Capricorn Exchange pAMM
Security Review

Cantina Managed review by:

R0bert, Lead Security Researcher
Ladboy233, Security Researcher

November 20, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

From Oct 5th to Oct 11th the Cantina team conducted a review of pamm on commit hash db1f6c4c. The team identified a total of **20** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	2	2	0
High Risk	4	4	0
Medium Risk	3	3	0
Low Risk	7	6	1
Gas Optimizations	0	0	0
Informational	4	0	4
Total	**20**	**15**	**5**

3 Findings

3.1 Critical Risk

3.1.1 `_balancing` solves quadratic unstably, handing out negative-spread fills

Severity: Critical Risk

Context: PricingEngine.sol#L284-L298

Description: The `PricingEngine._balancing` helper computes the first-leg tranche using $\text{amountIn} = (\sqrt{B^2 + 4 \cdot A \cdot nC} - B)/(2 \cdot A)$ followed by $\text{amountOut} = \text{reserveY} - \text{price} \cdot (\text{reserveX} + \text{amountIn})$. Around equilibrium, where $\text{reserveY} \approx \text{price} \cdot \text{reserveX}$, the discriminant term $4 \cdot A \cdot nC$ is tiny, so PRB-Math floors $\sqrt{B^2 + \varepsilon}$ back down to B. The subtraction then underestimates `amountIn` and when the code plugs that value into the final subtraction the trader exits with $\text{amountOut} > \text{price} \cdot \text{amountIn}$, a negative spread. Every swap that enters `_balancing` in this regime can be reversed immediately at the oracle price for a guaranteed profit, so LPs lose inventory even though the external price never moved.

Proof of Concept: See gist 4295ad39.

Recommendation: Reformulate the quadratic using the numerically stable identity $\text{amountIn} = (2 \cdot nC)/(B \cdot (\sqrt{1+\gamma} + 1))$ with $\gamma = 4 \cdot A \cdot nC/B^2$, then recompute `amountOut` from that root so `_balancing` always returns an on-curve fill and never hands out prices richer than the oracle.

Capricorn Exchange: Fixed in commit ac723f6a.

Cantina Managed: Fix verified. `_balancing` no longer calls the naive $(\sqrt{\text{discriminant}} - B)/(2A)$ form. Instead it delegates to `_safeQuadratic`, which algebraically rewrites the root using the stable $\sqrt{1+\gamma}$ identity so the subtraction never collapses near equilibrium. This matches the recommended refactor (compute $\gamma = 4AnC/B^2$, take the stable root, then scale by $B/(2A)$), with `tiny +wrap(1)` epsilons inserted purely to round up after each division. After the stable solve, `_balancing` enforces $\text{amountIn} \geq \text{amountOut}/\text{price}$ via the `amountInMin` clamp, guaranteeing that the tranche is never priced richer than the oracle even if rounding nudges the solve downward by 1 wei.

3.1.2 Allowing `c < 1` makes `_backOutX` invert the wrong curve, enabling round-trip arbitrage

Severity: Critical Risk

Context: PricingEngine.sol#L180-L206

Description: `configurePairParams` only rejects `c == 0`, so governors can set any `c > 0`:

```
// contracts/PricingEngine.sol:180+
function configurePairParams(..., uint256 c, ...) external onlyAdmin {
    if (c == 0) revert ParamsCZero();   // ← allows 0 < c < 1
    pairParams[oracleId] = PairParams({ c: wrap(c), ... });
}
```

When the pool is quote-heavy (`price * reserveX < reserveY`), the exact-in path enters the balancing branch and for large inputs, proceeds to the two-leg "*extend*" path. The first leg is computed, then `_extend` reconstructs a virtual base reserve for the second leg via _backOutX:

```
// contracts/PricingEngine.sol (excerpt)
function _backOutX(reserveTY, amountIn, amountOut, c) internal pure returns (UD60x18 reserveTX) {
    if ((c + UNIT) * amountOut <= reserveTY * c && c > UNIT) {
        reserveTX = (reserveTY - amountOut / c) * amountIn / amountOut;
    } else {
        reserveTX = ((reserveTY - amountOut) * amountIn * c) / amountOut; // <-used whenever c   1
    }
}
```

For `c < 1`, the if condition is false and the else branch is always selected. But in this regime the base curve used by `_baseIn` is:

$$\text{amountOut} = \frac{c \cdot \text{reserveY} \cdot \text{amountIn}}{c \cdot \text{reserveX} + \text{amountIn}}$$

Solving this equation for `reserveX` gives the correct inverse:

$$\text{reserveX} = \frac{(c \cdot \text{reserveY} - \text{amountOut}) \cdot \text{amountIn}}{c \cdot \text{amountOut}}$$

However, the code in the else branch computes instead:

$$\text{reserveX} = \frac{(\text{reserveY} - \text{amountOut}) \cdot \text{amountIn} \cdot c}{\text{amountOut}},$$

which is only equal when `c = 1`. For any $0.5 \times 10^{18} < c < 1 \times 10^{18}$, the implemented formula under-estimates the effective base reserve after the first leg. The second leg then prices the remaining input against an artificially shallower curve and over-pays. If the trader immediately reverses direction at the same oracle price, the reverse leg does not over-pay, so the round trip yields a net profit.

