



Security Review For Metric



Collaborative Audit Prepared For:
Lead Security Expert(s):

Metric
eeyore
TessKimy

Date Audited:

May 22 - June 19, 2026

Introduction

tba

Scope

Repository: Metric-OMM/metric-core

Audited Commit: 6aa6c3b489b84b8b1c50dc6a4967184df17aa395

Final Commit: 7b9ab567631a234ba5d467c646a1da9cbfb25479

Files:

- contracts/Extsload.sol
- contracts/interfaces/callbacks/IMetricOmmModifyLiquidityCallback.sol
- contracts/interfaces/callbacks/IMetricOmmSwapCallback.sol
- contracts/interfaces/hooks/IMetricOmmHooks.sol
- contracts/interfaces/IDepositAllowlistProvider.sol
- contracts/interfaces/IExtsload.sol
- contracts/interfaces/IMetricOmmPoolFactory/IMetricOmmPoolFactoryOwner.sol
- contracts/interfaces/IMetricOmmPoolFactory/IMetricOmmPoolFactoryPoolAdmin.sol
- contracts/interfaces/IMetricOmmPoolFactory/IMetricOmmPoolFactory.sol
- contracts/interfaces/IMetricOmmPool/IMetricOmmPoolActions.sol
- contracts/interfaces/IMetricOmmPool/IMetricOmmPoolCollectFees.sol
- contracts/interfaces/IMetricOmmPool/IMetricOmmPoolFactoryActions.sol
- contracts/interfaces/IMetricOmmPool/IMetricOmmPool.sol
- contracts/interfaces/IPriceProvider/IPriceProvider.sol
- contracts/interfaces/IPriceProvider/IPriceProviderSwapReporter.sol
- contracts/interfaces/ISwapAllowlistProvider.sol
- contracts/libraries/BinDataLibrary.sol
- contracts/libraries/LiquidityLib.sol
- contracts/libraries/MetricHooks.sol
- contracts/libraries/PoolActions.sol
- contracts/libraries/PoolStateLibrary.sol
- contracts/libraries/SignedMath.sol

- contracts/libraries/Slot0Library.sol
- contracts/libraries/SwapMath.sol
- contracts/MetricOmmPoolDeployer.sol
- contracts/MetricOmmPoolFactory.sol
- contracts/MetricOmmPool.sol
- contracts/types/FactoryOperation.sol
- contracts/types/FactoryStorage.sol
- contracts/types/HookTypes.sol
- contracts/types/PoolOperation.sol
- contracts/types/PoolStorage.sol
- contracts/utils/MetricReentrancyGuardTransient.sol

Repository: Metric-OMM/metric-periphery

Audited Commit: 90039f9b68f6b2253425acfb6497fcf38e28cc17

Final Commit: d210a84daf694c52a591d371ceb9b82cece0f79f

Files:

- contracts/base/MetricOmmSwapRouterBase.sol
- contracts/base/SelfPermit.sol
- contracts/common/MetricOmmPoolQuoter.sol
- contracts/hooks/base/BaseMetricHook.sol
- contracts/hooks/base/SubhookUtils.sol
- contracts/hooks/subhooks/DepositAllowlistSubhook.sol
- contracts/hooks/subhooks/OracleValueStopLossSubhook.sol
- contracts/hooks/subhooks/PriceVelocityGuardSubhook.sol
- contracts/hooks/subhooks/SwapAllowlistSubhook.sol
- contracts/hooks/subhooks/SwapReporterSubhook.sol
- contracts/interfaces/external/IERC20PermitAllowed.sol
- contracts/interfaces/IMetricOmmPoolLiquidityAdder.sol
- contracts/interfaces/IMetricOmmPoolSwapper.sol
- contracts/interfaces/IMetricOmmSimpleRouter.sol
- contracts/interfaces/IMulticall.sol

- contracts/interfaces/ISelfPermit.sol
- contracts/libraries/TransientCallbackPool.sol
- contracts/MetricOmmPoolLiquidityAdder.sol
- contracts/MetricOmmPoolSwapper.sol
- contracts/MetricOmmSimpleRouter.sol

Repository: Oracle-Based-Pool/smart-contracts-poc

Audited Commit: a7644f6baf09e9c595c64c26764b6b8ef7358a6d

Final Commit: 056c20454dd867e388986f83b78d05809b921e49

Files:

- contracts/interfaces/IChainlinkVerifier.sol
- contracts/interfaces/ICompressedOracleV1.sol
- contracts/interfaces/IOffchainFeedOracle.sol
- contracts/interfaces/IOffchainOracle.sol
- contracts/interfaces/IPriceProviderFactoryL2.sol
- contracts/interfaces/IPriceProviderFactory.sol
- contracts/oracles/compressed/CompressedOracle.sol
- contracts/oracles/compressed/OracleBase.sol
- contracts/oracles/providers/ChainlinkOracle.sol
- contracts/oracles/providers/OracleBase.sol
- contracts/oracles/providers/PythOracle.sol
- contracts/oracles/utils/ChainlinkVerifierL2.sol
- contracts/oracles/utils/ChainlinkVerifier.sol
- contracts/oracles/utils/Codebook256.sol
- contracts/oracles/utils/LazerConsumer.sol
- contracts/oracles/utils/OracleMath.sol
- contracts/oracles/utils/TimeMs.sol
- contracts/oracles/utils/U64x32.sol
- contracts/PriceProviderFactoryL2.sol
- contracts/PriceProviderFactory.sol
- contracts/PriceProviderL2.sol

- contracts/PriceProvider.sol
- contracts/ProtectedPriceProviderL2.sol
- contracts/ProtectedPriceProvider.sol

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
1	9	31

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: MetricOmmSimpleRouter router accepts an unverified pool address, letting a malicious pool drain user token allowances

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/104>

Summary

The MetricOmmSimpleRouter never verifies that `params.pool` is a real factory pool (MetricOmmPoolFactory.isPool() is never called), and its swap callback reads the payer to debit from the data the calling pool supplies. A malicious contract registered as the pool can therefore call the callback and pass its own data naming any victim as payer, draining the entire allowance that victim granted the router.

Vulnerability Detail

The entrypoint builds a data blob with the intended payer and hands it to swap:

```
IMetricOmmPoolActions(params.pool).swap(
    ..., abi.encode(JustPayCallbackData({tokenToPay: params.tokenIn, payer:
    ↪ msg.sender}))), params.hookData
);
```

This is only the data the router *intends*. There is no binding between the blob the router passes into `swap` and the blob that later arrives in `metricOmmSwapCallback`: the callback receives whatever data its caller (the pool) chooses to pass. A malicious pool simply discards the router's blob and supplies its own.

```
function metricOmmSwapCallback(int256 amount0Delta, int256 amount1Delta, bytes
    ↪ calldata data) external {
    ...
    _requireExpectedCallbackCaller(msg.sender);           // msg.sender == getPool()
    ↪ == stored params.pool (attacker)
    ...
    _justPayCallback(amount0Delta, amount1Delta, data);    // `data` is whatever the
    ↪ pool passed in
}

function _justPayCallback(int256 amount0Delta, int256 amount1Delta, bytes calldata
    ↪ data) private {
    JustPayCallbackData memory cb = abi.decode(data, (JustPayCallbackData)); // payer
    ↪ is read from the attacker's `data`
    _pay(cb.tokenToPay, cb.payer, msg.sender,
    ↪ uint256(_getPositiveAmount(amount0Delta, amount1Delta)));
```

```
}  
// _pay -> IERC20(cb.tokenToPay).safeTransferFrom(cb.payer, msg.sender = attacker  
↳ pool, amount)
```

Both `cb.payer` and `cb.tokenToPay` come straight from the callback data the attacker's pool controls, and the pulled `amount` comes from the deltas that same pool passes in. So the attacker chooses who is debited, which token, and how much, capped only by the victim's standing allowance and balance. The caller check does not help: `_requireExpectedCallbackCaller` only asserts `msg.sender == getPool()`, and `getPool()` returns the attacker-chosen `params.pool`, so it reduces to `attacker == attacker`.

The interface documents the missing provenance check ("implementations do not verify factory provenance"). That disclaimer would be acceptable if `payer` and `tokenToPay` were stored at entry and not alterable in the callback: an untrusted pool would still be usable either way, but the worst case would be the caller debiting only themselves, which merely lowers the risk to self-harm (Low). It is the alterable callback data carrying an alterable `payer`, not the missing pool check alone, that escalates this to draining arbitrary victims.

Impact

Any tokens a user has approved to the router are drainable, with no action required from the victim.

Code Snippet

<https://github.com/Metric-OMM/metric-periphery/blob/90039f9b68f6b2253425acfb6497fcf38e28cc17/contracts/MetricOmmSimpleRouter.sol#L46> <https://github.com/Metric-OMM/metric-periphery/blob/90039f9b68f6b2253425acfb6497fcf38e28cc17/contracts/MetricOmmSimpleRouter.sol#L190-L193> <https://github.com/Metric-OMM/metric-periphery/blob/90039f9b68f6b2253425acfb6497fcf38e28cc17/contracts/MetricOmmSimpleRouter.sol#L75> <https://github.com/Metric-OMM/metric-periphery/blob/90039f9b68f6b2253425acfb6497fcf38e28cc17/contracts/base/MetricOmmSwapRouterBase.sol#L59-L71> <https://github.com/Metric-OMM/metric-periphery/blob/90039f9b68f6b2253425acfb6497fcf38e28cc17/contracts/libraries/TransientCallbackPool.sol#L57-L59> <https://github.com/Metric-OMM/metric-periphery/blob/90039f9b68f6b2253425acfb6497fcf38e28cc17/contracts/interfaces/IMetricOmmSimpleRouter.sol#L11>

Tool Used

Manual Review

Recommendation

Do not trust the callback data to identify the payer. Mirror the old router: store the payer (and token/limits) in transient context at the entrypoint with `payer = msg.sender`, and in

the callback read the payer from that transient context, never from the pool-supplied data.

In addition, wire the `MetricOmmPoolFactory` as an immutable and gate the callback on `factory.isPool(msg.sender)` (and reject non-pool `params.pool` at entry) so only a genuine pool can invoke the callback at all.

Discussion

Oxklapouchy

Fix confirmed in [PR#38](#)

Issue M-1: addLiquidityWeighted has no output-side guard, enabling an LP sandwich

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/84>

Summary

`addLiquidityWeighted` derives the *composition* of the deposit from the pool's live cursor at call time, but the only guards it exposes are `maxAmountToken0` / `maxAmountToken1`, which cap the tokens **spent**. There is no minimum-shares, composition-band, or deadline guard. An attacker swaps the cursor off the fair price immediately before the victim's add, so the probe returns a skewed `need0` / `need1`; the victim deposits at that manipulated ratio (scaled down to fit the caps), and the attacker reverses the cursor in the same block to capture the impermanent-loss gap. The caps bound how much the victim spends, not the price at which their liquidity is placed.

Vulnerability Detail

The weighted add is a two-call sequence. First a **probe** asks the pool what a provisional share vector would cost *at the current cursor*; then the shares are rescaled to fit the caps and the **paying** add is executed:

```
// MetricOmmPoolLiquidityAdder.addLiquidityWeighted
try IMetricOmmPoolActions(pool).addLiquidity(owner, salt, weightDeltas,
    ↪ abi.encode(KIND_PROBE), hookData) {
    revert WeightedProbeInconclusive();
} catch (bytes memory reason) {
    (uint256 need0, uint256 need1) = _decodeLiquidityProbeOrBubble(reason);
    ↪ // cost at LIVE cursor
    LiquidityDelta memory scaled = _scaleWeightsToShares(weightDeltas,
    ↪ maxAmountToken0, maxAmountToken1, need0, need1);
    return _addLiquidity(pool, owner, salt, scaled, msg.sender, maxAmountToken0,
    ↪ maxAmountToken1, hookData);
}
```

`_scaleWeightsToShares` multiplies every weight by `min(max0/need0, max1/need1)`. This only scales the total size; the *ratio* of token0 to token1 the LP ends up depositing is whatever the pool's current bin composition is. In `LiquidityLib.addLiquidity`, a deposit into a non-empty (active) bin is charged strictly in proportion to the bin's live `token0BalanceScaled` / `token1BalanceScaled`, and a deposit into an empty bin is decided by the live `curBinIdx` / `curPosInBin`. All of these are moved by an ordinary swap: `SwapMath` walks the cursor across bins and rewrites each swept bin's `token0BalanceScaled` / `token1BalanceScaled`.

So the deposit composition is a pure function of pool state that any actor can set with a swap in the same block, and the caller has no parameter to constrain it. `max0` / `max1` only

cap the spend, the only on-callback check (`MaxAmountExceeded`) only re-checks the spend, and there is no deadline. Adding liquidity at a cursor that has been pushed away from fair is economically identical to trading at that wrong price: when the cursor is restored, the LP's pro-rata claim is worth less than what they put in, and the difference is realised by whoever moves the cursor back.

This weakness is specific to the weighted path. `addLiquidityExactShares` pins the share output, so a manipulated composition raises the cost on the heaved side past its cap and the add reverts (`MaxAmountExceeded`), which is working slippage protection. `addLiquidityWeighted` is exposed precisely because the rescaling keeps the spend under the caps while leaving the deposit ratio fully attacker-controlled, so it never fails closed.

Attack scenario (ETH/USDC, mid = 3000, a 200 bps active bin with the cursor at its midpoint, a true 50/50 composition):

1. The pool sits at the fair mid and the active bin is ~50/50. An LP wants to add 10000 USD of liquidity 50/50 and submits `addLiquidityWeighted` with `max0` ~ 5000 USD -of-ETH and `max1` ~ 5000 USDC, each with a 1% tolerance.
2. The attacker front-runs with a swap that buys ETH, pushing the cursor up ~50 bps. The active bin is now ETH-poor / USDC-rich, so the probe returns a skewed `need0` / `need1`.
3. The victim's add executes at the skewed ratio. The over-represented side hits its cap, so `_scaleWeightsToShares` scales the deposit down: the LP receives a partially-filled (~6700 USD), USDC-skewed position rather than the intended balanced 10000 USD.
4. The attacker back-runs, selling the ETH to restore the cursor to the fair mid. Restoring the now-deeper book sells slightly more ETH than was bought, and the surplus is paid out of the victim's mis-placed deposit.
5. The LP withdraws and is short of what they deposited (valued at the fair mid). The shortfall is captured by the attacker, minus the bid/ask spread paid on the round-trip.

This is a real victim-to-attacker transfer drawn from the victim's deposited capital, not a conservation-neutral rounding artifact, and it is profitable whenever the extracted impermanent loss exceeds the round-trip spread.

The victim's add is the entire profit source, with the victim's loss equal to the attacker's gain plus the round-trip cost. Extraction scales with how far the attacker can push the cursor, and an LP that spreads liquidity across several bins (the common case for a 50/50 add) lets the attacker push much further: the cursor crosses whole bins, flipping bin 0 from 50/50 to one-sided and converting the off-bins. For a 10000 USD aggregate 50/50 add with a 1% cap tolerance, ETH/USDC, 200 bps bins:

Victim 50/50 add	Attacker push	Victim loss	Attacker net @ 0.01 bps	@ 0.03 bps
single active bin	0.50%	4.19 USD	+3.79 USD	+3.73 USD

Victim 50/50 add	Attacker push	Victim loss	Attacker net @ 0.01 bps	@ 0.03 bps
three bins (-1, 0, +1)	1.0%	12.55 USD	+11.00 USD	+10.88 USD
three bins (-1, 0, +1)	1.5%	22.61 USD	+21.02 USD	+20.84 USD
three bins (-1, 0, +1)	2.0%	33.44 USD	+31.83 USD	+31.59 USD

A 200 bps bin lets the attacker push a full ~2% (one bin width) before the cursor leaves the spanned range, so a three-bin 50/50 add loses tens of USD per 10000 USD deposited, roughly 8x the single-bin case. The larger push is also more spread-robust: the extracted impermanent loss grows quadratically with the push while the attacker's round-trip spread cost grows only linearly, so the 2% three-bin sandwich stays profitable up to ~2.5 bps of spread (net +19.79 USD at 1 bps, breakeven near 2.5 bps), where the single-bin 0.5% push is already unprofitable by ~1 bps. The attack is permissionless, atomic, and repeatable.

Impact

Conditional loss of LP funds via an atomic, permissionless MEV sandwich of `addLiquidityWeighted`, bounded by the victim's outlay. The victim cannot defend by tightening `max0` / `max1`, because those cap spend, not the placement price, so a tight tolerance only yields a smaller, equally-mispriced position (unlike `addLiquidityExactShares`, where tight caps do fail closed). Magnitude scales with bin width and the number of bins the LP spans: a 50/50 add across three 200 bps bins loses tens of USD per 10000 USD deposited and stays profitable up to a few bps of spread. The attack works on thin-spread pools, which the protocol permits and which have no mandatory price-velocity guard.

Code Snippet

<https://github.com/Metric-OMM/metric-periphery/blob/90039f9b68f6b2253425acfb6497fcf38e28cc17/contracts/MetricOmmPoolLiquidityAdder.sol#L74-L96>
<https://github.com/Metric-OMM/metric-periphery/blob/90039f9b68f6b2253425acfb6497fcf38e28cc17/contracts/MetricOmmPoolLiquidityAdder.sol#L199-L216>

Tool Used

Manual Review

Recommendation

Give the liquidity adder an output-side guard so an LP can bound the *position they receive*, not only the tokens they spend: a minimum-shares (or minimum-total-liquidity)

floor, an expected-cursor or composition band the caller supplies and the add reverts outside of. This lets an honest LP fail closed when the pool has been moved off the price they signed for, instead of silently depositing at the manipulated composition.

