminating the spot-vs-tenor divergence on non-flat curves. The view and settlement paths were explicitly mirrored.
- **4.12** — `supplyCollateral` records the deposit block and `withdrawCollateral` reverts on a same-block withdrawal, so a buyer can no longer deposit, buy a swap at inflated free collateral, and withdraw within one transaction. The utilization-fee manipulation is no longer possible.
- **4.13** — Normal-expiry settlement now derives the floating rate from the index resolved at `entryTimestamp + swapTerm` over a fixed `swapTerm` duration, instead of the live index over uncapped elapsed time. Settlement timing no longer influences the payout.
- **4.14** — For Cumulative oracles, `update()` now always reads the live source index, so a permissionless same-block call can no longer lock a stale sample. The `elapsed == 0` short-circuit remains for the spot conventions (SpotRate and SpotCompoundRate), where it is a mathematical no-op: the spot index advances only by `rate × elapsed`, so a same-block recompute is bit-identical to the cached value and nothing exploitable remains.
- **4.15** — New swaps' fee P&L is now ramped in over the term via per-swap fee-accrual tracking (a swap contributes zero fee P&L at open), and `supplyCollateral` gained `minSharesOut`; the 1-block LP vest blocks deposit-then-exit-next-block atomic cycles.
- **4.16** — Per-bucket projected rates now come from a single batched oracle snapshot and each bucket's P&L is capped to posted collateral, removing the cheap independent-query manipulation. Deterministic timestamp-based bucket assignment and spot share pricing are unchanged, and the 1-block LP vest blocks the atomic deposit-exit cycle. Any residual cross-block distortion is bounded by the collateral cap and is the generic transaction-ordering surface of spot-priced LP shares, not a finding-specific gap.
- **4.17** — `buySwap` now takes a `rateBound` parameter (plus a `maxMarkup` bound) enforced at the core: the call reverts if the locked-in rate or markup is worse than the caller's limit. Slippage protection no longer depends on the adapter.
- **4.18** — A sweep action was added so tokens donated to the adapter are recoverable to the vault rather than permanently stranded; the permanent-stranding vector is closed. `realAssets()` still includes the raw `balanceOf` by design — it must count genuine out-of-band settlement proceeds — so a donation can only transiently pressure a curator-configured finite per-adapter cap until it is swept: self-funded griefing that is recoverable, not a permanent DoS.

- **4.19** — `getPoolSharePrice` and the adapter's `realAssets()` path now project a fresh rate index (`getFreshIndex` / virtual share price) instead of reading the stored index, so view-based NAV matches the state-changing paths. The stale-window omission is closed.
- **4.20** — The global collateral cap and its raw-`balanceOf` check in `supplyCollateral` / `buySwap` were removed entirely. The donation-based DoS vector no longer exists.