Proof of Concept: See gist 95ff9dfb.

Recommendation: Consider restricting unsafe configurations. Add the following check in the `configure-PairParams` function to stay in the conservative regime:

```
require(c >= 1e18, "params/c<1");
```

Capricorn Exchange: Fixed in ac723f6

Cantina Managed: Fix verified. `PricingEngine` now rejects any `c` below `1e18` via `if (c < uUNIT) revert ParamsCInvalid();`, so governors can no longer configure quote-heavy pools with `c < 1`.

3.2 High Risk

3.2.1 Exact-in swaps revert near equilibrium due to catastrophic cancellation in `_balancing` and `_smallInv`

Severity: High Risk

Context: PricingEngine.sol#L296

Description: When the pool is near equilibrium (`price * reserveX   reserveY`), the exact-in path routes through `_balancing` to compute the first tranche. `_balancing` solves a quadratic and currently implements:

```
// contracts/PricingEngine.sol:292-297 (approx)
UD60x18 tmp = B.pow(convert(2)) + (convert(4) * A * nC);
UD60x18 sqrtTerm = tmp.sqrt();
UD60x18 amountIn = (sqrtTerm - B) / (convert(2) * A);
```

Here, $x := 4 \cdot A \cdot nC$ becomes very small near balance. PRB-Math UD60x18 floors both pow and sqrt, so `sqrt(B^2 + x)` often returns $\leq$ B once rounding collapses the tiny increment. The subtraction `sqrtTerm - B` then underflows in fixed-point and triggers a `panic(0x11)`. This causes valid exact-in swaps to revert whenever reserves are sufficiently close to the oracle ratio, creating a denial-of-service window around the invariant even though all branch guards have been satisfied.

The main impact of this issue is that traders cannot execute otherwise-valid swaps near equilibrium; routers/aggregators experience intermittent reverts around the invariant.

- The issue exists in `_smallInv` as well.
- In `_smallInv` the code uses the unstable quadratic form:
  - For `mult < 1`: $(\sqrt{B^2 + 4 * A * nC} - B)/(2 * A)$.
  - For `mult > 1`: $(B - \sqrt{B^2 - 4 * nA * nC})/(2 * nA)$.

Both suffer catastrophic cancellation when `sqrt(...)   B`, which is exactly the regime the code often hit near balance or with small targets.

Proof of Concept: See gist fe7606d2.

Recommendation: Replace the unstable subtraction with the numerically stable identity that avoids catastrophic cancellation and keeps the square-root input near unity:

$$\gamma := \frac{4AnC}{B^2}, \qquad \text{amountIn} = \frac{2nC}{B, (\sqrt{1+\gamma}+1)}$$

A safe implementation that also avoids forming B^2 directly:

```
UD60x18 fourAnC = convert(4) * A * nC;
UD60x18 ratio   = fourAnC / B;
ratio           = ratio / B;               // ratio = 4*A*nC / B^2
UD60x18 sqrtOne = (UNIT + ratio).sqrt();   // √(1 + ratio)
UD60x18 denom   = B * (sqrtOne + UNIT);
amountIn        = (unwrap(denom) == 0) ? wrap(0) : (convert(2) * nC) / denom;
```

This is algebraically equivalent to

$$\frac{\sqrt{B^2+4AnC}-B}{2A}$$

but eliminates the $\sqrt{B^2+x}-B$ cancellation that causes underflow and reverts.

Capricorn Exchange: Fixed in commit ac723f6.

Cantina Managed: Fix verified. `_balancing` now calls `_safeQuadratic`, which implements the numerically stable identity recommended: it divides $4 \cdot A \cdot nC$ by B twice, computes $\sqrt{\text{UNIT} + \text{ratio}}$ (so the operand stays close to 1) and only then reconstructs the large root. This eliminates the $\sqrt{B+\varepsilon}-B$ cancellation that used to underflow. Moreover, `_smallInv`'s `UNIT > mult` branch, the one that previously mirrored `_balancing` and suffered the same cancellation, now reuses `_safeQuadratic` as well, so exact-out mirrors of those trades also stay stable near equilibrium. Finally, the remaining `_smallInv` branch (`mult > UNIT`) still uses the alternative form $(B-\sqrt{B^2-4\cdot nA\cdot nC})/(2\cdot nA)$ because that case handles the opposite curvature and it never encounters the "almost-equal subtraction" that caused the original revert since $\sqrt{B^2-\varepsilon}$ is strictly smaller than B in that regime.