Discussion

Oxklapouchy

Fix confirmed in [PR#29](#)

Issue M-2: Non-atomic Pool Deployment Enables Admin Takeover and DoS

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/86>

Summary

The pool deployment process does not pass the admin parameter to the deployed contract, instead relying on a factory-side mapping for admin storage. This separation creates a window where the pool exists but enabling denial of service through race conditions or allowing attackers to interfere with pool initialization.

Vulnerability Details

When a new pool is created through the factory, the deployment and admin assignment occur in separate steps:

```
pool = MetricOmmPoolDeployer(poolDeployer)
    .deploy(
        MetricOmmPoolDeployer.DeployParams({
            salt: params.salt,
            factory: address(this),
            token0: params.token0,
            token1: params.token1,
            priceProvider: params.priceProvider,
            depositAllowlistProvider: params.depositAllowlistProvider,
            swapAllowlistProvider: params.swapAllowlistProvider,
            // ... other parameters ...
        })
    );

poolAdmin[pool] = params.admin;
priceProviderTimelock[pool] = params.priceProviderTimelock;
```

The DeployParams struct passed to the deployer does not include an admin field. The admin address is not communicated to the newly deployed pool contract. Instead, it is recorded only in the factory's poolAdmin mapping after the pool already exists on-chain.

This design creates a gap where the pool is deployed and operational before its administrative authority is established in the factory's state. Functions that rely on pool admin verification check against the factory mapping:

```
function _checkPoolAdmin(address pool) private view {
    if (msg.sender != poolAdmin[pool]) revert NotPoolAdmin();
}
```

```
modifier onlyPoolAdmin(address pool) {  
    _checkPoolAdmin(pool);  
    _;  
}
```

Impact

An attacker can create a race condition where they deploy a pool with a specific salt before the legitimate pool creator does. If the deployer call and the admin mapping update are not atomic, an attacker can then invoke pool admin functions by first establishing themselves as the admin in the factory mapping. Additionally, if multiple transactions attempt to create pools with identical parameters, the first transaction to execute could assign itself as admin, causing the second transaction to either revert or operate under the wrong admin context. This enables denial of service attacks that either prevent legitimate pool creation or cause newly created pools to operate under attacker-controlled admin privileges.

Recommendation

Pass the admin address as part of the DeployParams struct to the pool deployer, and configure the pool to store this value as an immutable or state variable. This ensures the pool is aware of its administrator at construction time rather than relying on external factory state.

Discussion

Oxklapouchy

Fix confirmed in [PR#54](#)

Issue M-3: SwapMath sell-token0 closed form divides its denominator by 2^{64} , mispricing the swap against the trader

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/96>

Summary

`computeAnalyticalTargetPosForSellToken0` builds its closed-form denominator from two terms that must share the $Q_{64.64}$ (price) scale. The `token1`-balance term is wrongly divided by `ONE_X64` (`Math.ceilDiv(token1Balance * (ONE_X64 + feeX64), ONE_X64)`), which drops the 2^{64} scaling and leaves it amount-scaled instead of price-scaled. The denominator collapses to roughly the second (correctly price-scaled) term, the analytical target position is wrong, and the average price `buyToken1InBinSpecifiedIn` derives from it is biased against the trader, who pays more `token0` per `token1` received than the bin curve dictates.

Vulnerability Detail

The function documents the closed form (see its `NatSpec`) as:

$$d = (in_0 * c * P_c) / (T_1 * (1 + fee/264) * 264 + in_0 * c * \Delta P / (2M))$$

so the first denominator term is $T_1 * (1 + fee/264) * 264 = T_1 * (264 + fee)$, i.e. `token1Balance * (ONE_X64 + feeX64)`, expressed in $Q_{64.64}$ units. The implementation instead computes:

```
uint256 denominator = Math.ceilDiv(token1Balance * (ONE_X64 + feeX64), ONE_X64);
```

The extra `/ ONE_X64` divides this term by 2^{64} , turning it into `token1Balance * (1 + fee/264)`, which is approximately `token1Balance`, an amount-scaled value. The second term, added immediately after,

```
denominator += Math.mulDiv(inputAmount, currBinPos * deltaPriceX64, 2 *  
    ↪ MAX_POS_BIN, Math.Rounding.Ceil);
```

is correctly price-scaled because it carries `deltaPriceX64`, a $Q_{64.64}$ value. The two terms no longer share a scale. Since the `token1` term is normally the dominant one, dividing it by 2^{64} makes it negligible and collapses the denominator to roughly the second term alone.

A too-small denominator makes `deltaPos = Math.mulDiv(inputAmount, currBinPos * priceAtCurrPosX64, denominator)` too large, so `targetPos = currBinPos - deltaPos` is pushed too far down the bin. `buyToken1InBinSpecifiedIn` then derives the average

position and price from this corrupted target and converts it into the token0 the trader must pay:

```
avgPos    = (currBinPos + targetPos) / 2;
avgPrice  = calculatePriceAtBinPosition(lowerPriceX64, upperPriceX64, avgPos,
    ↪  Math.Rounding.Floor);
in0WithoutFeeScaled = calculateRequiredToken0(out1Scaled, avgPrice);    // = out1 *
    ↪  2^64 / avgPrice
```

Because `targetPos` is too low, `avgPos` and `avgPrice` are too low, and `calculateRequiredToken0` (which divides by `avgPrice`) returns a `token0` input that is too high for the `token1` delivered. On the exact-input buy-token1 (sell-token0) path the trader is therefore charged more `token0` per `token1` than the bin's price curve specifies. This path is taken whenever the trader's input is the binding constraint inside a bin, a partial fill that neither reaches the price limit nor exhausts the bin's `token1`, which is the ordinary case for normal-sized swaps.

Impact

Traders selling `token0` for `token1` receive a worse-than-correct average price on the in-bin partial-fill path: they pay more `token0` per `token1` than the pool's curve dictates. The mispricing is deterministic and applies to every qualifying swap, accruing to the pool and LPs at the trader's expense.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/6aa6c3b489b84b8b1c50dc6a4967184df17aa395/contracts/libraries/SwapMath.sol#L313>

Tool Used

Manual Review

Recommendation

Keep the `token1`-balance term in $Q_{64.64}$ (price) scale so both denominator terms share a scale and match the documented closed form: drop the erroneous division by `ONE_X64` and use `token1Balance * (ONE_X64 + feeX64)` directly. This restores the dominant denominator term and yields the correct analytical target position and average price.

Discussion

0xklapouchy

Fix confirmed in [PR#58](#)

The specific code this issue quotes, the standalone `denominator = Math.ceilDiv(token1Balance * (ONE_X64 + feeX64), ONE_X64);` ... block inside `computeAnalyticalTargetPosForSellToken0`, no longer exists.

It wasn't patched directly; it was removed as part of **PR#58**, which fixed "**Average price cannot be used for token1 → token0 conversion when token1 is distributed equally**" by reworking how sell-token (token1 to token0) pricing is computed throughout `SwapMath.sol`: the old arithmetic-mean-price division was replaced everywhere with an invert-both-endpoints / average-the-inversions / multiply approach (harmonic mean), and `computeAnalyticalTargetPosForSellToken0` was rewritten to mirror and delegate to the already-verified `computeAnalyticalTargetPosForBuyToken0` rather than keep its own independent closed form.

Issue M-4: Token scale multipliers can break the documented decimal invariant

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/101>

Summary

`_getScaleMultipliers` derives multipliers from `IERC20Metadata.decimals()` with no upper bound. Tokens with decimals above 18 or extreme decimal spreads can produce multipliers far beyond what the pool assumes, and downstream products in `createPool` can overflow or mis-scale initial share amounts.

Vulnerability Detail

During `createPool`, `_getScaleMultipliers` sets `internalDecimals` to the maximum of 18 and both tokens' reported decimals, then computes each multiplier as $10^{(\text{internalDecimals} - \text{tokenDecimals})}$ without validating the return value of `decimals()`. The pool documents these multipliers in `NatSpec`, an assumption that scaling stays within roughly 10^{18} :

```
/// @notice Multiplier to scale token0 external amounts to internal: 10^(max(18,
↪ decimals) - token0.decimals())
uint256 internal immutable TOKEN_0_SCALE_MULTIPLIER;
/// @notice Multiplier to scale token1 external amounts to internal: 10^(max(18,
↪ decimals) - token1.decimals())
uint256 internal immutable TOKEN_1_SCALE_MULTIPLIER;
```

A token reporting more than 18 decimals (or a pair with a very large decimal gap) can produce much larger multipliers. Those values feed directly into `initialScaledAmount0PerShareE18` and `initialScaledAmount1PerShareE18`, so creation may revert on overflow or the pool may start with internal amounts that no longer match the intended precision model.

Impact

Broken pool creation or incorrect scaling for early liquidity providers depending on token set.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/6aa6c3b489b84b8b1c50dc6a4967184df17aa395/contracts/MetricOmmPool.sol#L43-L46>
<https://github.com/Metric-OMM/metric-core/blob/6aa6c3b489b84b8b1c50dc6a4967184df17aa395/contracts/MetricOmmPoolFactory.sol#L625-L637>

<https://github.com/Metric-OMM/metric-core/blob/6aa6c3b489b84b8b1c50dc6a4967184df17aa395/contracts/MetricOmmPoolFactory.sol#L150-L158>

Tool Used

Manual Review

Recommendation

Enforce supported decimal ranges at pool creation and align factory scaling with the pool's documented precision assumptions.

Discussion

Oxklapouchy

Acknowledged only via documentation.

`_getScaleMultipliers/_validatePoolParameters` are unchanged, and no decimals bound was added anywhere in `MetricOmmPoolFactory.sol`. The only change is a `@dev` warning added to `createPool` in `IMetricOmmPoolFactory.sol` noting that exotic decimal spreads may revert or mis-scale, but the recommended fix (enforce supported decimal ranges) was not implemented.

Issue M-5: Price Field Overflow in Packed Stream Format Corrupts feedId for Prices Above ~43 USDT

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/103>

Summary

The `_verifyAndStream` path packs each feed's result into a single `uint256` word. In this layout, `normPrice` is given only 32 bits of space between the `spread` field (at bit offset 96) and the `feedId` field (at bit offset 128). However, `normPrice` is a `uint64` value expressed with `TARGET_DECIMALS = 8`. Any price greater than 42.94967295 ($(2^{32} - 1) / 10^8$) requires more than 32 bits, so its high bits overflow into the `feedId` field. Because the fields are combined with bitwise OR, both `normPrice` and `feedId` are corrupted in the packed output.

Vulnerability Details

`normPrice` is produced with 8 decimals of precision:

```
int256 private constant TARGET_DECIMALS = 8;
```

```
rawPrice = (pU * scale) / usdtRateU;  
...  
normPrice = rawPrice.toUint64();
```

The value is then packed into a `uint256` together with `feedId`, `spreadU`, and `tsPart`:

```
values[i] =  
    (uint256(feedId) << 128) |  
    (uint256(normPrice) << 96) |  
    (spreadU << 80) |  
    tsPart;
```

The field positions are:

- `feedId` at bit offset 128
- `normPrice` at bit offset 96
- `spread` at bit offset 80
- `tsPart` at bit offset 0

This leaves bits [96, 127] – exactly 32 bits – for `normPrice` before the `feedId` field begins at bit 128.

The corresponding `_unpack` confirms this intended layout:

```
function _unpack(
    uint256 packed
) internal pure returns (uint32 feedId, uint64 price, uint16 spread, uint64 ts) {
    assembly {
        feedId := shr(128, packed)
        price := shr(96, packed)
        spread := and(shr(80, packed), 0xFFFF)
        ts := and(packed, 0xFFFFFFFFFFFFFFFF)
    }
}
```

Because `normPrice` carries 8 decimals, the maximum price that fits in 32 bits is:

$$(2^{32} - 1) / 10^8 = 4,294,967,295 / 10^8 = 42.94967295$$

For any price above 42.94967295 USDT, `normPrice` occupies more than 32 bits, and the bits above position 127 are OR-ed into the `feedId` field. For example, a price of 3000.00000000 yields `normPrice` = 300000000000, which requires 39 bits, so 7 high bits overflow into `feedId`.

On unpack, `price := shr(96, packed)` reads a 64-bit value spanning bits [96, 159], which overlaps the `feedId` field at bits [128, 159], and `feedId := shr(128, packed)` reads the bits that were partly written by the overflowing `normPrice`. Both returned values are therefore incorrect.

Impact

For any feed whose normalized price exceeds 42.94967295, the packed `uint256` produced by `_verifyAndStream` contains a corrupted `normPrice` and a corrupted `feedId`. Consumers that unpack this value receive an incorrect price and an incorrect feed identifier.

This affects the `_verifyAndStream` output only. The `_verifyAndStore` path stores `normPrice` in the full-width `OracleData.price (uint64)` field and is not affected.

Recommendation

Allocate a full 64 bits for `normPrice` in the packed layout and shift the higher fields up accordingly, then update `_unpack` to match. For example:

```
values[i] =
    (uint256(feedId) << 192) |
    (uint256(normPrice) << 128) |
    (spreadU << 112) |
    tsPart;
```

```

function _unpack(
  uint256 packed
) internal pure returns (uint32 feedId, uint64 price, uint16 spread, uint64 ts) {
  assembly {
    feedId := shr(192, packed)
    price := and(shr(128, packed), 0xFFFFFFFFFFFFFFFF)
    spread := and(shr(112, packed), 0xFFFF)
    ts := and(packed, 0xFFFFFFFFFFFFFFFF)
  }
}

```

The exact offsets should be chosen based on the actual width required by `tsPart` (currently masked to 72 bits), ensuring no two fields overlap. Note also that `feedId` and `price` should be masked on unpack so that adjacent fields cannot leak into the returned values.

Discussion

horror-bun

fixed at <https://github.com/Oracle-Based-Pool/smart-contracts-poc/commit/6e0ff65524c381a4cb113a0c0d8ebcb243ae1971> by removing unused faulty function

0xklapouchy

Fix confirmed in [6e0ff65](#)

Issue M-6: Average price cannot be used for token1 → token0 conversion when token1 is distributed equally

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/106>

Summary

Going from token0 to token1 using a single average price is correct. Going the other way, from token1 to token0 with that same average price, is not.

When token1 is assumed to be spread equally across the positions and the price is a sloped line, the amount of token0 backing a range of token1 positions is a sum of reciprocals of the prices. That sum is governed by the harmonic mean of the prices, not by one over the average price. The conversion helper that turns token1 into token0 divides by the average price, so it always returns a bit less token0 than the true amount. The error is zero only when the price is flat and grows with the width of the bin.

The code is not broken line by line. The mismatch is mathematical: `calculateRequiredToken0` uses the wrong summary statistic for the token1→token0 direction.

Vulnerability Details

There are two conversion helpers:

```
// token0 -> token1 : multiply by price
function calculateRequiredToken1(uint256 token0Amount, uint256 avgPriceX64)
    internal pure returns (uint256 requiredToken1)
{
    requiredToken1 = Math.ceilDiv(token0Amount * avgPriceX64, ONE_X64);
}

// token1 -> token0 : divide by price
function calculateRequiredToken0(uint256 token1Amount, uint256 avgPriceX64)
    internal pure returns (uint256 requiredToken0)
{
    requiredToken0 = Math.ceilDiv(token1Amount << 64, avgPriceX64);
}
```

Take a tiny bin of three positions, each holding 10 units of liquidity, price going up by 1 each step (p, p+1, p+2).

Going token0 → token1 you multiply by price, and it works:

```
10*p + 10*(p+1) + 10*(p+2)
= 10*(p + p+1 + p+2)
```

$$\begin{aligned}
 &= 10*(3p + 3) \\
 &= 30*(p + 1)
 \end{aligned}$$

$p+1$ is the average price, so multiplying the total token0 by the average price gives the right token1. The plain average is the correct summary here.

Going the other way, token1 → token0, you divide by price, and it does not work:

$$\begin{aligned}
 &10/p + 10/(p+1) + 10/(p+2) \\
 &= 10*(1/p + 1/(p+1) + 1/(p+2))
 \end{aligned}$$

This cannot be turned into $30/(p+1)$. You cannot pull $p+1$ out of a sum of reciprocals. The right factor is the harmonic mean of the prices, not one over the average price, and the average of $1/\text{price}$ is always bigger than $1/\text{average price}$ when the price varies. So `calculateRequiredToken0` with the average price always returns a bit less token0 than the true amount.

That is the reason a quadratic relation is needed on the token1 side. Summing token1 against a linear price gives a quadratic, and undoing it (token1 → token0) is not a single division by the mid price.

The under-count of token0 shows up in both swaps that use this conversion, but who pays for it depends on the direction:

- **Going down, with buyToken1 specified out**, token0 is the trader input into the bin. The bin receives less token0 than fair, and later when that token0 is sold back out the bin cannot return all the token1 it owes, so the pool loses. The wiring for that path:

```

// buyToken1...Out (down): linear position, then divide-by-mid-price for the
↪ input
finalBinPos = calculateBinPositionAfterSellingAmount1( ... );           //
↪ even-spread move
uint256 avgPos = (currBinPos + finalBinPos) / 2;
uint256 avgPriceX64 = calculatePriceAtBinPosition(lowerPriceX64,
↪ upperPriceX64, avgPos, Math.Rounding.Floor);
uint256 amountInScaled = calculateRequiredToken0(amountOutScaled,
↪ avgPriceX64); // token1 -> token0

```

Impact

The size of the per-swap error is the gap between the average price and the harmonic price, and it grows with how wide the bin is.

For a very narrow bin it is dust. For wider bins it is percent-level. It also adds up over repeated swaps, so even a narrow bin can drift in a meaningful way over many trades.

Recommendation

Using logarithmic mean of the price range will solve this problem.

Appendix

Each position holds equal liquidity, price is linear in position, and token0 out per position is (token1 at that position) / (price there). In the continuous limit (your uint104.max positions), the sum becomes an integral. Let ρ = token1 per unit position (constant), $P(t) = P_1 + b t$ with slope b = price per position:

$$\text{token0 out} = \int_{t_1}^{t_2} \frac{\rho dt}{P_1 + b t} = \frac{\rho}{b}$$

$$\ln P_2 \overline{P_1}$$

Now $\frac{\rho}{b} = \frac{\rho(t_2 - t_1)}{b(t_2 - t_1)} = \frac{\text{total token1 in range}}{P_2 - P_1}$, so everything position-related cancels:

$$\text{token0 out} = \frac{Q}{P_2 - P_1} \ln P_2 \overline{P_1}$$

where Q = total token1 swept, P_1 = price at range start, P_2 = price at range end. The position count never appears – uint104.max or 5, same formula.

Interpretation

$\frac{P_2 - P_1}{\ln(P_2/P_1)}$ is the **logarithmic mean** of the two endpoint prices, so this is just

$$\text{token0 out} = \frac{Q}{\text{logmean}(P_1, P_2)}$$

Discussion

konrad-metric

I acknowledge that.

but the calculation will get complex as we also need to find the price at the end of the swap (P_{cut}). am I right?

For example: ETH/USD pool Lets assume we have 10ETH and current bin lower price is 1000, and current price is 1100. (the bin upper price is not important).

user wants to buy x (example 3) ETH. we need to first calculate the P_{cut} and then the P_{avg} .

Step 1: Calculate Maximal Price (P_{cut}) The cutoff (maximal) price depends entirely on the percentage of the total pool being bought ($\frac{x}{E_{\text{total}}}$).

$$P_{\text{cut}} = P_{\min} \cdot \left(\frac{P_{\max}}{P_{\min}} \right)^{\frac{x}{E_{\text{total}}}}$$

Example Calculation:

- **Inputs:** $P_{\min} = 1000$, $P_{\max} = 1100$, $E_{\text{total}} = 10$, $x = 3$
- **Math:** $1000 \cdot (1.1)^{3/10} = \mathbf{\$1,029.01}$

Step 2: Calculate Average Price (P_{avg}) Because the asset distribution is inherently logarithmic, the effective average price between the starting price and the cutoff price is their exact **Logarithmic Mean**.

$$P_{\text{avg}} = \frac{P_{\text{cut}} - P_{\min}}{\ln(P_{\text{cut}}) - \ln(P_{\min})}$$

Example Calculation:

- **Inputs:** $P_{\min} = 1000$, $P_{\text{cut}} = 1029.01$
- **Math:** $\frac{1029.01 - 1000}{\ln(1029.01) - \ln(1000)} = \frac{29.01}{0.02859} = \mathbf{\$1,014.58}$

am I getting it correctly?

DemoreXTess

Hey @konrad-metric, $P(\text{cut})$ should not change like that if we don't change price linearity. $P(\text{cut})$ should remain same and should be calculated based on end-up position. Step2 is also not fully correct but logic is good because we only face with this issue while buying token1, therefore bigger price is current price comparing to end-up. Instead it should be like that:

$$P_{\text{avg}} = \frac{P_{\text{current}} - P_{\text{target}}}{\ln(P_{\text{current}}) - \ln(P_{\text{target}})}$$

Calculation for "specified output" is very simple I shared it under #105. However, "specified in" functions becomes really hard to calculate when we buy token1. Maybe Newtonian search can be used to solve it because newtonian is really helpful for this kind of hard to find solutions.

konrad-metric

@DemoreXTess

what do you think about using harmonic mean?

in our current code we use arithmetic mean and we divide by it what doesn't work. you suggested dividing by logarithmic mean.

What if we take mean of inverse of prices and we multiply by it? This is inverse of harmonic mean. So if we would like to divide by inverse of it we would be comparing dividing by arithmetic mean, log mean and harmonic mean.

$$(\frac{1}{p_{current}} + \frac{1}{p_{target}})/2$$

because the following holds: $AM \geq LM \geq HM$, the division by HM would result in highest amount to be paid by user.

It also quite easy to follow and understand.

DemoreXTess

@konrad-metric

If we use that it will overcharge trader for sure. We need to see how much exactly. I will come back to you after testing this pricing method.

DemoreXTess

The numbers below come from `test_enumerate_out_only` in `SwapMathEnumerateOutOnlyAuditTest`. Each run starts a single ETH/USDC bin at the top of its range holding 100,000 USDC, sells ETH into the bin, runs a batch of randomized up and down middle swaps, then fully drains the ETH back out. Net ETH flow is zero and the fee is zero, so in a perfectly fair pool the bin should end with exactly its starting USDC.

Each cell reports the worst max GAIN (in USDC) over the full selector and percentage sweep. The two knobs varied are the number of middle swaps (3, 10, 50) and the bin upper price tick (2010, 2050, 2100, with the lower tick fixed at 2000). We should also note that gains from logarithmic mean comes from #105 which means it's not the logmean's fault.

Logarithmic mean

middle swaps	upper 2010	upper 2050	upper 2100
3	0.3132	7.6699	29.9114
10	0.5244	12.8412	50.0825
50	1.2964	31.7644	123.9836

Harmonic mean

middle swaps	upper 2010	upper 2050	upper 2100
3	0.9363	22.9313	89.4509
10	1.4567	35.6685	139.1164
50	3.2900	80.6059	314.7323

While tests I used following calculation for buy token1 specified output function:

```

uint256 pA = calculatePriceAtBinPosition(lowerPriceX64, upperPriceX64, currBinPos,
    ↪ Math.Rounding.Floor);
uint256 pB = calculatePriceAtBinPosition(lowerPriceX64, upperPriceX64, finalBinPos,
    ↪ Math.Rounding.Floor);
pA = (1 << 128) / pA;
pB = (1 << 128) / pB;
uint256 avgPrice = (pA + pB) / 2;
// calculateRequired1 will be used intentionally because we will multiply with this
    ↪ price
uint256 amountInScaled = calculateRequiredToken1(amountOutScaled, avgPrice);

```

christos-metric

@ DemoreXTess to be clear, when you say Each cell reports the worst max GAIN (in U SDC) do you mean that there are still ways, even with the new math that the pool can be drained? because we are saying the opposite on #105 right?

DemoreXTess

Hey @christos-metric , no I don't mean that. I mean all of these are pool's unfair gain against the traders. Yes, we're always on pool's side but ofcourse I care about traders too that's why I said "worst max GAIN" ☒

christos-metric

@ DemoreXTess okay understood thank you! That was the only way this could make sense lol

i was curious, for the LM case, shouldnt the gain be exactly 0? given that we are performing the math correctly with LM, which means there is no loss in the conversions, how can it be that there is a gain? what am i missing?

DemoreXTess

It comes from #105, because we didn't make any change for #105 because price provider may provide another price comparing to previous price and our liquidity distribution becomes broken again:

1. New price is higher than previous

Now, token0 is more valueable our L/M = A defition from desmos becomes broken because token1 liquidity won't be enough to apply "A" rule because token1 is less valueable comparing to token0

2. New price is lower than previous

In this case, it's opposite. Token1 liquidity has surplus and when we reach position 0 we can't consume all token1 liquidity.

How #105 causes pool value increase ?

When we compare our current model against the one I built in Desmos, we notice a key difference: every time a trader buys token0, the token1 they pay in gets deposited at

lower price points. I call this a "rebalance" because the protocol is literally redistributing liquidity, pushing it down toward lower price points. As a result, traders now have to spend more token0 to buy that token1 back, since it's sitting at lower price points. The same thing happens in reverse. Once token1 is fully consumed (i.e., traders have supplied enough token0 to absorb it all), the pool ends up with surplus token0 instead. The model then rebalances on the token0 side, placing that excess liquidity at higher price points.

Therefore, it generates more revenue to LPs by rebalancing.

DemoreXTess

By the way, it sounds like a feature. I think you can use it in marketing because I can't find a way to fix #105 properly due to oracle based market maker's own mechanics ☒

Issue M-7: Notional fee overcharged on gross input

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/111>

Summary

When someone does an exact-output swap (they specify how much they want to receive), the notional fee is taken from the gross input they have to pay. The problem is that this gross input already has the per-bin buy/sell fee and the oracle base fee baked into it. So the notional fee ends up being a fee charged on top of other fees. The practical result is that the real notional fee rate is no longer the `notionalFeeE8` you configured – it quietly goes up whenever a bin's add fee is higher or the oracle spread is wider. Exact-input swaps don't have this problem, so the two swap modes charge differently for the same trade.

Vulnerability Details

The notional fee is applied near the end of `_executeSwap`, after the swap math has already figured out the input and output amounts.

For exact-input, the fee comes out of the output – the clean amount the trader actually receives:

```
uint256 notionalFeeScaled = uint256(-amount1DeltaScaled) * notionalFeeE8 / 1e8;
```

That's fine: it's just `f` times whatever the trader got.

For exact-output, the fee is grossed up off the input instead:

```
uint256 notionalFeeScaled = uint256(amount0DeltaScaled) * notionalFeeE8 / (1e8 -  
↪ notionalFeeE8);  
amount0DeltaScaled = amount0DeltaScaled + int256(notionalFeeScaled);
```

The catch is what `amount0DeltaScaled` is here – it's the gross input the trader pays, and the swap math builds that input fee-inclusive. In `SwapMath` the per-bin input is `netNotional * (1 + baseFee + addFee)`:

```
uint256 onePlusSell = ONE_X64 + currBinSellFeeX64; // = baseFee + addFeeSell  
uint256 totalIn0Scaled = grossInputWithBinFeeCeil(in0WithoutFeeScaled, onePlusSell);
```

So the real economic notional of an exact-output swap is just the fixed output the trader asked for, and the notional fee should be a flat `f` of that, no matter what the bin or oracle fees are. But because the code computes it from the gross input, the fee base is inflated by `(1 + baseFee + addFee)`. Put the two side by side:

- exact-input: $\text{fee} = f \times \text{actual output} \rightarrow \text{independent of bin/base fees}$

- exact-output: $\text{fee} \propto f \times \text{output} \times (1 + \text{baseFee} + \text{addFee}) \rightarrow$ grows with the bin add fee and the oracle spread

Concretely: if an admin bumps up one bin's add fee, or the oracle spread widens, the notional fee collected on exact-output swaps through that bin goes up too – even though nobody touched `notionalFeeE8`. The extra amount increases what the trader pays and lands in the notional fee pot.

Impact

This only affects exact-output swaps. The effective notional rate becomes roughly $\text{notionalFeeE8} \times (1 + \text{baseFee} + \text{addFee})$ instead of the configured rate.

The size of the overcharge per trade is about $\text{notionalFee} \times (\text{baseFee} + \text{addFee})$ – a small, second-order amount since it's one small fee fraction multiplied by another. It's bounded by the bin add fee (a uint16 in E6, so at most ~6.55%) and the oracle base fee.

As for who it touches: exact-output traders pay a bit more than the configured notional rate would suggest, and that extra goes into the notional fee pot, which `collectFees` later hands to the protocol and the pool admin. LPs and exact-input traders are not affected. The same trade also costs a different notional fee depending on whether it's sent as exact-input or exact-output, with exact-output coming out higher.

Recommendation

Charge the notional fee on the fee-exclusive notional in both directions, so it stops compounding on the per-bin add fee and the oracle base fee. For exact-output that means using the pre-bin-fee notional (effectively the specified output amount) as the base instead of the gross input.

To do this, have the swap math functions (`buyToken*InBinSpecified*`) also return the accumulated pre-fee notional, and use that value as the fee base in `_executeSwap` rather than the fee-inclusive input/output deltas. The net notional can't be recovered after the fact at the point the fee is applied, because the add fee differs from bin to bin within a multi-bin swap, so it needs to be tracked inside the swap loop where the per-bin net amount is known.

Issue M-8: Providers missing `getTokens()`

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/113>

Summary

`ProtectedPriceProvider` and `AnchoredPriceProvider` declare that they implement `IPriceProvider`, but they expose the token pair as `token0()` / `token1()`. The metric-core factory, which is what actually consumes a price provider, expects the core `IPriceProvider` with `getTokens()`. The two interfaces don't line up, so when the factory tries to validate one of these providers it calls a function the provider doesn't have, and the call reverts. As a result a pool cannot be created (or repointed) with these providers.

Vulnerability Details

The core interface and factory require `getTokens()`:

```
// metric-core/contracts/interfaces/IPriceProvider/IPriceProvider.sol
function getTokens() external view returns (address baseToken, address quoteToken);

// metric-core/contracts/MetricOmmPoolFactory.sol (_validatePriceProvider)
(address baseToken, address quoteToken) = IPriceProvider(priceProvider).getTokens();
if (baseToken != token0 || quoteToken != token1) revert
→ PriceProviderTokenMismatch();
```

`_validatePriceProvider` runs on pool creation and on every price-provider update, so it is on the critical path for using a provider at all.

The providers, however, expose the pair under different names and implement a *different* `IPriceProvider` (the one defined inside `smart-contracts-poc`, which uses `token0()` / `token1()`):

```
// smart-contracts-poc/contracts/ProtectedPriceProvider.sol
contract ProtectedPriceProvider is IPriceProvider { ... }
function token0() external view override returns (address) { ... }
function token1() external view override returns (address) { ... }
// no getTokens()

// smart-contracts-poc/contracts/AnchoredPriceProvider.sol - same shape
```

Because the providers have no `getTokens()` selector and no fallback that could answer it, the factory's `IPriceProvider(priceProvider).getTokens()` call reverts. (Even a fallback wouldn't help: `getTokens()` is decoded as two return addresses, so an empty return would fail decoding.)

The same mismatch affects the other providers in the package (`PriceProvider`, and the L2 variants), since none of them implement `getTokens()` either.

Impact

A pool cannot be deployed with `ProtectedPriceProvider` or `AnchoredPriceProvider`: `createPool` reverts inside `_validatePriceProvider`. The same revert blocks pointing an existing pool at one of these providers. In other words, the production price providers in `smart-contracts-poc` cannot be wired to metric-core pools as written. No funds are involved – it is a deploy-time/integration break that surfaces on the first attempt to use one of these providers.

Recommendation

Align the provider interface with the one metric-core actually calls. Since the providers already hold the pair, add `getTokens()` returning the same values:

```
function getTokens() external view returns (address baseToken, address quoteToken) {  
    return (token0(), token1());  
}
```

Apply it to every provider intended for use with metric-core (`ProtectedPriceProvider`, `AnchoredPriceProvider`, `PriceProvider`, and the L2 variants), or unify the `smart-contracts-poc` `IPriceProvider` with the metric-core `IPriceProvider` so the two packages share one definition.

Discussion

Oxklapouchy

Fix confirmed in [PR#57](#)

Issue M-9: LP value leaks to arbitrageurs whenever the oracle lags the market, and the bin spread cannot cover the gap [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/120>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The pool prices every swap against the oracle-provided mid ($\text{sqrt}(\text{bid} * \text{ask})$) and charges a fee built from the oracle bid/ask spread (baseFeeX64) plus per-bin (addFeeBuyE6 / addFeeSellE6) and notional (notionalFeeE8) fees. Whenever the on-chain oracle price lags the real off-chain market and the round-trip fee is smaller than that lag gap, an arbitrageur trades against the lagged quote and unwinds after the price catches up, netting the difference from LPs. This is structural LVR (Loss-Versus-Rebalancing), a known property of every oracle-priced AMM. The protocol's own operating strategy (push fresh prices every block, tight staleness, spreads sized to each pair) correctly mitigates it for the pools the team runs. The issue remains valid because pool creation is permissionless: pools can be created with any bin width, any spread/fees, any price provider, and any freshness bound, and in those pools the spread can fail to cover the lag and the leak becomes extractable.

Vulnerability Detail

The pool derives the execution price and the base spread from a single oracle quote:

```
midPriceX64 = Math.sqrt(bidPriceX64 * askPriceX64);
baseFeeX64  = Math.mulDiv(askPriceX64, ONE_X64, midPriceX64, Ceil) - ONE_X64; // =
↳ ask/mid - 1
```

Freshness is bounded only by $\text{MAX_TIME_DELTA}(\text{isStale}: (\text{now} - \text{refTime}) > \text{maxDelta})$. Within that window, and for moves smaller than the updater's trigger, the on-chain price still lags the market. The fee charged is baseFeeX64 plus the per-bin additional fee (admin-set, capped near 6.55%) plus the notional fee; spreadFeeE6 only splits the spread fee between LPs and the protocol. Crossing the bin curve is price impact that is recovered on the reverse leg, so the bin width itself is not a round-trip cost; the only round-trip cost is the fee. Because bins are stored by distance from the oracle mid ($\text{absolute price} = \text{mid} * (1 \pm \text{dist}/1\text{e}6)$), an oracle mid jump shifts the whole grid by the full jump while the cursor distance is preserved. The arbitrageur captures that grid shift and pays the fee only on the size he trades, not on the jump.