3.2.2 `_balancing` discriminant uses huge square; `sqrt` overflows on large pools

Severity: High Risk

Context: PricingEngine.sol#L284-L298

Description: In the quote-heavy path of exactIn, `_balancing` solves a quadratic by first building a large discriminant and then taking a square root:

```
// contracts/PricingEngine.sol:292-299
UD60x18 tX = reserveY / price;
UD60x18 A  = (convert(2) - mult) * price;
UD60x18 B  = ((convert(2) * c - UNIT) * reserveY) + price * reserveX;
UD60x18 nC = c * (reserveY - price * reserveX) * tX;

UD60x18 tmp = B.pow(convert(2)) + (convert(4) * A * nC);
UD60x18 root = tmp.sqrt();                           // ← can overflow
UD60x18 amountIn  = (root - B) / (convert(2) * A);
UD60x18 amountOut = reserveY - price * (reserveX + amountIn);
```

When reserves are very large (especially `reserveY`) and/or `c` is sizeable, `B` itself becomes huge. Squaring it via `B.pow(convert(2))` dominates the discriminant and crosses PRB-Math UD60x18's maximum representable input for sqrt. At that point `sqrt()` reverts with `PRBMath_UD60x18_Sqrt_Overflow`, which causes every exact-in swap in the quote-heavy branch to revert. Because this depends only on scale, the condition can persist across blocks and effectively denies service in one direction until reserves shrink or parameters change.

Proof of Concept: See gist 1972cf24.

Recommendation: Reformulate the quadratic using a numerically stable identity that keeps the square-root input near 1 and avoids building B^2. Let ratio = $\frac{4AnC}{B^2}$ and compute:

```
UD60x18 fourAnC = convert(4) * A * nC;
UD60x18 ratio   = fourAnC / B;         // divide once
ratio           = ratio / B;           // divide twice → (4*A*nC)/B²
UD60x18 sqrtOne = (UNIT + ratio).sqrt();
```

```
UD60x18 denom   = B * (sqrtOne + UNIT);
amountIn        = (unwrap(denom) == 0) ? wrap(0) : (convert(2) * nC) / denom;
```

This is algebraically equivalent to $(\sqrt{B \cdot B + 4 \cdot A \cdot nC} - B)/(2 \cdot A)$ but never feeds a giant operand to `sqrt`, eliminating the overflow class.

Capricorn Exchange: Fixed in commit ac723f6.

Cantina Managed: Fix verified. `_balancing` no longer builds `B.pow(2)`; it now delegates to `_safeQuadratic`. That helper rewrites the quadratic root with sequential divisions by B, keeping the square-root argument $\sim 1 + 4 \cdot A \cdot nC/B^2$ instead of a giant B^2 term. Consequently, determinant $=$ convert$(4) \cdot A \cdot nC$ -> divide by B twice $\rightarrow$ add `UNIT` $\rightarrow$ `sqrt`, matching the numerically stable identity from the recommendation.

3.2.3 `_multedSpread` underflows and reverts when `mult > 1` in the balancing uncapped branch

Severity: High Risk

Context: PricingEngine.sol#L415-L438

Description: In the quote-heavy region (`price * reserveX < reserveY`), small/medium exact-in trades (i.e., `amountInX18` $\leq$ `balancingX`) take the uncapped path:

```
// contracts/PricingEngine.sol (quote-heavy small tranche)
amountOutH = _multedSpread(tX, reserveY, amountInX18, params.c, price, mult, /*capped=*/false);
```

`_multedSpread` blends the base curve with a linear leg and then subtracts `mult * price`:

```
function _multedSpread(..., UD60x18 price, UD60x18 mult, bool capped) internal pure returns (UD60x18) {
    UD60x18 baseOut    = _baseIn(reserveTX, reserveTY, amountIn, c);
    UD60x18 finalPrice = price + (mult * baseOut / amountIn);
    if (finalPrice < mult * price && capped) {
        return _cappedOut(reserveTX, reserveTY, amountIn, c); // only for capped=true
    } else {
        finalPrice = finalPrice - mult * price;                // <-- can underflow
        return finalPrice * amountIn;
    }
}
```

This assumes `mult` $\leq$ `1` in the uncapped path. However, governance can configure `spreadMinMult > 1` (the code explicitly allows it), or in `RebalClass.EXPONENTIAL` the quote-heavy formula can produce `mult > 1` when `rebalParam0 > 1`. In those cases, for realistic parameter sets the expression `price + mult * baseOut / amountIn` can be $\leq$ `mult * price`, and the subtraction underflows in `UD60x18` (unsigned), triggering `panic(0x11)`. Because `capped=false` in this tranche, the fallback guard is not taken and the entire swap/quote reverts, even though all branch guards are satisfied.

The main impact is a persistent DoS for one side of the market whenever `mult > 1` coincides with the quote-heavy uncapped path.

Proof of Concept: See gist a240d29d.

Recommendation: Enforce `mult` $\leq$ `1` whenever `capped == false` (i.e., in the balancing/uncapped tranche), or explicitly guard the subtraction:

```
// Option A: clamp in caller when using the uncapped path
UD60x18 multBal = mult > UNIT ? UNIT : mult;
amountOutH = _multedSpread(tX, reserveY, amountInX18, c, price, multBal, /*capped=*/false);
```

```
// Option B: guard inside _multedSpread
UD60x18 baseOut = _baseIn(reserveTX, reserveTY, amountIn, c);
UD60x18 tmp     = price + (mult * baseOut / amountIn);
if (tmp <= mult * price) {
    if (capped) return _cappedOut(reserveTX, reserveTY, amountIn, c);
    return wrap(0); // or revert with a descriptive error instead of panic
}
UD60x18 finalPrice = tmp - mult * price;
return finalPrice * amountIn;
```

Document and/or constrain configuration so that quote-heavy (balancing) regimes never operate with `mult > 1`.