The leak is profitable whenever the inter-update price move exceeds the round-trip fee. Two extraction modes, depending on the price provider:

1. Pushable provider (Pyth Lazer / Chainlink, updated by submitting a signed report). Atomic sandwich: swap on the current not-yet-stale price, submit the fresh report to move the mid in the arbitrageur's favor, swap back, all in one transaction. Cost is only the round-trip fee.
2. Keeper-updated provider (the compressed oracle, where prices cannot be pushed on demand). Natural-drift: the arbitrageur observes the real market, sees the keeper's on-chain price lagging, swaps against the lagged quote, and unwinds once the keeper catches up.

In both, profit per round trip is approximately $\text{inter_update_move} - \text{round_trip_fee}$. If the reported bid/ask spread plus $\text{addFeeBuy}/\text{addFeeSell}$ do not exceed the update gap, the difference is net profit taken from LPs. The protocol's `SwapMathAnalysis.md` derives a minimum fee $f_{\min}(S) = 1 - \sqrt{2/(S+1)}$ (about 25% of bin width), but that bound only prevents in-bin round-trip drainage from the curve path dependency; it does not address the oracle-to-market gap.

This is the expected OMM trade-off, and the correct defense is fresh prices plus spreads sized to the residual lag. The team plans to operate its own pools that way (bin width roughly 0.25% to 2%, about 30 to 60 bins, spreads roughly 0.2% to 5%, prices pushed each block), which is sufficient while prices stay fresh. The residual risk is that the protocol is permissionless infrastructure: anyone can create a pool with a wide bin, a low spread, a slow or unfreshened price provider, or a large `MAX_TIME_DELTA`. In such a pool the round-trip fee can be well below the inter-update move and the leak is continuously extractable, with the loss borne by that pool's LPs.

Impact

LPs in any pool where the round-trip fee is smaller than the oracle-to-market lag lose value to arbitrage on every price-discovery event. For the team's own pools, fresh per-block pushes and sized spreads mitigate this. For permissionless pools created with low spreads, wide bins, or uncontrolled price freshness, the leak is real and cumulative. Per-trade invariants are unaffected; the loss is the residual loss the fee does not absorb because the price traded against is itself lagged.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/33d0a3431fb711cfb84052d8380a09be69beef01/contracts/MetricOmmPool.sol#L818>
<https://github.com/Oracle-Based-Pool/smart-contracts-poc/blob/edbc5f0e7efe9f3177e49eea52636d4930550c1b/contracts/AnchoredPriceProvider.sol#L246>

Tool Used

Manual Review

Recommendation

Acknowledge oracle price lag as an inherent property of the OMM design and defend it with freshness plus spread, while recognizing no static spread fully closes it under unbounded lag:

1. Keep prices fresh. Push updates as often as possible (per block) and configure very short staleness windows: Pyth Lazer and Chainlink with a small `MAX_TIME_DELTA`, and keep the compressed-oracle keeper tight. The smaller the served-price age, the smaller the exploitable gap.
2. Size the spread to the lag. The round-trip fee (`baseFeeX64` plus `addFee` plus notional) for a pool should exceed the worst realistic inter-update move for that pair, not merely the oracle's reported bid/ask.
3. Guard permissionless pools. Because pool creation is open, consider enforcing a minimum spread floor and a maximum `MAX_TIME_DELTA` relative to expected volatility at creation, and documenting to pool creators that a spread below the pair's inter-update move leaves their LPs exposed. The team's own configuration is a good reference; third-party pools that deviate from it carry the risk.

Discussion

christos-metric

thank you for the comments here!

As you say on your document as well The protocol's own operating strategy (push fresh prices every block, tight staleness, spreads sized to each pair) correctly mitigates it for the pools the team runs.

Given the permissionless nature of our protocol, we want to let people create whatever they want, but we will only recommend LPing on pools we deem safe (at least in our UI). So this is similar to what you said on 3!

With that said, first one is already something our code runs right through time staleness.

The second one we could do but "realistic" inter-update is tough to know in advance.

Let us know if there are still concerns with this, we consider it mitigated on our side based on my comments above

Oxklapouchy

Acknowledged

Issue L-1: Redundant else case for computeAnalyticalTargetPosForBuyToken0 function

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/88>

The else block at lines 271-273 executes when `deltaPriceX64 == 0`, representing zero price difference between bin bounds. However, the function precondition (line 295) requires `lowerPriceX64 < upperPriceX64`, making a zero delta impossible. The linear fallback computation `deltaPos = qX128 >> 128` can never execute. Removing this dead code eliminates maintenance confusion and clarifies that the quadratic formula is the sole computation path. The comment "this case is not possible" correctly identifies the issue but the code should be removed entirely rather than retained as unreachable logic.

Discussion

0xklapouchy

Fix confirmed in [PR#54](#)

Issue L-2: Constructor does not emit initialization events for fee caps and defaults

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/89>

Summary

The constructor sets `max*Fee*` caps and default protocol fees but does not emit `FeeCapsUpdated`, `SpreadProtocolFeeDefaultUpdated`, or `ProtocolNotionalFeeDefaultUpdated`, unlike the corresponding owner setters.

Vulnerability Detail

The constructor initializes fee caps to the hard limits and sets default protocol spread and notional fees to zero, but unlike `setFeeCaps`, `setDefaultSpreadProtocolFeeE6`, and `setDefaultProtocolNotionalFeeE8`, it emits none of the corresponding events. Indexers and monitoring tooling that rely on event streams to track factory configuration will not observe the initial state unless they read storage directly at deploy block.

Impact

Off-chain indexers and integrators miss the initial configuration at deploy time.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/6aa6c3b489b84b8b1c50dc6a4967184df17aa395/contracts/MetricOmmPoolFactory.sol#L101-L108>

Tool Used

Manual Review

Recommendation

Emit the same events on construction as when those values are updated later.

Discussion

Oxklapouchy

Fix confirmed in [PR#54](#)

Issue L-3: Lowering fee caps below current defaults can permanently block pool creation

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/90>

Summary

`setFeeCaps` does not verify that new caps remain at or above current default protocol fees (`spreadProtocolFeeE6`, `protocolNotionalFeeE8`). If caps are lowered below defaults, `_validatePoolParameters` reverts on every `createPool`.

Vulnerability Detail

`setFeeCaps` updates the four max-fee storage slots after checking only the hard protocol limits. It never compares the new protocol caps against the factory's current default protocol fees (`spreadProtocolFeeE6`, `protocolNotionalFeeE8`). If the owner lowers a cap below the stored default, every subsequent `createPool` hits `_validatePoolParameters`, which reverts with `ProtocolFeeTooHigh` because the default protocol fee now exceeds the cap. Recovery requires raising caps again or lowering defaults, an easy misconfiguration with no guardrail at write time.

Impact

Owner foot-gun: all new pool creation DoS until defaults or caps are corrected.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/6aa6c3b489b84b8b1c50dc6a4967184df17aa395/contracts/MetricOmmPoolFactory.sol#L274-L276>

<https://github.com/Metric-OMM/metric-core/blob/6aa6c3b489b84b8b1c50dc6a4967184df17aa395/contracts/MetricOmmPoolFactory.sol#L525-L526>

Tool Used

Manual Review

Recommendation

Validate cap updates against current defaults and existing pool configs, or auto-adjust defaults when caps are lowered.

Discussion

0xklapouchy

Fix confirmed in [PR#54](#)

Issue L-4: `_validateDeployFeeRates` is redundant and uses a weaker sum check

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/91>

Summary

`_validateDeployFeeRates` checks only that combined protocol+admin fees fit the sum of caps. Per-component limits are already enforced in `_validatePoolParameters`, making this check redundant; the sum check is weaker than the individual cap checks.

Vulnerability Detail

At the end of `createPool`, `_validateDeployFeeRates` compares the combined protocol-plus-admin spread and notional rates against the sum of the respective caps. The same deployment path already ran `_validatePoolParameters`, which enforces each protocol and admin component against its own cap. The deploy-time sum check is therefore redundant, and strictly weaker: it would accept some combinations that per-component validation already rejected, while adding no meaningful extra assurance if the earlier checks remain in place.

Impact

No direct exploit; misleading defense-in-depth and possible confusion if caps or defaults change independently.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/6aa6c3b489b84b8b1c50dc6a4967184df17aa395/contracts/MetricOmmPoolFactory.sol#L500-L508>

Tool Used

Manual Review

Recommendation

Remove redundant validation.

Discussion

0xklapouchy

Fix confirmed in [PR#54](#).

Issue L-5: setPoolProtocolFee silently clamps admin fees without an event

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/92>

Summary

When the owner updates a pool's protocol fees via `setPoolProtocolFee`, admin fees above the new admin caps are clamped downward with no dedicated event reflecting the clamp.

Vulnerability Detail

When the factory owner calls `setPoolProtocolFee`, the function loads the pool's stored admin fee components and, if either exceeds the current admin caps, silently reduces them before updating `poolFeeConfig` and calling `setPoolFees` on the pool. The protocol-fee update events fire, but nothing signals that the admin portion was clamped. Off-chain systems keyed only to admin-set events or stored config may report stale admin fee rates after a cap tightening.

Impact

Integrators and admins may believe admin fee rates unchanged when they were reduced without notice.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/6aa6c3b489b84b8b1c50dc6a4967184df17aa395/contracts/MetricOmmPoolFactory.sol#L305-L306>

Tool Used

Manual Review

Recommendation

Emit explicit events when admin fee components are clamped.

Discussion

0xklapouchy

Issue L-6: collectPoolFees is permissionless but not documented on the owner interface

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/93>

Summary

collectPoolFees has no access control, anyone can trigger fee collection for a pool. Behavior is acceptable but it lives on the factory owner interface.

Vulnerability Detail

collectPoolFees sits on the factory owner interface but carries no onlyOwner modifier, any address may call it and trigger collectFees on the pool with the stored fee split. That may be intentional (keepers, admins, or bots flushing fees), but the interface gives no hint that access is open. Operators reading IMetricOmmPoolFactoryOwner may assume owner-only behavior and miss that third parties can drive fee collection at any time.

Impact

No fund loss; operator or integrator confusion about who may call the function.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/6aa6c3b489b84b8b1c50dc6a4967184df17aa395/contracts/MetricOmmPoolFactory.sol#L339-L349>

Tool Used

Manual Review

Recommendation

Document that fee collection is intentionally permissionless and clarify intended callers in the owner interface.

Discussion

0xklapouchy

Fix confirmed in [PR#54](#)

Issue L-7: Protocol unpause cannot restore full operation if pool admin is unavailable [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/94>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`protocolUnpausePool` only transitions pause level from 2 → 1. Full unpause to level 0 requires the pool admin. If the admin key is lost, the owner cannot fully restore the pool.

Vulnerability Detail

The pause model lets the protocol owner force level 2 from level 0 or 1, and `protocolUnpausePool` can only step back to level 1. Returning to level 0 (full operation) requires the pool admin to call `unpausePool`. If the admin key is lost, compromised, or unresponsive after a protocol-level pause, the owner has no on-chain path to fully restore trading, even though they were able to impose the highest pause level in the first place.

Impact

Pool remains partially paused indefinitely unless admin cooperates.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/6aa6c3b489b84b8b1c50dc6a4967184df17aa395/contracts/MetricOmmPoolFactory.sol#L359-L362>

Tool Used

Manual Review

Recommendation

Define an owner recovery path to full unpause after a timelock or document this as an accepted admin-dependency in the pause design.

Discussion

0xklapouchy

Acknowledged

Issue L-8: WrongBinArrays revert is unreachable under current validation

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/95>

Summary

After bin unpacking and length checks, `posBinCount == 0` leading to `WrongBinArrays()` cannot occur, the earlier `BinLengthZero` check on the first packed slot already guarantees at least one positive bin.

Vulnerability Detail

In `_unpackAndValidateBinStates`, the positive-bin loop requires the first lane of each packed word to have non-zero length; an empty first lane reverts with `BinLengthZero`. That guarantees at least one positive bin is counted before the loop finishes. The later `if (posBinCount == 0) revert WrongBinArrays()` branch can never execute under the current control flow, it is dead defensive code left over from an earlier or alternate validation shape.

Impact

Dead code path only; no runtime effect.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/6aa6c3b489b84b8b1c50dc6a4967184df17aa395/contracts/MetricOmmPoolFactory.sol#L565>

Tool Used

Manual Review

Recommendation

Remove unreachable checks.

Discussion

0xklapouchy

Fix confirmed in [PR#54](#)

Issue L-9: Pool admin and fee destination are not bound to the CREATE2 deploy address

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/97>

Summary

`createPool` is permissionless. The pool address is fixed by CREATE2 from `DeployParams`, but `admin` and `adminFeeDestination` are stored in factory mappings after deploy and are excluded from the salt. An attacker can front-run with identical deploy parameters and different admin values.

Vulnerability Detail

The deployer receives a `DeployParams` struct that fully determines the CREATE2 address (tokens, fees, bins, hooks, salt) but not who will administer the pool. After deployment, the factory writes `poolAdmin[pool]` and `poolAdminFeeDestination[pool]` from `calldata`, so the deterministic address and the admin role are decoupled.

Because `createPool` is permissionless, anyone who sees or can predict a victim's parameters (public mempool, fixed config) can submit the same deploy payload with a different admin and win ordering. The pool lands at the address integrators expect, but with the attacker as admin; the victim's transaction then reverts on CREATE2 collision or must redeploy under a different salt.

Impact

Deploy griefing and, if address is trusted without checking admin, attacker-controlled pool administration and fee routing.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/6aa6c3b489b84b8b1c50dc6a4967184df17aa395/contracts/MetricOmmPoolFactory.sol#L164-L198>

Tool Used

Manual Review

Recommendation

Tie pool administration to deployment so address and admin cannot diverge: via CREATE2 args, caller binding, or creator-specific salt derivation.

Discussion

0xklapouchy

Fix confirmed in [PR#54](#)

Issue L-10: Swap router does not validate the target pool against the factory

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/98>

Summary

`MetricOmmPoolSwapper` accepts an arbitrary `pool` address on every entry point and never verifies it via the factory (`isPool`). It trusts the pool's returned deltas and the deltas passed into its callback, with no `balanceOf` accounting of its own. A malicious pool enables approval phishing and can sweep any incidental WETH/ETH balance the router holds.

Vulnerability Detail

Every swap entry forwards a caller-supplied `pool` to `_swapWithContext`, which calls `pool.swap(...)` and reads `pool.getImmutables()` for the token addresses inside `metricOmmSwapCallback`. None of this is checked against the factory's `isPool`. Because the router settles from a transient context whose payer is always `msg.sender`, a third party's approval cannot be spent directly, but a user tricked into calling the router with an attacker-controlled `pool` will have their approved tokens pulled to that pool (`_payInput -> safeTransferFrom(payer, pool, amount)`), bounded only by their approval.

Separately, on the native-output path the router calls `_unwrapAndSendNative(recipient, amountOut)` using the pool-reported `amountOut`, so a fake pool that reports an inflated output drains any WETH the router happens to hold; the native-input path can likewise sweep stray ETH via `_payInput's address(this).balance` branch. Funds in transit during a legitimate swap settle atomically and are not at risk; the exposure is incidental balance the router is not supposed to hold between transactions.

Impact

Elevated phishing surface (a malicious-pool interaction looks like a normal router call to wallets) and theft of any incidental WETH/ETH balance held by the router.

Code Snippet

<https://github.com/Metric-OMM/metric-periphery/blob/90039f9b68f6b2253425acfb6497fcf38e28cc17/contracts/MetricOmmPoolSwapper.sol#L648-L660>

<https://github.com/Metric-OMM/metric-periphery/blob/90039f9b68f6b2253425acfb6497fcf38e28cc17/contracts/MetricOmmPoolSwapper.sol#L699-L710>

<https://github.com/Metric-OMM/metric-periphery/blob/90039f9b68f6b2253425acfb6497fcf38e28cc17/contracts/MetricOmmPoolSwapper.sol#L725-L729>

Tool Used

Manual Review

Recommendation

Validate `pool` against the factory (e.g. `require(factory.isPool(pool))`) at each entry or in `_startSwap`, so only canonical pools can be targeted. This also removes the precondition for any fake-pool delta manipulation.

Discussion

Oxklapouchy

Fix confirmed in [PR#34](#)

`MetricOmmPoolSwapper` was removed in favor of new `MetricOmmSimpleRouter`.

Issue L-11: Raw swap() function provide no amount-based slippage protection

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/99>

Summary

The public `swap()` function expose only `priceLimitX64`, a hard price cap, with no `minAmountOut` / `maxAmountIn`. A transaction left in the mempool can be sandwiched if the price moves within the limit, because a price cap is not an output-amount guarantee.

Vulnerability Detail

Unlike `swapExactInput` / `swapExactOutput`, which enforce `minAmountOut` / `maxAmountIn`, the raw `swap()` family only forwards `priceLimitX64` to the pool. The price limit bounds the marginal fill price but not the realized output amount, so a swap can still execute at a worse aggregate amount than the user expected as long as the marginal price stays within the limit. With no amount-based guard, a pending raw-swap transaction is exposed to sandwiching and adverse price movement.