Capricorn Exchange: Fixed in commit ac723f6.

Cantina Managed: Fix verified. In both swap directions the quote-heavy (balancing, uncapped) path now enforces `mult <= UNIT` before `_multedSpread` is called.

3.2.4 Rebalancing allows draining the reserves

Severity: High Risk

Context: PricingEngine.sol#L586-L593

Description: The pricing engine's rebalancing branch pays the entire reserve deficit to the first trader who moves in the "correct" direction. When `price * reserveX < reserveY`, `_exactInWithScore` calls `_balancing` and, if the caller's `amountInX18` is even slightly larger than the returned `balancingX`, it immediately invokes `_extend` to bridge the deficit. Because there is no cap tying `amountOut` to the trader's input or to the available reserves, a 1 `token1` input can withdraw 499999 `token0` from a 500 000 token0 pool. `PAMMPool::swap` never blocks this because `pricingEngine.setMaxInputAmount(token0/token1)` is left at zero, meaning "unbounded", so the vulnerable flow is:

```
// contracts/PricingEngine.sol
if (price * reserveX < reserveY) {
    (UD60x18 balancingX, UD60x18 balancingY) = _balancing(...);
    if (amountInX18 > balancingX) {
        amountOutH = _extend(reserveY, amountInX18 - balancingX, balancingX, balancingY, params.c);
    } else {
        amountOutH = _multedSpread(...);
    }
}
```

The exploit test test/foundry/PAMMPoolExploit.t.sol reproduces the failing fuzz seed: it deploys the real components, sets the oracle mantissa to 1654860 (expo −6), and seeds the pool with 500000 `token0` / 827430 `token1`. Running `forge test --match-path test/foundry/PAMMPoolExploit.t.sol -vv` shows `[loop 1] direction=token1->token0 | input=1` immediately returning 455400 token1 after the round trip, dropping the pool to 500 000/372031. Impact: an attacker can drain roughly 55 % of the quote reserves with negligible cost.

Recommendation: Cap the rebalancing subsidy so `amountOut` never exceeds the lesser of `reserveOut` and `amountIn * oraclePrice * (1 - fee)` and amortize `_balancing` across multiple trades instead of paying it all to the first caller. Additionally, configure `pricingEngine.setMaxInputAmount` for both tokens to prevent any single swap from draining the scarce reserve, even a conservative per-token cap would block the 1 unit exploit.

Capricorn Exchange: Fixed in commit fe6be101.

Cantina Managed: Fix verified.

3.3 Medium Risk

3.3.1 `EXP` rebalancing uses an unsafe exponent bound

Severity: Medium Risk

Context: PricingEngine.sol#L401-L405

Description: In the `Exponential` rebalancing class, the spread multiplier for the base-heavy side (`epsilon > 1`) is computed as:

```
// contracts/PricingEngine.sol:396-404 (excerpt)
UD60x18 base = convert(2);
UD60x18 exp  = (eps - UNIT) / params.rebalParam1;
if (exp >= convert(196)) { // overflow, default to spreadMaxMult
    mult = params.spreadMaxMult;
} else {
    mult = base.pow(exp);  // effectively exp2(exp) because base==2
}
```

`UD60x18.pow` with `base = 2` reduces to `exp2(exp)` under PRB-Math.The maximum supported input for `exp2` in `UD60x18` is approximately 192×10^{18}, not 196. With the current guard (196), configurations with a small `rebalParam1` (intended to make spreads reactive) and moderate skew can yield exp $\in [192, 196)$. In that range, the guard does not trigger, `base.pow(exp)` calls `exp2` with an out-of-bounds exponent and PRB-Math reverts with `PRBMath_UD60x18_Exp2_InputTooBig`. Once the pool's imbalance drives exp into this window, every swap that evaluates this branch reverts bricking the pool on the base-heavy side until parameters or state change.

Proof of Concept: See gist fada79c4.

Recommendation: Clamp against the correct bound and short-circuit before calling `pow`:

```
// Use 192.0e18 as the safe exp2 upper bound in UD60x18
if (exp >= convert(192)) {
    mult = params.spreadMaxMult; // saturate instead of reverting
} else {
    mult = base.pow(exp);
}
```

Capricorn Exchange: Fixed in commit ac723f6.

Cantina Managed: Fix verified. In the new `PricingEngine` version the base-heavy path computes `exp = (eps - UNIT) / params.rebalParam1` and immediately checks `if (exp >= convert(192)) { mult = params.spreadMaxMult; } else { mult = base.pow(exp); }`. That `192e18` threshold matches PRB-Math's documented limit, so `base.pow(exp)` is never invoked with an out-of-range exponent and the branch saturates instead of reverting.

3.3.2 Endorser cannot cancel the endorsement when configuration changes

Severity: Medium Risk

Context: PricingEngine.sol#L97-L117

Description: The endorsement schema is mostly solid:

```
function hashStruct(EndorsementData memory data) internal pure returns (bytes32) {
    return keccak256(
        abi.encode(
            ENDORSEMENT_TYPEHASH,
            data.endorser,
            data.trader,
            data.oracleId,
            data.zeroForOne,
            data.amountSpecified,
            data.recipient,
            data.deadline,
            data.nonce
        )
    );
}
```

Endorsements are non-cancelable once signed, and the signed payload does not bind to mutable protocol state beyond `oracleId`.

Because the pool's pricing dependencies can change (e.g., the `PAMMPool` pricing engine address or the oracle configuration behind a given `oracleId`), an otherwise valid signature may become stale yet still executable. That creates a mismatch between the signer's intent and current risk parameters.

Recommendation: Invalidate endorsements whenever pricing dependencies change, and cryptographically bind endorsements to the current config.

Capricorn Exchange: Fixed in commit f87af74.

Cantina Managed: Fix verified.

3.3.3 `mult has to be` ≤ 1 invariant is not properly enforced

Severity: Medium Risk

Context: PricingEngine.sol#L572-L587

Description: The `_balancing` helper solves a quadratic where `A = (2 - mult) * price` appears in the denominator:

```
UD60x18 A  = (convert(2) - mult) * price;
UD60x18 amountIn = (sqrt(B^2 + 4*A*nC) - B) / (2*A);
```

The math assumes `A > 0`, i.e., `mult < 2`. However, the function's comment ("mult has to be ≤ 1") is not enforced anywhere, and `_mult(...)` can yield `mult > 1`. If `mult` ≤ 2, then `A` $\leq$ `0`, making the division invalid (division-by-zero or negative denominator), and even when `1 < mult < 2`, the algebraic intent of the balancing solution no longer holds. This leads to fragile behavior, potential underflow, and revert paths in quote-heavy states. Separately, the original implementation formed a huge discriminant ($B^2 + 4AnC$) and fed it to `sqrt`, which can overflow PRB-Math's `UD60x18` sqrt domain for large reserves/`c`.

Proof of Concept: Add test to fe7606d2.

```
function test_Repro_MultGreaterThanTwo_TriggersBalancingUnderflow() public {
    address admin = makeAddr("admin");
    address operator = makeAddr("operator");
    address lp = makeAddr("lp");

    vm.startPrank(admin);
    OracleRegistry oracleRegistry = new OracleRegistry();
    PricingEngine pricingEngine = new PricingEngine(address(oracleRegistry), admin);
    PAMMPoolFactory factory = new PAMMPoolFactory();
    SegmenterRegistry segmenterRegistry = new SegmenterRegistry(admin);
    TestERC20 token0 = new TestERC20("Token0", "TK0", 18);
    TestERC20 token1 = new TestERC20("Token1", "TK1", 6);

    oracleRegistry.grantRole(oracleRegistry.OPERATOR_ROLE(), operator);
    oracleRegistry.grantRole(oracleRegistry.GUARD_ROLE(), admin);
    segmenterRegistry.setOperator(operator);

    // Register pair and create pool
    bytes32 oracleId = oracleRegistry.registerOracle(address(token0), address(token1), 18, 6);

    // Choose EXPONENTIAL with rebalParam0 > 1, so when eps <= 1   mult = rebalParam0^(positive) > 1.
    // Set spreadMaxMult high so clamping does not limit us; we want mult > 2.
    // We'll arrange eps ~= 0.5 so exponent ~= 1 and mult   rebalParam0.
    pricingEngine.configurePairParams(
        oracleId,
        1e18,                               // c
        PricingEngine.RebalClass.EXPONENTIAL, // rebal class
        1e16,                               // spreadMinMult = 0.01
        100e18,                             // spreadMaxMult = 100
        3e18,                               // rebalParam0 = 3  (BASE > 1   mult > 1)
        1e18                                // rebalParam1 (unused for eps<=1 path)
    );
    pricingEngine.setSegmenterRegistry(address(segmenterRegistry));
    pricingEngine.setMaxInputAmount(address(token0), 0);
    pricingEngine.setMaxInputAmount(address(token1), 0);

    // Create pool
    bytes32 salt = hex"1111111111111111111111111111111111111111111111111111111111111111";
    address poolAddr = factory.createPool(
        address(token0),
        address(token1),
        oracleId,
        address(pricingEngine),
        operator,
        admin,
        11, // fee bps (irrelevant here)
        salt
    );
    PAMMPool pool = PAMMPool(poolAddr);
    pool.grantRole(pool.LP_ROLE(), lp);
    vm.stopPrank();




    // Oracle price: 1 token0 = 4 token1
    {
        bytes32[] memory ids = new bytes32[](1);
        int64[] memory prices = new int64[](1);
        uint64[] memory confs = new uint64[](1);
        int32[] memory expos = new int32[](1);
        ids[0] = oracleId;
        prices[0] = 4;
        confs[0] = 0;
        expos[0] = 0;
        vm.prank(operator);
        oracleRegistry.updateNativePrices(ids, prices, confs, expos);
    }

    // Reserves: make pool QUOTE-heavy so the code takes the _balancing branch (price*reserveX < reserveY).
    // Also set reserve0 small so eps ~ 0.5.
    // eps (zeroForOne=true) = (reserve0*price + reserve1) / (2*reserve1) = (price*R0/R1 + 1)/2   ~0.5 if R0 is
    ↪   tiny.
    uint256 r0 = 1e12;    // 1e-6 token0
    uint256 r1 = 1_000_000; // 1,000,000 token1
    token0.mint(lp, r0);
    token1.mint(lp, r1);
    vm.startPrank(lp);
    token0.approve(address(pool), type(uint256).max);
    token1.approve(address(pool), type(uint256).max);
    pool.deposit(r0, r1);
    vm.stopPrank();

    // Now quote zeroForOne (token0 -> token1). With eps 0.5 and EXPONENTIAL, mult   rebalParam0 = 3 > 2.
    // That enters `_balancing` and computes A = (2 - mult) * price => underflows UD60x18 (since 2 - 3 < 0).
    uint256 amountIn = 1e9; // tiny input is enough to exercise the path
    vm.expectRevert(stdError.arithmeticError);
    pool.quoteExactIn(address(token0), amountIn);
}
```