Impact

Users calling the raw `swap()` path can receive materially less output (or pay more) than intended, including via MEV sandwiching, with no amount floor to protect them.

Code Snippet

<https://github.com/Metric-OMM/metric-periphery/blob/90039f9b68f6b2253425acfb6497fcf38e28cc17/contracts/MetricOmmPoolSwapper.sol#L69-L93>

<https://github.com/Metric-OMM/metric-periphery/blob/90039f9b68f6b2253425acfb6497fcf38e28cc17/contracts/MetricOmmPoolSwapper.sol#L357>

Tool Used

Manual Review

Recommendation

Document clearly that the raw `swap()` overloads carry no amount-based slippage protection and are intended for advanced callers, and steer normal users to `swapExactInput` / `swapExactOutput`. Optionally add explicit `minAmountOut` / `maxAmountIn` parameters to the raw path.

Discussion

Oxklapouchy

Fix confirmed in [PR#34](#)

`MetricOmmPoolSwapper` was removed in favor of new `MetricOmmSimpleRouter`.

Issue L-12: Liquidity-add router does not validate the target pool against the factory

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/100>

Summary

`MetricOmmPoolLiquidityAdder` accepts an arbitrary `pool` address on every entry point and never verifies it via the factory (`isPool`). A user tricked into calling the adder with an attacker-controlled `pool` will have their approved tokens pulled to that pool in the callback, bounded only by their `max0`/`max1` caps.

Vulnerability Detail

Every add path forwards a caller-supplied `pool` to `_addLiquidity`, which calls `pool.addLiquidity(...)`; the pool then calls back `metricOmmModifyLiquidityCallback`, which reads `token0`/`token1` from `pool.getImmutables()` and does `safeTransferFrom(payer, pool, amount)`. None of this is checked against the factory's `isPool`. A malicious `pool` fully controls the callback's delta arguments and the `getImmutables()` token addresses, so it can pull up to `max0` of one attacker-named token and `max1` of another from the payer.

The blast radius is bounded: `payer` is always `msg.sender` (no third-party approval can be spent), the transfer recipient is forced to `msg.sender` (the pool itself), and the per-token amount is capped by the payer-chosen `max0`/`max1`. So this is a phishing surface, not an unconditional theft. Because the adder is a canonical router users broadly approve, a malicious-pool interaction can appear as a trusted, previously-used contract to wallets and raise no warning.

Impact

Phishing: a user induced to call the adder against a malicious pool loses up to `max0`/`max1` of the tokens they approved. No loss for users interacting with legitimate pools.

Code Snippet

<https://github.com/Metric-OMM/metric-periphery/blob/90039f9b68f6b2253425acfb6497fcf38e28cc17/contracts/MetricOmmPoolLiquidityAdder.sol#L44>

<https://github.com/Metric-OMM/metric-periphery/blob/90039f9b68f6b2253425acfb6497fcf38e28cc17/contracts/MetricOmmPoolLiquidityAdder.sol#L143-L151>

Tool Used

Manual Review

Recommendation

Validate `pool` against the factory (e.g. `require(factory.isPool(pool))`) at each entry or in `_addLiquidity`, so only canonical pools can be targeted.

Discussion

Oxklapouchy

Acknowledged only via documentation.

`MetricOmmPoolLiquidityAdder`'s constructor and every add function are unchanged, no `FACTORY` reference or `isPool()` check was added anywhere in the contract. The only change is a `@dev` comment stating the caller is responsible for supplying a legitimate pool address and that a malicious pool can pull tokens up to the caller-provided max caps, but the recommended fix (validate the pool against the factory) was not implemented.

Issue L-13: InvalidHooksConfig error is declared but unused in the factory

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/102>

Summary

`IMetricOmmPoolFactory.InvalidHooksConfig` remains in the interface, but hook validation was moved to `MetricHooks` and the factory no longer reverts with this error.

Vulnerability Detail

`InvalidHooksConfig` is still declared on `IMetricOmmPoolFactory`, but hook configuration is validated inside `MetricHooks.validateHooksConfig` during `createPool`, which reverts with the library's own errors. The factory-level error is never thrown.

Impact

Dead code.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/6aa6c3b489b84b8b1c50dc6a4967184df17aa395/contracts/interfaces/IMetricOmmPoolFactory/IMetricOmmPoolFactory.sol#L52>

Tool Used

Manual Review

Recommendation

Remove dead errors from the public interface.

Discussion

[0xklapouchy](#)

Fix confirmed in [PR#54](#)

Issue L-14: Inconsistent swap dynamics in a single bin [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/105>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The bin math tries to keep three things true at the same time:

1. The price is linear across the bin.
2. token0 is spread equally over the positions.
3. token1 is spread equally over the positions.

All three cannot hold together unless the price never changes, and the price *does* change inside a bin, so the assumptions fight each other.

The code itself is not broken. Every function does what it was written to do, and there is no single faulty line. The problem is the design. It does not keep the pool value constant. If you run a series of partial swaps in one bin with zero fee and no net token movement, the bin can end with less value than it started with, or with more value than it should, depending on which way you swap.

Because the bin only stores totals and re-spreads them equally on every swap, the liquidity a trader puts in does not stay in the positions it actually paid for. It gets smeared across the whole bin, and the small error builds up as the position moves back and forth.

Vulnerability Details

The price inside a bin is a straight line between the lower and upper price:

```
function calculatePriceAtBinPosition(
    uint256 lowerPriceX64,
    uint256 upperPriceX64,
    uint256 position,
    Math.Rounding rounding
) internal pure returns (uint256 priceX64) {
    uint256 maxSubCurrPos = MAX_POS_BIN - position;
    unchecked {
        if (rounding == Math.Rounding.Floor) {
            priceX64 = (lowerPriceX64 * maxSubCurrPos + upperPriceX64 * position) /
                ↪ MAX_POS_BIN;
        } else {
            priceX64 = Math.ceilDiv(lowerPriceX64 * maxSubCurrPos + upperPriceX64 *
                ↪ position, MAX_POS_BIN);
        }
    }
}
```

```

    }
}

```

The position is moved by treating each token as if it were spread evenly over its positions. For token1, going down, the move is a plain ratio of the balance:

```

function calculateBinPositionAfterSellingAmount1(
    uint256 currBinPos,
    uint256 tradedAmount1,
    uint256 availableToken1,
    Math.Rounding rounding
) internal pure returns (uint256 finalBinPos) {
    unchecked {
        if (rounding == Math.Rounding.Floor) {
            return (currBinPos * (availableToken1 - tradedAmount1)) / availableToken1;
        } else {
            return Math.ceilDiv(currBinPos * (availableToken1 - tradedAmount1),
                ↪ availableToken1);
        }
    }
}

```

token0 uses the mirror of this on the other side. So both tokens are treated as evenly spread over their positions, while the price is a sloped line. Even token0 per position and even token1 per position can only both be true when the price is flat. Once the price slopes from lower to upper, the two even-spread assumptions contradict each other, and the code just rebuilds the per-position spread from the stored total as if it were even, on every single swap.

This is also why liquidity ends up in the wrong place. The bin keeps one number for token0 and one for token1, nothing per position. Say the positions run 0 to 100, you are at position 20, and a trader buys token0 so the new position becomes 50. The token1 the trader paid belongs to the positions that were crossed, so it should sit in positions 20 to 50 at those positions' prices. Instead the bin only adds it to the token1 total, and the next swap reads that total back as if it were even over 0 to 50. The fresh token1 is now spread over positions that were never traded, the price it was bought at is gone, and the following swap prices it at the wrong average.

Because the position moves up and down over many partial swaps, the per-swap error stacks up, the drift can get large for some setups, and a round trip ends away from where value conservation says it should be.

Impact

A run of partial swaps that returns the bin to no net token0 (token0 starts and ends at zero) does not return the starting token1. In the out direction the pool loses token1. In the in direction the trader loses, by being overcharged. Either way value is not conserved.

A swap fee can hide small leaks but it does not fix the cause. This is a value-conservation break at the core swap level. Depending on the bin and the swap pattern it lets value be taken from liquidity providers, or traders to be overcharged, without paying for it. We consider it Medium.

PoC

The test drives the four in-bin swap primitives directly with no fee and no price limit, and checks the one thing a value-conserving AMM must satisfy. Pull token0 into the bin, oscillate, then sell all of it back out, and the bin should end with at least the token1 it started with.

How the test is set up: token0 is ETH and token1 is USDC. One bin with price range 2000 to 2001, fee 0, no price limit. It starts at the top of the bin holding 100,000 USDC and 0 ETH. Each run does one down swap to pull ETH into the bin, then 4 to 20 middle swaps whose side (buy token0 or buy token1) comes from the fuzzed selector bits and whose sizes come from the fuzzed seed (each swap is 1 to 99 percent of the relevant balance so it stays partial), then one final swap that drains all the ETH back out. The bin starts and ends with 0 ETH, so with no fee the final USDC must be at least the starting USDC. The test checks `final - start` is not negative and fails with `BIN LOST VALUE` otherwise. `testFuzz_out_only_no_value_loss` uses the `SpecifiedOut` primitives everywhere, `testFuzz_in_only_no_value_loss` uses the `SpecifiedIn` ones.

The test file, `test/SwapMathIncorrectLiquidity.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.35;

import {Test, console2} from "forge-std/Test.sol";
import {SwapMath} from "../contracts/libraries/SwapMath.sol";
import {BinState} from "../contracts/types/PoolStorage.sol";

/// @notice Single-bin sandbox that demonstrates the value leak caused by the
/// "both token0 and token1 distributed equally across positions" design.
///
/// ETH = token0, USDC = token1. Bin price range 2000 -> 2001, fee = 0%, no price
  ↳ limit.
/// Start position = MAX (top of the bin => all USDC, zero ETH), initial
  ↳ liquidity = 100,000 USDC.
///
/// A run performs:
/// 1. one DOWN swap that pulls ETH into the bin (so the bin now holds some
  ↳ ETH),
/// 2. N MIDDLE swaps that oscillate the position up and down inside the bin,
/// 3. one FINAL full drain that sells ALL the ETH back out.
///
/// The bin therefore starts and ends with exactly 0 ETH (net ETH flow = 0). With
  ↳ a 0% fee and a
```

```

/// value-conserving AMM, the bin MUST end with at least the USDC it started
↪ with. The fuzz tests
/// assert that invariant; it does not hold, which proves the design leaks value.
contract SwapMathIncorrectLiquidityTest is Test {
    uint256 internal constant ONE_X64 = 0x10000000000000000;
    uint256 internal constant MAX_POS_BIN = type(uint104).max;
    uint256 internal constant WAD = 1e18;

    function _freshBin() internal pure returns (BinState memory) {
        return BinState({
            token0BalanceScaled: 0, // ETH
            token1BalanceScaled: uint104(100_000 * WAD), // USDC
            lengthE6: 0,
            addFeeBuyE6: 0,
            addFeeSellE6: 0
        });
    }

    /// @dev Execute one partial swap of the chosen type, returning the new position.
    /// tokenBit: 0 => buyToken1 (price DOWN, spends bin USDC, pays ETH in),
    /// 1 => buyToken0 (price UP, spends bin ETH, pays USDC in).
    /// methodBit: 0 => SpecifiedIn, 1 => SpecifiedOut.
    function _doMiddleSwap(
        BinState memory bin,
        uint256 pos,
        uint256 lower_,
        uint256 upper_,
        uint8 tokenBit,
        uint8 methodBit,
        uint256 pct
    ) internal pure returns (uint256 newPos) {
        if (tokenBit == 0) {
            // ---- buyToken1*: price DOWN, needs bin USDC. priceLimit = lower_ (no
            ↪ clamp).
            if (methodBit == 0) {
                // SpecifiedIn: ETH in sized to buy ~pct% of USDC.
                uint256 ethIn = uint256(bin.token1BalanceScaled) * pct / (100 * 2001);
                SwapMath.SwapState memory st =
                    SwapMath.SwapState({amountSpecifiedRemainingScaled: ethIn,
                        ↪ amountCalculatedScaled: 0, protocolFeeAmountScaled: 0});
                (uint256 fp,,,) = SwapMath.buyToken1InBinSpecifiedIn(bin, pos, st, 0,
                    ↪ lower_, upper_, 0, 0);
                require(st.amountSpecifiedRemainingScaled == 0, "mid buyToken1In not
                    ↪ consumed");
                newPos = fp;
            } else {
                // SpecifiedOut: exact USDC out = pct% of USDC.
                uint256 usdcOut = uint256(bin.token1BalanceScaled) * pct / 100;
                SwapMath.SwapState memory st =

```

```

        SwapMath.SwapState({amountSpecifiedRemainingScaled: usdcOut,
        ↪ amountCalculatedScaled: 0, protocolFeeAmountScaled: 0});
    (uint256 fp,,,) = SwapMath.buyToken1InBinSpecifiedOut(bin, pos, st, 0,
    ↪ lower_, upper_, 0, 0);
    require(st.amountSpecifiedRemainingScaled == 0, "mid buyToken1Out not
    ↪ output");
    newPos = fp;
}
} else {
    // ---- buyToken0*: price UP, needs bin ETH. priceLimit = upper_ (no clamp).
    if (methodBit == 0) {
        // SpecifiedIn: USDC in sized to buy ~pct% of ETH.
        uint256 usdcIn = uint256(bin.token0BalanceScaled) * 2000 * pct / 100;
        SwapMath.SwapState memory st =
            SwapMath.SwapState({amountSpecifiedRemainingScaled: usdcIn,
            ↪ amountCalculatedScaled: 0, protocolFeeAmountScaled: 0});
        (uint256 fp,,,,) = SwapMath.buyToken0InBinSpecifiedIn(bin, pos, st, 0,
        ↪ lower_, upper_, type(uint256).max, 0);
        require(st.amountSpecifiedRemainingScaled == 0, "mid buyToken0In not
        ↪ consumed");
        newPos = fp;
    } else {
        // SpecifiedOut: exact ETH out = pct% of ETH.
        uint256 ethOut = uint256(bin.token0BalanceScaled) * pct / 100;
        SwapMath.SwapState memory st =
            SwapMath.SwapState({amountSpecifiedRemainingScaled: ethOut,
            ↪ amountCalculatedScaled: 0, protocolFeeAmountScaled: 0});
        (uint256 fp,,,) = SwapMath.buyToken0InBinSpecifiedOut(bin, pos, st, 0,
        ↪ lower_, upper_, type(uint256).max, 0);
        require(st.amountSpecifiedRemainingScaled == 0, "mid buyToken0Out not
        ↪ output");
        newPos = fp;
    }
}
}
}

/// @notice Generalized sequence with an arbitrary number of MIDDLE swaps.
/// FIRST swap (gets ETH into the bin) and the final drain use `method`; each
    ↪ middle's token side
/// is selector bit (i mod 8), each swap's size is derived from `pctSeed`.
    ↪ Returns signed USDC delta.
function runSeqN(uint8 selector, uint256 pctSeed, uint8 method, uint256
    ↪ numMiddles)
    external
    pure
    returns (int256 deltaUSDC)
{
    BinState memory bin = _freshBin();
    uint256 pos = MAX_POS_BIN;
    uint256 lower_ = 2000 * ONE_X64;

```

```

uint256 upper_ = 2001 * ONE_X64;
uint256 startUSDC = uint256(bin.token1BalanceScaled);

// FIRST: down-swap to pull ETH into the bin (token side 0), method-dependent.
pos = _doMiddleSwap(bin, pos, lower_, upper_, 0, method, _pctFrom(pctSeed, 0));
_assertPartial(pos);

// N MIDDLE swaps: token side = selector bit (i mod 8), size derived from the
↳ seed.
for (uint256 i = 0; i < numMiddles; i++) {
    uint8 tokenBit = uint8((selector >> (i & 7)) & 1);
    pos = _doMiddleSwap(bin, pos, lower_, upper_, tokenBit, method,
↳ _pctFrom(pctSeed, i + 1));
    _assertPartial(pos);
}

// LAST: full-drain ALL remaining ETH, drain method matches `method`.
if (method == 0) {
    uint256 usdcIn = uint256(bin.token0BalanceScaled) * 2100;
    SwapMath.SwapState memory st =
        SwapMath.SwapState({amountSpecifiedRemainingScaled: usdcIn,
↳ amountCalculatedScaled: 0, protocolFeeAmountScaled: 0});
    (uint256 fp,,,) = SwapMath.buyToken0InBinSpecifiedIn(bin, pos, st, 0,
↳ lower_, upper_, type(uint256).max, 0);
    pos = fp;
} else {
    uint256 ethDrain = uint256(bin.token0BalanceScaled);
    SwapMath.SwapState memory st =
        SwapMath.SwapState({amountSpecifiedRemainingScaled: ethDrain,
↳ amountCalculatedScaled: 0, protocolFeeAmountScaled: 0});
    (uint256 fp,,,) = SwapMath.buyToken0InBinSpecifiedOut(bin, pos, st, 0,
↳ lower_, upper_, type(uint256).max, 0);
    pos = fp;
    require(st.amountSpecifiedRemainingScaled == 0, "drain incomplete");
}
require(uint256(bin.token0BalanceScaled) == 0, "ETH not drained");

deltaUSDC = int256(uint256(bin.token1BalanceScaled)) - int256(startUSDC);
}

/// @notice OUT-ONLY fuzz with many middle swaps. Invariant: bin must NOT lose
↳ USDC value.
function testFuzz_out_only_no_value_loss(uint8 selector, uint256 pctSeed, uint8
↳ numMiddlesRaw) public view {
    uint256 numMiddles = 4 + (uint256(numMiddlesRaw) % 17); // 4..20 middle swaps
    try this.runSeqN(selector, pctSeed, 1, numMiddles) returns (int256 delta) {
        require(delta >= 0, "BIN LOST VALUE (out-only)");
    } catch {
        // infeasible sequence (dust / non-partial swap) -> skip, not a value-loss
        ↳ failure
    }
}

```


even though the bin started and ended with zero ETH and there was no fee. That is the leak, the pool losing value to the trader. The loss amounts are not just a few wei; in other scenarios they are significant enough to make the issue High severity for both the trader and the provider side.

Recommendation

Fix is not trivial for this particular issue because different prices will be fetched from price providers and liquidity distributions will be always affected from it.

Discussion

konrad-metric

This is a problem we are aware and we discuss it here: <https://github.com/Metric-OMM/metric-core/blob/dev-hooks/docs/SwapMathAnalysis.md>.

I am curious about your recommended solution. I do not fully understand what happens when pL, pH, and pC change on Price Provider price change. For me it seems that the amount of the token1 will increase/decrease inside a bin - if it is so it cant be like this as it could break the pool completely. Could you please verify if it is true or if I am missing something?

konrad-metric

Let me share some thoughts.

If we fix: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/106>.

There will be no possibility to drain pool if: A1. we start in drain impossible state - lets call it equilibrium state A2. there is no price provider price updates.

However: B1. if we are in equilibrium state and price is updated we end in non-equilibrium state. B2. it wont be possible to drain all funds but some amount until the bin is in equilibrium. B3. The drain is bigger the more price has changed.

there is no way to resolve the latest if we want to keep the distance of current price to PP price in % terms constant at PP update (a.k.a. position in bin)

The only alternative is to recalculate the bin position at each PP update. together with the issue/106 there will be no drain at all but there may be arbitrage opportunity that will lead to lose of value anyway.

DemoreXTess

Hey @konrad-metric, I run some fuzz tests in order to check issue impacts separately (#106 and #105), #106's fix for output specified functions were easy and I tested only out functions in order to see #105's exact impact.

My tests confirmed that #105 is actually doesn't cause value leak, instead it overcharge trader. All the tests are ended with positive value in the pool.

I also understood the reason behind it while I compare it with linearly increasing liquidity design for token1. Simply when buying token0, some percentage of the token1 trader provided goes to lower price points after rebalancing the liquidity. Therefore trader has to provide more token0 to buy that token1 back from low price points. So, actually we can't even say a overcharge for it. Just next swaps affected from rebalancing and pool earned some extra value from it. This confirms that issue #106 was hiding the actual impact of this issue.

I am planning to reduce the severity of this issue to Low and re-write impact title.

Btw, this is the code I used for "specified out" function if you want to check out:

```
uint256 p1X64 = calculatePriceAtBinPosition(lowerPriceX64, upperPriceX64,
    ↪ finalBinPos, Math.Rounding.Floor);
uint256 p2X64 = calculatePriceAtBinPosition(lowerPriceX64, upperPriceX64,
    ↪ currBinPos, Math.Rounding.Floor);
uint256 amountInScaled;
if(p2X64 == p1X64) { // zero division prevented here
    amountInScaled = Math.ceilDiv(amountOutScaled << 64, p1X64);
} else {
    amountInScaled = Math.mulDiv(amountOutScaled,
    ↪ uint128(ABDKMath64x64.ln(int128(int256(Math.mulDiv(p2X64, 1 << 64,
    ↪ p1X64))))), (p2X64 - p1X64));
}
```

Please let me know your thoughts.

konrad-metric

@DemoreXTess, can you share fuzz tests that you have used?

DemoreXTess

Ofcourse, before this test I applied the SwapMath.sol change that I described above for only specified out buy token0 function. Let me describe it. test_enumerate_out_only checks that a single ETH/USDC SwapMath bin never loses value over a zero-fee, net-zero round trip. It starts a bin with 100,000 USDC, sells ETH in, runs 50 randomized up/down middle swaps, then drains all ETH back out. So final USDC should always be ≥ starting USDC. It sweeps 8 selector seeds over a percentage grid and logs the worst loss / best gain per selector. That's a good test to check biggest unfair gain or biggest loss scenarios.

You can directly paste it to /test directory and run with `forge test --match-contract SwapMathEnumerateOutOnlyAuditTest -vvv` command. You can also change for loop numbers in runSeq function to simulate other gain/loss cases. In this test, there are 50 middle swap by default.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.35;

import {Test, console2} from "forge-std/Test.sol";
```

```

import {SwapMath} from "../contracts/libraries/SwapMath.sol";
import {BinState} from "../contracts/types/PoolStorage.sol";

/// @notice Standalone harness for the OUT-ONLY selector enumeration.
///   ETH = token0, USDC = token1. The bin starts at the top (pos = MAX => all
///   ↳ USDC, zero ETH)
///   holding 100,000 USDC, with price range [2000, 2010). Fee is 0\% throughout.
///   Each sequence: pull ETH into the bin (down-swap), oscillate via random middle
///   ↳ swaps, then
///   fully drain the ETH back out. With 0\% fee and net-zero ETH flow the bin
///   ↳ should NOT lose USDC.
contract SwapMathEnumerateOutOnlyAuditTest is Test {
    uint256 internal constant ONE_X64 = 0x10000000000000000;
    uint256 internal constant MAX_POS_BIN = type(uint104).max;
    uint256 internal constant WAD = 1e18;

    // -----
    // The test under examination.
    // -----

    /// @notice Enumerate the 8 out-only token-side combos over a pct grid; report
    /// ↳ any losses.
    function test_enumerate_out_only() public view {
        uint256[3] memory grid = [uint256(25), 55, 90];
        uint256 losing = 0;
        uint256 globalMaxLoss = 0;
        uint256 globalMaxGain = 0;
        int256 worstDelta = 0;
        uint8 worstSel;
        uint8 bestSel;

        // 8 selectors x (3^4) percentage tuples. `runSeq` is called externally so an
        /// ↳ infeasible
        // sequence (a swap that reverts / hits a bin boundary) can be caught instead
        /// ↳ of aborting.
        for (uint256 s = 0; s < 8; s++) {
            uint256 maxLoss = 0;
            uint256 maxGain = 0;
            for (uint256 a = 0; a < 3; a++) {
                for (uint256 b = 0; b < 3; b++) {
                    for (uint256 c = 0; c < 3; c++) {
                        for (uint256 d = 0; d < 3; d++) {
                            try this.runSeq(uint8(s), grid[a], grid[b], grid[c], grid[d], 1)
                                ↳ returns (int256 delta) {
                                // delta = final USDC - start USDC. Negative => the bin LOST value.
                                if (delta < 0 && uint256(-delta) > maxLoss) maxLoss =
                                    ↳ uint256(-delta);
                                if (delta > 0 && uint256(delta) > maxGain) maxGain = uint256(delta);
                            } catch {

```

```

        require(false, "infeasible combo (non-partial / not consumed) ->
        ↪ skip");
    }
}
}
}
}
if (maxLoss > 0) {
    losing++;
    console2.log("OUT-ONLY selector", s, "max LOSS (wei)", maxLoss);
    if (maxLoss > globalMaxLoss) {
        globalMaxLoss = maxLoss;
        worstSel = uint8(s);
    }
} else {
    console2.log("OUT-ONLY selector", s, "no loss (min delta >= 0)");
}
if (maxGain > 0) {
    console2.log("OUT-ONLY selector", s, "max GAIN (wei)", maxGain);
    if (maxGain > globalMaxGain) {
        globalMaxGain = maxGain;
        bestSel = uint8(s);
    }
} else {
    console2.log("OUT-ONLY selector", s, "no gain (max delta <= 0)");
}
}
console2.log("=====");
console2.log("OUT-ONLY losing selectors (of 8) =", losing);
console2.log("OUT-ONLY worst selector          =", worstSel);
console2.log("OUT-ONLY worst max LOSS ( USDC )    = %18e", globalMaxLoss);
console2.log("OUT-ONLY best selector              =", bestSel);
console2.log("OUT-ONLY best max GAIN ( USDC )      = %18e", globalMaxGain);
worstDelta; // silence unused
console2.log("=====");
}

// -----
// Sequence runner (called externally by the enumeration so reverts are
// ↪ catchable).
// -----

// Every swap uses the chosen `method`. First swap = buyToken1 (pull USDC, pay
// ↪ ETH) to get ETH
// into the bin from the top; 50 middle swaps whose token side AND size are
// ↪ pseudo-random per swap
// (seeded by selector + pct5 + i, since a uint8 selector can't carry many
// ↪ independent sides);
// last swap = full drain via buyToken0. Returns signed USDC delta (final -
// ↪ start).

```

```

function runSeq(uint8 selector, uint256 p1, uint256 pa, uint256 pb, uint256 pc,
    ↪ uint8 method)
    external
    pure
    returns (int256 deltaUSDC)
{
    uint256[3] memory pcts = [pa, pb, pc];
    BinState memory bin = _freshBin();
    uint256 pos = MAX_POS_BIN;
    uint256 lower_ = 2000 * ONE_X64;
    uint256 upper_ = 2010 * ONE_X64;
    uint256 startUSDC = uint256(bin.token1BalanceScaled);

    // FIRST (fixed): buyToken1 -- pull p1\% of USDC out, pay ETH in. Price DOWN
    ↪ from the top.
    pos = _doMiddleSwap(bin, pos, lower_, upper_, 0, method, p1);
    _assertPartial(pos);

    // 3 MIDDLE swaps. A uint8 selector cannot carry many independent token sides,
    ↪ so the entropy
    // comes from a per-swap hash that mixes selector + pcts + i. Each swap gets
    ↪ its own random
    // token side AND its own random size (1..99\%) -- nothing is predetermined.
    for (uint256 i = 0; i < 50; i++) {
        uint8 tokenBit = uint8(uint256(keccak256(abi.encode("side", selector, pcts,
            ↪ i)))) & 1); // 0 = buyToken1 (down), 1 = buyToken0 (up)
        uint256 pct = _pctFrom(uint256(keccak256(abi.encode(selector, pcts))), i); //
            ↪ random 1..99\%
        pos = _doMiddleSwap(bin, pos, lower_, upper_, tokenBit, method, pct);
        _assertPartial(pos);
    }

    // LAST (fixed): full-drain ALL remaining ETH, drain method matches `method`.
    // method == 0 -> buyToken0InBinSpecifiedIn (supply USDC, exact input;
    ↪ leftover input expected)
    // method == 1 -> buyToken0InBinSpecifiedOut (exact ETH out)
    if (method == 0) {
        uint256 usdcIn = uint256(bin.token0BalanceScaled) * 2100; // enough to buy
            ↪ out all ETH
        SwapMath.SwapState memory st =
            SwapMath.SwapState({amountSpecifiedRemainingScaled: usdcIn,
                ↪ amountCalculatedScaled: 0, protocolFeeAmountScaled: 0});
        (uint256 fp,,,) = SwapMath.buyToken0InBinSpecifiedIn(bin, pos, st, 0,
            ↪ lower_, upper_, type(uint256).max, 0);
        pos = fp;
    } else {
        uint256 ethDrain = uint256(bin.token0BalanceScaled);
        SwapMath.SwapState memory st =
            SwapMath.SwapState({amountSpecifiedRemainingScaled: ethDrain,
                ↪ amountCalculatedScaled: 0, protocolFeeAmountScaled: 0});
    }
}

```

```

        (uint256 fp,,,) = SwapMath.buyToken0InBinSpecifiedOut(bin, pos, st, 0,
        ↪ lower_, upper_, type(uint256).max, 0);
        pos = fp;
        require(st.amountSpecifiedRemainingScaled == 0, "drain incomplete");
    }
    require(uint256(bin.token0BalanceScaled) == 0, "ETH not drained");

    deltaUSDC = int256(uint256(bin.token1BalanceScaled)) - int256(startUSDC);
}

// -----
// Helpers.
// -----

/// @dev Fresh bin at the top of the range: 100,000 USDC, zero ETH.
function _freshBin() internal pure returns (BinState memory) {
    return BinState({
        token0BalanceScaled: 0,
        token1BalanceScaled: uint104(100_000 * WAD),
        lengthE6: 0,
        addFeeBuyE6: 0,
        addFeeSellE6: 0
    });
}

/// @dev Execute one partial swap of the chosen type, returning the new position.
///      tokenBit: 0 => buyToken1 (price DOWN, spends bin USDC), 1 => buyToken0
///      ↪ (price UP, spends bin ETH).
///      methodBit: 0 => SpecifiedIn, 1 => SpecifiedOut.
///      Sizes are scaled from the bin's price bounds so each swap stays partial
///      ↪ (1..99\% of a balance).
function _doMiddleSwap(
    BinState memory bin,
    uint256 pos,
    uint256 lower_,
    uint256 upper_,
    uint8 tokenBit,
    uint8 methodBit,
    uint256 pct
) internal pure returns (uint256 newPos) {
    if (tokenBit == 0) {
        // ---- buyToken1*: price DOWN, needs bin USDC. priceLimit = lower_ (no
        ↪ clamp).
        if (methodBit == 0) {
            // SpecifiedIn: ETH in sized to buy ~pct\% of USDC (avg price < upper =>
            ↪ USDC out < pct\% => partial).
            uint256 ethIn = uint256(bin.token1BalanceScaled) * pct / (100 * upper_ /
            ↪ ONE_X64);
            SwapMath.SwapState memory st =

```

```

        SwapMath.SwapState({amountSpecifiedRemainingScaled: ethIn,
            ↪ amountCalculatedScaled: 0, protocolFeeAmountScaled: 0});
    (uint256 fp,,,) = SwapMath.buyToken1InBinSpecifiedIn(bin, pos, st, 0,
        ↪ lower_, upper_, 0, 0);
    require(st.amountSpecifiedRemainingScaled == 0, "mid buyToken1In not
        ↪ consumed");
    newPos = fp;
} else {
    // SpecifiedOut: exact USDC out = pct\% of USDC (< balance => partial).
    uint256 usdcOut = uint256(bin.token1BalanceScaled) * pct / 100;
    SwapMath.SwapState memory st =
        SwapMath.SwapState({amountSpecifiedRemainingScaled: usdcOut,
            ↪ amountCalculatedScaled: 0, protocolFeeAmountScaled: 0});
    (uint256 fp,,,) = SwapMath.buyToken1InBinSpecifiedOut(bin, pos, st, 0,
        ↪ lower_, upper_, 0, 0);
    require(st.amountSpecifiedRemainingScaled == 0, "mid buyToken1Out not
        ↪ output");
    newPos = fp;
}
} else {
    // ---- buyToken0*: price UP, needs bin ETH. priceLimit = upper_ (no clamp).
    if (methodBit == 0) {
        // SpecifiedIn: USDC in sized to buy ~pct\% of ETH (avg price > lower =>
            ↪ ETH out < pct\% => partial).
        uint256 usdcIn = uint256(bin.token0BalanceScaled) * lower_ / ONE_X64 * pct
            ↪ / 100;
        SwapMath.SwapState memory st =
            SwapMath.SwapState({amountSpecifiedRemainingScaled: usdcIn,
                ↪ amountCalculatedScaled: 0, protocolFeeAmountScaled: 0});
        (uint256 fp,,,) = SwapMath.buyToken0InBinSpecifiedIn(bin, pos, st, 0,
            ↪ lower_, upper_, type(uint256).max, 0);
        require(st.amountSpecifiedRemainingScaled == 0, "mid buyToken0In not
            ↪ consumed");
        newPos = fp;
    } else {
        // SpecifiedOut: exact ETH out = pct\% of ETH (< balance => partial).
        uint256 ethOut = uint256(bin.token0BalanceScaled) * pct / 100;
        SwapMath.SwapState memory st =
            SwapMath.SwapState({amountSpecifiedRemainingScaled: ethOut,
                ↪ amountCalculatedScaled: 0, protocolFeeAmountScaled: 0});
        (uint256 fp,,,) = SwapMath.buyToken0InBinSpecifiedOut(bin, pos, st, 0,
            ↪ lower_, upper_, type(uint256).max, 0);
        require(st.amountSpecifiedRemainingScaled == 0, "mid buyToken0Out not
            ↪ output");
        newPos = fp;
    }
}
}
}

```

```

/// @dev Per-swap percentage in [1,99] derived from a seed and index.

```

```

function _pctFrom(uint256 seed, uint256 i) internal pure returns (uint256) {
    return 1 + (uint256(keccak256(abi.encode(seed, i))) \% 99);
}

/// @dev Require the swap stayed strictly inside the bin (did not hit either
    ↪ boundary).
function _assertPartial(uint256 pos) internal pure {
    require(pos > 0 && pos < MAX_POS_BIN, "swap was not partial (hit a bin
        ↪ boundary)");
}
}

```

Issue L-15: LiquidityLib current-bin partial products use unchecked multiplication

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/107>

Summary

In `LiquidityLib.addLiquidity`, the current bin (empty state) branch computes two partial products, `token0Proportion * ctx.initialScaledToken0PerShareE18` and `token1Proportion * ctx.initialScaledToken1PerShareE18`, with a `raw *` before passing them into `Math.mulDiv`. Every other multiplication in the same function uses the `_checkedMul` helper to guard against overflow, but these two do not, and the whole branch runs inside an unchecked block. While `token{0,1}Proportion` is bounded by `uint104.max`, `initialScaled...PerShareE18` is a creator set value with no on chain upper bound, so a pathological per share configuration can make the raw product wrap modulo 2^{256} and produce a wrong `amountScaled`. This is a consistency and hardening gap rather than a demonstrated exploit.