Recommendation: Enforce the contract on `mult` for the balancing branch. Clamp `mult` to ≤ 1 at the call site and add a defensive check inside `_balancing`:

```
error BalancingRequiresMultLeOne();

function _clampLeOne(UD60x18 x) internal pure returns (UD60x18) {
    return x < UNIT ? x : UNIT;
}

// Call site (quote-heavy branch)
UD60x18 mBal = _clampLeOne(mult);
(UD60x18 balancingX, UD60x18 balancingY) = _balancing(reserveX, reserveY, params.c, price, mBal);

// Inside _balancing
if (mult > UNIT) revert BalancingRequiresMultLeOne();
```

This preserves existing behavior elsewhere (where `mult > 1` may be desired) while guaranteeing the balancing math's preconditions.

Capricorn Exchange: Fixed in commit ac723f6.

Cantina Managed: Fix verified. In the `PricingEngine` exact-in and exact-out the quote-heavy branch now begins with `require(mult <= UNIT, "mult > UNIT");`. Any configuration or runtime state that would push `mult` above 1 causes the call to revert before `_balancing` is invoked, so `A = (2 - mult) * price` never goes non-positive. Base-heavy branches retain the complementary `require(mult >= UNIT, "mult < UNIT");`, so each regime only allows multipliers that satisfy its math assumptions. `_balancing` itself doesn't need an extra guard because the only paths that reach it already satisfy `mult` ≤ 1.

3.4 Low Risk

3.4.1 Reserves can drop below accrued fees, permanently blocking fee claims

Severity: Low Risk

Context: PAMMPool.sol#L251-L269

Description: Both withdraw and `claimFees` operate on the raw reserves without protecting the fee balance. `withdraw` only checks `amount0 <= reserve0` and `amount1 <= reserve1`, then subtracts those amounts even if the remainder is less than the outstanding `accruedFee{0,1}`. Later, `claimFees` reads `claim0 = accruedFee0` and `claim1 = accruedFee1`, zeroes the counters and reverts with `InsufficientLiquidity()` whenever `reserve0 < claim0` or `reserve1 < claim1`. Swaps exacerbate this because swap updates `reserve0/reserve1` and performs `TransferHelper.safeTransfer(tokenOut, recipient, amountOut)` without reserving the fee balance. A near-total one-for-zero swap can leave `reserve0` below `accruedFee0`, so the very next `claimFees` call will revert. With only one LP today the effect is mostly cosmetic, but as soon as multiple LPs participate a withdrawing or trading LP can leave the pool owing more fees than it holds, permanently denying the remaining LPs access to their accrued rewards.

Recommendation: Track fee collateral separately from trading reserves or, at minimum, enforce `reserve0 - amount0 >= accruedFee0` and `reserve1 - amount1 >= accruedFee1` inside `withdraw`, while making `swap` operate on net reserves that exclude `accruedFee{0,1}` so the pool can never send out more than the spendable portion.

Capricorn Exchange: Fixed in commit 0db8e9a.

Cantina Managed: Fix verified.

3.4.2 Quoter/Swap revert with panic when a reserve is zero

Severity: Low Risk

Context: PAMMPool.sol#L199-L241

Description: `PAMMPool.quoteExactIn` delegates pricing to `PricingEngine.exactIn` even when one side of the pool is empty or effectively dust. Inside the engine, the spread helper `_mult` computes the imbalance metric `eps` by dividing by twice the opposite reserve:

```
// contracts/PricingEngine.sol (excerpt)
UD60x18 eps;
if (zeroForOne) {
    eps = (reserve0 * price + reserve1) / (convert(2) * reserve1);
} else {
    eps = (reserve1 / price + reserve0) / (convert(2) * reserve0);
}
```

A fresh pool, or a pool that just withdrew an entire side, will have `reserve0 == 0` or `reserve1 == 0`. The divisions above then hit a division-by-zero and Solidity throws `panic(0x12)` before any rebalancing logic can run. Because `quoteExactIn` simply forwards the engine's revert, routers and integrators see an opaque low-level panic instead of a structured "insufficient liquidity" error; the same path is reachable from `swap` and will also cause a panic. Even when reserves are not exactly zero but extremely small, the denominator terms magnify the inputs that drive the EXP-class exponent logic, so downstream calculations hit their numerical limits more quickly and swaps continue to revert until liquidity is restored.