Vulnerability Detail

When a bin is empty and is the current bin, initial liquidity is split between the two tokens by position, and the scaled amounts are computed as:

```
} else { // we are in current bin
  uint256 token0Proportion = type(uint104).max - ctx.curPosInBin;
  uint256 token1Proportion = ctx.curPosInBin;
  amount0Scaled =
    (Math.mulDiv(
      token0Proportion * ctx.initialScaledToken0PerShareE18, // raw multiply, not
      ↪ _checkedMul
      sharesToAdd,
      uint256(type(uint104).max) * 1e18,
      Math.Rounding.Ceil
    ));
  amount1Scaled =
    (Math.mulDiv(
      token1Proportion * ctx.initialScaledToken1PerShareE18, // raw multiply, not
      ↪ _checkedMul
      sharesToAdd,
      uint256(type(uint104).max) * 1e18,
      Math.Rounding.Ceil
    ));
}
```

Every sibling multiplication in this function is guarded. The other empty bin branches use `_checkedMul(ctx.initialScaledToken0PerShareE18, sharesToAdd)` and `_checkedMul(ctx.`

`initialScaledToken1PerShareE18, sharesToAdd)`, and the existing shares branch uses `_checkedMul(binState.token0BalanceScaled, sharesToAdd)`. Only the two current bin partial products above use a raw `*`, and they sit inside the function level unchecked block, so an overflow wraps silently instead of reverting.

`token0Proportion` and `token1Proportion` are each bounded by `type(uint104).max`, so they alone cannot overflow. The risk comes from the other operand. `ctx.initialScaledToken0PerShareE18` and `ctx.initialScaledToken1PerShareE18` are derived from creator supplied per share amounts multiplied by the pool scale multiplier (`initialAmountPerShareE18 * scaleMultiplier`), with no enforced on chain upper bound. If a pool is created with a pathological per share value, the product `token0Proportion * ctx.initialScaledToken0PerShareE18` can exceed 2^{256} . The raw multiply then wraps, `Math.mulDiv` receives a corrupted numerator, and `amountScaled` is computed incorrectly. Unlike the guarded paths, no revert protects against this case.

Because reaching the overflow requires a creator chosen extreme per share configuration, the likelihood is low and the practical impact is limited, but the inconsistency is a real hardening gap: the function clearly intends every multiplication that involves a creator set scaled value to be overflow checked, and these two are not.

Impact

Informational. The raw partial products are not overflow checked like the rest of the function, so a pathological creator set `initialScaled...PerShareE18` can wrap the multiply and yield a wrong `amountScaled` in the current bin initial liquidity branch.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/33d0a3431fb711cfb84052d8380a09be69beef01/contracts/libraries/LiquidityLib.sol#L90-L107>

Tool Used

Manual Review

Recommendation

Use `_checkedMul()` for the two current bin partial products (`token0Proportion * ctx.initialScaledToken0PerShareE18` and `token1Proportion * ctx.initialScaledToken1PerShareE18`), consistent with every other multiplication in `addLiquidity`, so an overflow reverts rather than wrapping silently.

Discussion

Oxklapouchy

Fix confirmed in [PR#60](#)

Fixed, but via a different mechanism than recommended. `initialScaledAmount0/1PerShareE18` had no upper bound in the prior commit, so the reported overflow was reachable at the time. The factory now rejects either value $\geq \text{type}(\text{uint128}).\text{max}$ at `createPool` time, and that bounded value is what flows into `ctx.initialScaledToken0/1PerShareE18` in `LiquidityLib`. Combined with `token0Proportion/token1Proportion` being bounded by $\text{type}(\text{uint104}).\text{max}$, the raw product is now bounded by $2^{104} * 2^{128} = 2^{232}$, safely inside `uint256`. The overflow precondition was removed at the source rather than guarded at the call site.

Issue L-16: PriceVelocityGuardExtension uses the arithmetic mid instead of the geometric mid

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/108>

Summary

`PriceVelocityGuardExtension.beforeSwap` derives its reference mid as the arithmetic mean $(\text{bidPriceX64} + \text{askPriceX64}) / 2$, while the pool computes the price it actually trades on as the geometric mean $\text{Math.sqrt}(\text{bidPriceX64} * \text{askPriceX64})$ (`MetricOmmPool._midAndSpreadFeeX64FromBidAsk`). The velocity guard therefore caps the inter-block change of a quantity that is not exactly the pool's traded mid. The offset between the two means cancels block-to-block at a stable spread and is second-order in the spread (sub-basis-point at realistic spreads), so this is a calibration/exactness note rather than a security bug; there is no attacker advantage.

Vulnerability Detail

The guard computes its mid arithmetically:

```
address pool_ = msg.sender;
uint128 midPrice = (bidPriceX64 + askPriceX64) / 2;
```

and bounds the squared per-block change of that value:

```
uint256 delta = midPrice > prevMid ? uint256(midPrice - prevMid) : uint256(prevMid
↵ - midPrice);
uint256 changeE18 = (delta * 1e18) / uint256(prevMid);
uint256 actualSq = changeE18 * changeE18;
uint256 allowedSq = uint256(maxChange) * uint256(maxChange) * (1 + blockDiff);
if (actualSq > allowedSq) {
    revert PriceVelocityExceeded(actualSq, allowedSq);
}
```

The pool, however, anchors its execution price on the geometric mid:

```
midPriceX64 = Math.sqrt(bidPriceX64 * askPriceX64);
```

By the AM-GM inequality the arithmetic mean is always greater than or equal to the geometric mean, and the gap grows with the bid/ask spread. The guard claims to bound "how fast the provided price can move," but it bounds the arithmetic mid rather than the geometric mid the pool actually prices on, so the threshold it enforces does not correspond exactly to the economically meaningful price move it is meant to constrain.

Impact

Informational. Calibration exactness only. The guard bounds a quantity (arithmetic mid) that differs second-order from the pool's traded quantity (geometric mid). There is no exploit, no fund loss.

Code Snippet

<https://github.com/Metric-OMM/metric-periphery/blob/cc8fef9781fba405cfd1945fe6bf49a00da132bb/contracts/extensions/PriceVelocityGuardExtension.sol#L47>

<https://github.com/Metric-OMM/metric-core/blob/33d0a3431fb711cfb84052d8380a09be69beef01/contracts/MetricOmmPool.sol#L817>

Tool Used

Manual Review

Recommendation

Compute the guard's reference mid as the geometric mid `Math.sqrt(bidPriceX64 * askPriceX64)`, matching `MetricOmmPool._midAndSpreadFeeX64FromBidAsk`, so the velocity bound is enforced on exactly the same quantity the pool trades on.

Discussion

0xklapouchy

Fix confirmed via [PR#60](#) and [PR#42](#)

Issue L-17: Codebook `getTable()` off-by-one drops index 255 [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/109>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `getTable()` view helper in `Codebook256` allocates an array sized `MAX_INDEX` (255) and loops `i < MAX_INDEX`, so it returns only 255 of the 256 codebook entries and never emits index 255 (value 10000). The helper is view-only and is not used by the pricing decode path, which indexes the table correctly, so pricing is unaffected.

Vulnerability Detail

`MAX_INDEX` is 255 (the maximum 8-bit index), but the table holds 256 entries (indices 0 to 255). The helper sizes the array to 255 and stops the loop at 254:

```
uint8 internal constant MAX_INDEX = type(uint8).max; // 255

function getTable() external pure returns (uint16[] memory t) {
    t = new uint16[] (MAX_INDEX);           // 255, should be 256
    for (uint256 i; i < MAX_INDEX; i++) { // 0..254, drops index 255
        t[i] = _valueAt(i);
    }
}
```

The array is one element too short and the loop never reads index 255, so the returned table omits the last entry. The live decode path is correct and does return index 255.

Impact

Informational. The view helper omits one entry (index 255).

Code Snippet

<https://github.com/Oracle-Based-Pool/smart-contracts-poc/blob/edbc5f0e/contracts/oracles/utils/Codebook256.sol#L14-L15>

Tool Used

Manual Review

Recommendation

Size the array to include index 255 and extend the loop bound.

Issue L-18: BaseMetricExtension onlyPool check is spoofable

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/110>

Summary

`BaseMetricExtension.onlyPool` authenticates a caller by calling `IMetricOmmPool(msg.sender).getImmutables().factory` and comparing it to the extension's `FACTORY`. That value is self-reported by the caller: any contract can implement `getImmutables()` to return a struct whose `factory` field equals `FACTORY`, so any contract can pass `onlyPool`. The correct check is `IMetricOmmPoolFactory(FACTORY).isPool(msg.sender)`, a value held by the trusted factory that a caller cannot forge.

Vulnerability Detail

The modifier is:

```
modifier onlyPool() {
    if (IMetricOmmPool(msg.sender).getImmutables().factory != FACTORY) {
        revert OnlyPool(msg.sender, FACTORY);
    }
    _;
}
```

`getImmutables()` is an arbitrary external call into `msg.sender`. A malicious contract can return any `PoolImmutables` it likes, including one whose `factory == FACTORY`, and thereby satisfy the modifier. The check authenticates a field the caller controls, not membership in the set of pools the factory actually deployed. The factory exposes `isPool` (backed by `poolToIdx`), which is the unforgeable membership oracle.

The defect is correctness gap: the authentication primitive is wrong, even though no current callback is reachable on a spoofed pool and writes persistent state keyed by something other than `msg.sender`.

Impact

There is no current exploit path. The risk is latent: any future extension that trusts `onlyPool` for value-bearing logic would inherit a forgeable authentication boundary.

Code Snippet

<https://github.com/Metric-OMM/metric-periphery/blob/cc8fef9781fba405cfd1945fe6bf49a00da132bb/contracts/extensions/base/BaseMetricExtension.sol#L21>

Tool Used

Manual Review

Recommendation

Replace the self-asserted check with factory membership:

```
modifier onlyPool() {  
    if (!IMetricOmmPoolFactory(FACTORY).isPool(msg.sender)) {  
        revert OnlyPool(msg.sender, FACTORY);  
    }  
    _;  
}
```

Discussion

0xklapouchy

Fix confirmed in [PR#42](#)

Issue L-19: Uncommon price limit input design

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/112>

Summary

To ask for an unconstrained swap (no price limit), the caller has to pass a different "magic" value depending on the direction: 0 for a `zeroForOne` swap (price going down) and `type(uint128).max` for a `!zeroForOne` swap (price going up). The swapper and the router validate the price limit and revert when the wrong-direction extreme is passed, but they never normalize it. Most integrators treat 0 as the universal "no limit" value, and doing that here makes every price-up swap revert with `InvalidPriceLimitForDirection`. There's no money at risk – it's an integration footgun.

Vulnerability Details

The "unconstrained" value is asymmetric. The router base spells it out in `_openLimit`:

```
function _openLimit(bool zeroForOne) internal pure returns (uint128) {
    return zeroForOne ? 0 : type(uint128).max;
}
```

The validation only ever rejects the opposite extreme – it does not convert anything:

```
// MetricOmmPoolSwapper.sol (an identical copy lives in MetricOmmSwapRouterBase.sol)
function _validatePriceLimit(bool zeroForOne, uint128 priceLimitX64) private pure {
    if (zeroForOne) {
        if (priceLimitX64 == type(uint128).max) revert
        ↪ InvalidPriceLimitForDirection(true, priceLimitX64);
        return;
    }
    if (priceLimitX64 == 0) revert InvalidPriceLimitForDirection(false,
        ↪ priceLimitX64);
}
```

The gap is that it only *rejects*, it never *converts*. A caller who follows the common "0 means no limit" convention gets:

- `zeroForOne` (down): 0 is accepted and runs unconstrained – works as they expect.
- `!zeroForOne` (up): 0 is rejected with `InvalidPriceLimitForDirection` – surprise revert.

Impact

This is an integration / usability issue only – no funds are at risk and no accounting is affected.

An integrator or front-end that uses one uniform "no price limit" value (whichever of 0 or `type(uint128).max` they pick) will have half of their swaps revert: the `!zeroForOne` ones if they standardize on 0, the `zeroForOne` ones if they standardize on `type(uint128).max`. Until they learn that the sentinel is direction-dependent, the result is failed transactions and wasted gas. Once they pass the correct per-direction value (0 for down, `max` for up), everything works.

Recommendation

Normalize the limit instead of rejecting it, so 0 works as "unconstrained" in both directions.

Discussion

Oxklapouchy

Fix confirmed in [PR#42](#)

Issue L-20: Data providers are using wrong spread fee

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/114>

Summary

`MetricOmmPoolDataProvider` builds its quoted bid/ask the wrong way. It adds the protocol/admin spread-fee **split** (`protocolSpreadFeeE6 + adminSpreadFeeE6`) on top of the marginal price as if it were a price spread, and it never includes the actual oracle base fee (the bid/ask spread the trader really pays). So the bid/ask and depth prices returned by the lens don't match what a swap actually costs. This is a read-only view, so no funds are involved, but integrators reading these quotes get wrong numbers.

Vulnerability Details

In the pool, the fee a trader pays in a bin is `baseFee + addFee`:

- `baseFee` is the oracle bid/ask spread, `ask/mid - 1` (see `_midAndSpreadFeeX64FromBidAsk` in the pool).
- `addFee` is the per-bin `addFeeBuyE6 / addFeeSellE6`.

`spreadFeeE6` (`protocolSpreadFeeE6 + adminSpreadFeeE6`) is something else entirely: it's the share of the collected fee that goes to protocol/admin versus LPs. It is not an extra charge on the trader.

The lens uses that split as the spread and drops the base fee:

```
// getBestBidAndAsk
uint256 buySpreadFeeE6 = uint256(protocolSpreadFeeE6) + uint256(adminSpreadFeeE6)
↪ + uint256(addFeeBuyE6);
uint256 sellSpreadFeeE6 = uint256(protocolSpreadFeeE6) + uint256(adminSpreadFeeE6)
↪ + uint256(addFeeSellE6);

uint256 askBeforeNotional = Math.mulDiv(marginalPriceX64, ONE_E6 + buySpreadFeeE6,
↪ ONE_E6, Math.Rounding.Ceil);
uint256 bidAfterSpread = Math.mulDiv(marginalPriceX64, ONE_E6, ONE_E6 +
↪ sellSpreadFeeE6, Math.Rounding.Floor);
```

The oracle bid/ask are right there (`bidFromOracleX64`, `askFromOracleX64`, and `midPriceX64 = sqrt(bid*ask)`), so the base fee could be derived, but it isn't.

The same mistake feeds the depth ladder. `_loadDepthEnv` sets the (misleadingly named) base spread to the same split:

```
env.baseSpreadE6 = uint256(protocolSpreadFeeE6) + uint256(adminSpreadFeeE6);
```

and `_marginalBestBidAsk` and the per-bin executable prices then do `baseSpreadE6 + addFee` the same way. So `getBestBidAndAsk`, the reference bid/ask in `getLiquidityDepth`, and the ladder's executable prices are all affected.

Impact

Read-only impact – no funds at risk and actual swaps are unaffected, since they use the pool's own math. The effect is that any off-chain consumer of the lens (front-ends, routers, aggregators, price displays, slippage estimates) gets inaccurate bid/ask, reference prices, and depth executable prices. The error grows with the configured `spreadFeeE6` (which is added as if it were spread) and ignores the oracle spread, so the quotes can be off by a large margin under normal configurations.

Recommendation

Use the oracle base fee instead of the protocol/admin split.

Apply the same change to `_loadDepthEnv` (set `env.baseSpreadE6` from the oracle base fee, not `protocolSpreadFeeE6 + adminSpreadFeeE6`) so the depth ladder and reference prices use the real trader-facing fee. The `mid = sqrt(bid*ask)` construction makes the base fee symmetric, so one `baseFeeE6` is correct for both the buy and sell sides.

Issue L-21: Deposit allowlist has no whitelist only period end handling

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/115>

Summary

DepositAllowlistExtension gates deposits per-address with no global "allow all" switch, so there is no way to end the allowlist-only period and open the pool to everyone.

Vulnerability Details

beforeAddLiquidity reverts unless the depositor is individually allowed, and the only admin control is setAllowedToDeposit(pool, depositor, allowed) for one address at a time:

```
function beforeAddLiquidity(address, address owner, uint80, LiquidityDelta
    ↪ calldata, bytes calldata)
    external view override returns (bytes4)
{
    if (!allowedDepositor[msg.sender][owner]) revert
    ↪ IMetricOmmPoolActions.NotAllowedToDeposit();
    return IMetricOmmExtensions.beforeAddLiquidity.selector;
}
```

To go from allowlist-only to public, the admin would have to whitelist every depositor individually, which is not feasible for an open set.

Impact

Operational only – no funds at risk. Once the extension is active, the pool is permanently restricted to explicitly listed depositors; there is no path to a fully open deposit phase.