Recommendation: Short-circuit at the pool surface with a clear error before calling the engine, so integrators never see a low-level panic. At the top of both quoteExactIn and swap, after you read reserves, add:

```
if (reserve0 == 0 || reserve1 == 0) revert InsufficientLiquidity();
```

Optionally enforce a small minimum reserve threshold to avoid near-zero denominators that can destabilize downstream math.

Capricorn Exchange: Fixed in commit e9a65c1.

Cantina Managed: Fix verified.

3.4.3 Constructor-Based role setup breaks with proxy

Severity: Low Risk

Context: OracleRegistry.sol#L102-L103, PricingEngine.sol#L93-L94

Description: The contract includes a `__gap` for upgradeability but assigns roles in a constructor. When deployed behind a proxy, the constructor does not execute, leaving roles unset and effectively no admin on the proxied instance.

Recommendation: Make the contract either non-upgradeable or adopt a proper upgradeable pattern:

- Use OpenZeppelin's `Initializable/AccessControlUpgradeable`.
- Move role setup into an `initialize` (or `reinitializer`) function guarded by `initializer`.
- In the implementation contract's constructor, call `_disableInitializers()` to prevent misuse.

- Ensure the proxy calls `initialize(...)` immediately after deployment to grant `DEFAULT_ADMIN_ROLE` and any other roles.

Capricorn Exchange: Fixed in commit bbe1a13.

Cantina Managed: Fix verified.

3.4.4 `setPricingEngine` missing token validation

Severity: Low Risk

Context: PAMMPool.sol#L165-L171

Description: When switching the pricing engine (or updating `oracleId`), the contract does not re-validate that the referenced oracle is registered and oriented for the current pair (`token0/token1`). A misconfigured or stale `oracleId` can point to an unregistered market or to a `(base,quote)` that doesn't match the pair, causing incorrect pricing, misquotes, or loss of funds.

Recommendation: Replicate the constructor's validation inside the setter that changes the pricing engine and/or `oracleId`. Require that the market exists and that `(base, quote)` matches `(_token0, _token1)` before applying the change.

Capricorn Exchange: Fixed in commit ca890ab.

Cantina Managed: Fix verified.

3.4.5 `withdraw` in `PAMMPool` does not have `whenNotPaused` modifier

Severity: Low Risk

Context: PAMMPool.sol#L388-L390

Description: `withdraw` lacks a pause guard. During emergencies (oracle failure, pricing bug, exploit), LP withdrawals would remain callable, preventing the protocol from halting outflows and containing damage.

Recommendation: Gate `withdraw` with `whenNotPaused`. If withdrawals must remain available under specific conditions, document and implement a separate `emergencyWithdraw()` with tighter limits and auditing hooks instead of leaving `withdraw` unguarded.

Capricorn Exchange: Fixed in commit bdc4ca1.

Cantina Managed: Fix verified.

3.4.6 Avoid hardcode `MAX_EXTERNAL_ORACLE_STALENESS` and `MAX_NATIVE_PRICE_STALENESS`

Severity: Low Risk

Context: OracleRegistry.sol#L57-L59

Description: Oracle and price staleness thresholds are hard-coded. This reduces flexibility across networks/feeds and forces redeploys to tune risk. Calls to `getPriceNoOlderThan` will revert if the on-chain update is older than the provided heartbeat (e.g., 3600 seconds).

Recommendation: Make staleness thresholds configurable. Consider fetching with `getPriceUnsafe` and enforcing the staleness check in a flexible manner.

Capricorn Exchange: Fixed in commit ef9132e.

Cantina Managed: Fix verified.

3.4.7 `PAMMPool.sol` is not compatible with rebasing token

Severity: Low Risk

Context: PAMMPool.sol#L352-L354

Description: The pool is not compatible with rebasing (elastic-supply) tokens. The contract tracks internal reserves (`reserve0/ reserve1`) and then asserts equality with live balances after each operation:

```
uint256 balance0 = IERC20Minimal(token0).balanceOf(address(this));
uint256 balance1 = IERC20Minimal(token1).balanceOf(address(this));
if (balance0 != reserve0 || balance1 != reserve1) {
    revert FeeOnTransferDetected(balance0, reserve0, balance1, reserve1);
}
```

A positive/negative rebase changes `balanceOf(address(this))` without a transfer, so the next call sees `balances` `reserves` and reverts. Even if this equality check were removed, pricing/invariant math would be wrong because storage reserves would be stale relative to actual balances.

Recommendation: Ensure neither base token nor quote token is a rebase token.

Capricorn Exchange: Acknowledged. The pool creation and token choice is at the discretion of the pool deployer, who needs to be be cautious with not using a rebasing token.

Cantina Managed: Acknowledged.

3.5 Informational

3.5.1 Blocklisted traders can swap by routing through a clean wrapper

Severity: Informational

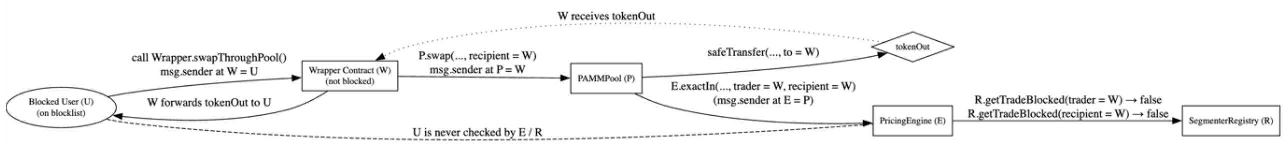
Context: PricingEngine.sol#L452-L459

Description: `PricingEngine.exactIn` enforces the `SegmenterRegistry` blocklist by looking at the immediate caller and the on-chain recipient:

```
// contracts/PricingEngine.sol:452-458
if (trader != address(0)) {
    bool p1 = segmenterRegistry.getTradeBlocked(trader);
    if (p1) revert UnauthorizedSwap();
}
if (recipient != address(0)) {
    bool p2 = segmenterRegistry.getTradeBlocked(recipient);
    if (p2) revert UnauthorizedSwap();
}
```

`PAMMPool` passes `msg.sender` as trader and the swap recipient as recipient. A wallet that sits on the blocklist can trivially deploy an unblocked wrapper contract, fund it, and have the wrapper call swap with `recipient = wrapper`. The registry now sees the wrapper address in both checks, finds it unblocked and returns without reverting. Once the pool transfers the output to the wrapper, the wrapper immediately forwards the tokens to the originally blocked wallet. Because the registry never inspects the ultimate beneficiary and the pool has no way to distinguish a wrapper from a genuine trader, the blocklist is effectively unenforceable: any banned user can route swaps through a single thin proxy and trade indefinitely.

Graphical representation of the bypass:


W receives tokenOut
call Wrapper.swapThroughPool()
msg.sender at W = U
Wrapper Contract (W)
(not blocked)
P.swap(..., recipient = W)
msg.sender at P = W
PAMMPool (P)
safeTransfer(..., to = W)
tokenOut
Blocked User (U)
(on blocklist)
W forwards tokenOut to U
E.exactIn(..., trader = W, recipient = W)
(msg.sender at E = P)
R.getTradeBlocked(trader = W) → false
R.getTradeBlocked(recipient = W) → false
PricingEngine (E)
SegmenterRegistry (R)
U is never checked by E / R

Recommendation: Merely informative. In this case, the only way to enforce a proper user whitelist/blacklist is by requesting to pass signature validation upon every swap.

Capricorn Exchange: Acknowledged.

Cantina Managed: Acknowledged.

3.5.2 PAMM fuzzing harness

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Link efd1379c.

We built a dedicated fuzzing suite to exercise the PricingEngine's swap math under randomized market configurations. The harness runs as a Foundry-style property test. It seeds random reserves, curve parameters, fee settings, and trade sizes, then simulates an exact-in swap in one direction using `PricingEngine.exactIn`, updates the virtual reserves as `PAMMPool` would and immediately performs the reverse exact-in swap to push value back into the original asset. It asserts that the round trip never yields a positive net balance for the trader once fees are applied. Whenever the engine returns a profit, the harness records the full parameter set, providing reproducible counterexamples. This fuzz target gives broad coverage across the parameter space and quickly exposes curve inconsistencies or rounding edge cases that manual review would likely miss.

Recommendation: Incorporate this fuzzing target (or an adapted in-repo version) into the automated test suite so every code change runs the round-trip invariant and flags numerical regressions before deployment.

Capricorn Exchange: Acknowledged.

Cantina Managed: Acknowledged.

3.5.3 Unused Code

Severity: Informational

Context: OracleRegistry.sol#L34-L35, PAMMPool.sol#L52-L53

Description:

- Error `RateLimitExceed` is not used.
- Event `EngineVersionUpdated` is not emitted.

Recommendation: Remove the unused code.

Capricorn Exchange: Acknowledged.

Cantina Managed: Acknowledged.

3.5.4 Token existence check is missing in `TransferHelper.sol`

Severity: Informational

Context: TransferHelper.sol#L8-L24

Description: The contract does not verify that the token address points to a deployed ERC-20 contract. As a result, transfers could silently succeed (no revert) against a non-existent or non-compliant token, leading to unexpected behavior or lost funds.

Recommendation: Add an explicit token-existence check (e.g., `address(token).code.length > 0`) and use `SafeERC20` for transfers to handle non-standard return values.

Reference: OpenZeppelin `SafeERC20` (see call pattern around optional returns): `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol#L196`.

Capricorn Exchange: Acknowledged.

Cantina Managed: Acknowledged.