Recommendation

Add a per-pool public flag that bypasses the per-address check:

```
mapping(address pool => bool) public allowAll;

function setAllowAll(address pool_, bool enabled) external onlyPoolAdmin(pool_) {
    allowAll[pool_] = enabled;
    emit AllowAllSet(pool_, enabled);
}
```

```
// in beforeAddLiquidity:  
if (!allowAll[msg.sender] && !allowedDepositor[msg.sender][owner]) {  
    revert IMetricOmmPoolActions.NotAllowedToDeposit();  
}
```

Discussion

Oxklapouchy

Fix confirmed in [PR#42](#)

Issue L-22: Pool constructor does not emit initialization events for the price provider, fees, and per-bin fees [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/116>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `MetricOmmPool` constructor sets `priceProvider`, `spreadFeeE6`, `notionalFeeE8`, and the per-bin additional fees without emitting the matching events (`PriceProviderUpdated`, `SpreadFeeUpdated`, `NotionalFeeUpdated`, `BinAdditionalFeesUpdated`), unlike the setters for the same state which do emit them.

Vulnerability Detail

The setters emit on every change (`PriceProviderUpdated`, `SpreadFeeUpdated`, `NotionalFeeUpdated`, and `BinAdditionalFeesUpdated` for per-bin fees). The constructor writes the same state silently:

```
priceProvider = priceProvider_; // no PriceProviderUpdated
...
spreadFeeE6   = spreadFeeE6_;    // no SpreadFeeUpdated
notionalFeeE8 = notionalFeeE8_;  // no NotionalFeeUpdated
// per-bin additional fees set in the bin-state loop // no BinAdditionalFeesUpdated
```

An indexer reconstructing pool configuration from events therefore sees the price provider, pool fees, and per-bin fees only after the first post-deploy setter call, not at deploy.

Impact

Off-chain indexers and integrators miss the initial configuration at deploy time.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/33d0a3431fb711cfb84052d8380a09be69beef01/contracts/MetricOmmPool.sol#L129>
<https://github.com/Metric-OMM/metric-core/blob/33d0a3431fb711cfb84052d8380a09be69beef01/contracts/MetricOmmPool.sol#L139-L140> <https://github.com/Metric-OMM/metric-core/blob/33d0a3431fb711cfb84052d8380a09be69beef01/contracts/interfaces/IMetricOmmPool/IMetricOmmPoolFactoryActions.sol#L12-L31>

Tool Used

Manual Review

Recommendation

Emit the existing events from the constructor for the initial values it sets: `PriceProviderUpdated`, `SpreadFeeUpdated`, `NotionalFeeUpdated`, and one `BinAdditionalFeesUpdated` per initialized bin, mirroring the setters.

Discussion

konrad-metric

The Factory emits this info in `PoolCreated` event. The indexer must subscribe for this event anyway to notice newly created pools.

0xklapouchy

Acknowledged

Issue L-23: MetricOmmPool external functions are missing @inheritdoc NatSpec

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/117>

Summary

Most external functions in `MetricOmmPool` that implement an interface method carry no `/// @inheritdoc` tag, so they inherit no documentation. Only `inSwap`, `getImmutables`, and `getSellAndBuyPrices` use it.

Vulnerability Detail

The following external functions implement a method declared in an interface (`IMetricOmmPoolActions`, `IMetricOmmPoolCollectFees`, `IMetricOmmPoolFactoryActions`) but do not prefix it with `/// @inheritdoc <Interface>`:

Function	Interface
<code>addLiquidity</code>	<code>IMetricOmmPoolActions</code>
<code>removeLiquidity</code>	<code>IMetricOmmPoolActions</code>
<code>swap</code>	<code>IMetricOmmPoolActions</code>
<code>simulateSwapAndRevert</code>	<code>IMetricOmmPoolActions</code>
<code>collectFees</code>	<code>IMetricOmmPoolCollectFees</code>
<code>setPoolFees</code>	<code>IMetricOmmPoolFactoryActions</code>
<code>setPause</code>	<code>IMetricOmmPoolFactoryActions</code>
<code>setBinAdditionalFees</code>	<code>IMetricOmmPoolFactoryActions</code>
<code>setPriceProvider</code>	<code>IMetricOmmPoolFactoryActions</code>

Impact

Generated docs and tooltips for these functions are empty, and the file is internally inconsistent with the three functions that do use `@inheritdoc`.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/33d0a3431fb711cfb84052d8380a09be69beef01/contracts/MetricOmmPool.sol#L178>

<https://github.com/Metric-OMM/metric-core/blob/33d0a3431fb711cfb84052d8380a09be69beef01/contracts/MetricOmmPool.sol#L194>

Tool Used

Manual Review

Recommendation

Add `/// @inheritdoc <Interface>` above each of the nine functions, matching the pattern already used by `inSwap`, `getImmutables`, and `getSellAndBuyPrices`.

Discussion

Oxklapouchy

Fix confirmed in [PR#60](#)

Issue L-24: Unused internal function `_getMidPriceAndBaseFeeX64` in `MetricOmmPool`

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/118>

Summary

`MetricOmmPool._getMidPriceAndBaseFeeX64` is defined but never called anywhere in the codebase. It is dead code.

Vulnerability Detail

The internal function `_getMidPriceAndBaseFeeX64()` has no call site in `MetricOmmPool` or any other contract. It is redundant and only adds bytecode and review surface.

Impact

Informational. Dead code only, no functional or security impact.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/33d0a3431fb711cfb84052d8380a09be69beef01/contracts/MetricOmmPool.sol#L802>

Tool Used

Manual Review

Recommendation

Remove the unused `_getMidPriceAndBaseFeeX64` function.

Discussion

0xklapouchy

Fix confirmed in [PR#60](#)

Issue L-25: Pool bid/ask invariant check is weaker than the provider's strict $\text{bid} < \text{ask}$

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/119>

Summary

The price providers enforce a strict $\text{bid} < \text{ask}$ invariant: when $\text{bidOut} \geq \text{askOut}$ they return the stalled sentinel $(0, \text{type}(\text{uint128}).\text{max})$. The pool, however, only rejects $\text{bid} > \text{ask}$ (strict greater, not $\geq$), so a non-sentinel zero-spread quote ($\text{bid} == \text{ask}$) would pass the pool's check. The hard invariant is enforced on the provider side but not fully mirrored in the pool.

Vulnerability Detail

Each provider rejects equal bid and ask, treating it as stalled:

```
if (bidOut >= askOut) return (0, type(uint128).max);
```

The pool consumes the provider price in `_getBidAndAskPriceX64` and validates it with a weaker comparison:

```
if (bid > ask) revert BidGreaterThanAsk();  
if (bid == 0) revert BidIsZero();
```

The pool uses $>$ rather than $\geq$, so a $\text{bid} == \text{ask}$ (zero spread) quote is not rejected by `BidGreaterThanAsk`. For the in-scope providers this is unreachable, because they emit the $(0, \text{type}(\text{uint128}).\text{max})$ sentinel instead of a real $\text{bid} == \text{ask}$, and the pool catches that sentinel through the $\text{bid} == 0$ check. But the pool does not itself enforce the strict invariant, so a custom or future provider returning a non-zero $\text{bid} == \text{ask}$ would feed a zero-spread price into the pool, yielding $\text{baseFeeX64} = \text{sqrt}(\text{ask}/\text{bid}) - 1 = 0$. The same strict $\text{bidOut} \geq \text{askOut}$ rejection exists in `PriceProvider`, `PriceProviderL2`, and `ProtectedPriceProviderL2`.

Impact

Informational. The strict $\text{bid} < \text{ask}$ invariant is guaranteed by the in-scope providers but only loosely mirrored in the pool ($\text{bid} > \text{ask}$). No impact with the in-scope providers; a non-conforming provider returning $\text{bid} == \text{ask}$ would not be rejected by the pool.

Code Snippet

<https://github.com/Metric-OMM/metric-core/blob/33d0a3431fb711cfb84052d8380a09be69beef01/contracts/MetricOmmPool.sol#L798-L799>

<https://github.com/Oracle-Based-Pool/smart-contracts-poc/blob/edbc5f0e/contracts/ProtectedPriceProvider.sol#L237>

Tool Used

Manual Review

Recommendation

Mirror the provider's strict invariant in the pool: change the check to `if (bid >= ask) revert BidGreaterThanAsk();` so the pool defensively rejects zero-spread quotes regardless of which provider is configured.

Discussion

0xklapouchy

Fix confirmed in [PR#60](#)

Issue L-26: Missing publisher count validation lets low quality prices flow into the oracle [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/121>

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability details

The consumer parses Pyth Lazer payloads and extracts the price, exponent and confidence for each feed. It has a nice property config system so you can decide at deploy time which fields every feed must carry

///	ID	Name	Size
///	0	Price	8
///	1	BestBidPrice	8
///	2	BestAskPrice	8
///	3	PublisherCount	2
///	4	Exponent	2
///	5	Confidence	8

The thing is nothing forces PublisherCount to be part of that config and even if it is present nothing checks its value. Look at what actually gets read out of each feed

```
switch pid
case 0 { price := signextend(7, shr(192, shl(8, w))) }
case 4 { expo := signextend(1, shr(240, shl(8, w))) }
case 5 { conf := shr(192, shl(8, w)) }
```

Only price, exponent, confidence and timestamp are pulled. PublisherCount falls into the default branch which does nothing. So the number of publishers behind a given price is completely ignored during parsing and streaming

A Lazer price aggregated from a single publisher is basically one party's word. It is way easier to push a stale or manipulated quote when only one source is contributing versus when you require say three or more publishers to agree. The whole point of an aggregated oracle is that many publishers smooth out bad data and reverting to a single publisher quietly removes that guarantee

Impact

Prices sourced from a single publisher or a very thin publisher set get accepted and normalized just like healthy multi publisher prices

```
// price <= 0 → default
if iszero(sgt(p, 0)) {
    mstore(slot, or(defaultPacked, shl(128, feedId)))
    continue
}

// Pack raw data with marker bit
mstore(...)
```

There is no floor on how many publishers stand behind these numbers. If a single publisher is compromised or just glitches the bad price becomes the protocol price. Since the USDT conversion price feeds into normalization for every other feed a bad single publisher USDT quote poisons all the downstream prices at once

Anything reading this oracle then acts on a bad number. That means mispriced swaps bad liquidations and drainable liquidity because the price the protocol trusts does not reflect the real market

Fix

Read PublisherCount during parsing and enforce a configurable minimum.

Then have an immutable minimum set in the constructor and reject any feed below it. Make PublisherCount a required property in the expected set so a payload can never omit it and dodge the check. Same idea applies to confidence a healthy update should carry both price and confidence and you can bound the spread so a price with a blown out confidence interval does not get treated as reliable

Issue L-27: One stale feed reverts the whole price update batch [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/122>

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability details

The store path loops over every feed in the batch and writes fresh prices into storage. Before writing it runs a freshness check on each feed's own timestamp

```
TimeMs ts = toTimeMs(tsMs);
ts.revertIfAfterBlockTimeWithDrift(MAX_TIME_DRIFT);

if (!__feedIds.contains(feedId)) {
    revert NotRegisteredFeedId(feedId);
}

if (ts.isAfter(__data[feedId].timestampMs)) {
    __data[feedId] = IOffchainOracle.OracleData({
        price: normPrice,
        spread: spreadU.toUint16(),
        volatility: 0xFFFF,
        timestampMs: ts
    });
}
```

The problem is this reverts the entire call the moment any single feed's timestamp is too far ahead of block time. A batch normally carries many feeds and they do not all share the exact same timestamp. If one feed comes in with a timestamp outside the drift window the whole update dies and none of the other perfectly healthy feeds get stored

Notice the code already handles a missing timestamp gracefully right above it

```
if (tsMs == 0) continue;
```

so the pattern of skipping a bad single feed instead of killing the batch is already established here. The drift check just does not follow it

Impact

A batch update is all or nothing on the drift check. One feed with a slightly off timestamp blocks every other feed in the same payload from updating. So good prices get thrown away because of one unrelated bad entry

This is mostly a liveness and freshness issue rather than a direct fund loss. Prices go stale when they did not need to and a publisher that pushes one feed with a bad clock can stall the whole batch. Low severity but worth fixing since the healthy feeds should not pay for one bad one.

Fix

Treat a feed that fails the drift check the same way a missing timestamp is treated just skip that feed and let the rest of the batch proceed. Instead of a hard revert branch on the drift condition and continue

```
TimeMs ts = toTimeMs(tsMs);  
if (ts.isAfterBlockTimeWithDrift(MAX_TIME_DRIFT)) continue; // skip this feed, keep  
↪ the batch
```

That way a single stale or skewed feed only loses its own update and every other fresh feed in the batch still gets written

Issue L-28: Missing confidence lets the spread collapse to zero [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/123>

This issue has been acknowledged by the team but won't be fixed at this time.

Vulnerability details

When a feed gets normalized the spread is derived straight from the confidence value that came out of the payload

```
uint64 conf = (raw >> 112 & X64).toUint64();  
...  
spreadU = Math.ceilDiv(BPS_BASE * uint256(conf), pU);  
if (spreadU > BPS_BASE) spreadU = BPS_BASE;
```

The code never verifies that confidence was actually present in the report. Following the Pyth Lazer sdk some reports carry a valid price timestamp and exponent but simply do not include a confidence field.

<https://github.com/pyth-network/pyth-crosschain/blob/52543d72ac45481889f1aa29f1d19464fc98c2db/lazer/contracts/evm/src/PythLazerLib.sol#L366-L371>

In that case conf ends up as 0 and the spread becomes

```
spreadU = Math.ceilDiv(BPS_BASE * 0, pU); // = 0
```

So a feed that just omitted confidence gets recorded with a spread of 0 which reads as a perfectly tight price with no uncertainty at all. That is the exact opposite of what a missing confidence should mean. Missing confidence means we know less about the price not more

Worth noting the property parser only pulls price exponent and confidence and everything else falls through, so nothing forces confidence to be part of the expected property set and nothing rejects a feed that came without it

Impact

A price with unknown confidence gets stored as if it were maximally tight

- Any consumer that uses spread to size slippage bounds or to decide how much to trust a quote will treat this price as rock solid
- It causes loss of base fee which uses this price reports

Fix

Require confidence to be present. First make confidence a required property in the expected set so a feed cannot silently drop it. Then when confidence is missing or zero, skip that feed report.

Issue L-29: Oracle blacklist bypass [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/124>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The oracle's pool blacklist is meant to be an admin-only control, but the permissionless `register()` function clears it. Anyone can pay the registration fee – which defaults to 1 wei – to remove a pool from the blacklist, so an admin cannot durably block a pool from reading prices.

Vulnerability Details

Blacklisting is admin-gated:

```
function setBlacklist(address account, bool value) external onlyRole(ADMIN_ROLE) {  
    ↪ ... }
```

and it gates the on-chain price read (`price()` requires `!blacklisted[pool]`). But `register()` is permissionless and payable, and clears the blacklist as a side effect:

```
function register(bytes32 feedId, address pool, address factory) external payable  
    ↪ feedExists(feedId) {  
    require(msg.value >= registrationFee, InsufficientFee(msg.value,  
    ↪ registrationFee));  
    ...  
    if (blacklisted[pool]) {  
        blacklisted[pool] = false;  
        emit BlacklistUpdated(pool, false);  
    }  
    registeredPool[feedId][pool] = true;  
    ...  
}
```

The only requirements are that `factory` is approved and recognizes `pool` via `isPool` – which the blacklisted pool already satisfies. The registration fee defaults to 1 wei (`registrationFee = 1 wei` in the constructor), so lifting the blacklist is effectively free. The admin can re-blacklist, but the caller can immediately re-`register()`, producing a grieving loop; raising `registrationFee` to make this costly also raises the cost of every legitimate registration.

Impact

Operational control only – no funds are at risk. The admin blacklist does not reliably keep a pool from using the oracle: any blacklisted pool can be un-blacklisted permissionlessly for the registration fee (1 wei by default), defeating the purpose of the blacklist.

Recommendation

Don't let a permissionless path clear an admin control. Reject blacklisted pools in `register()` instead of un-blacklisting them, and leave lifting the blacklist to `setBlacklist`:

```
require(!blacklisted[pool], Blacklisted(pool));
```

(in place of the `if (blacklisted[pool]) { ... } block`).

Issue L-30: Chainlink oracle hardcoded decimal [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/125>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

ChainlinkOracle._toMid8 assumes every Data Streams report price is 18 decimals and always divides by a fixed PRICE_SCALE = 1e10. Chainlink documents that v3 reports may be 8 or 18 decimals. If a feed ever reports an 8-decimal price, dividing it by 1e10 yields a price ~1e10x too small, so the oracle would hand pools a near-zero mid. No real 8-decimal Data Streams feed is known today, so this is a latent risk rather than an active one.

Vulnerability Details

The decimal conversion is hardcoded:

```
uint256 internal constant PRICE_SCALE = 1e10;

/// @dev 18-decimal Data Streams price → 8-decimal OracleData price.
function _toMid8(int192 price) private pure returns (uint64) {
    require(price > 0, InvalidReportPrice());
    return (uint256(int256(price)) / PRICE_SCALE).toUint64();
}
```

Chainlink NatSpec explains that price has 8 or 18 decimals:

<https://github.com/smartcontractkit/chainlink-local/blob/f8c0efe8685660dac07e08f4558f1b578ae991aa/scripts/data-streams/ReportVersions.js#L106>

_toMid8 is used for the mid price of every schema (_normalizeV3, _normalizeV4, _normalizeHFS). It always assumes the input is 18 decimals and divides by 1e10 to reach the internal 8-decimal representation. There is no per-feed decimals check, and the v3/v4 report schemas don't carry a decimals field, so an 8-decimal report can't be detected on-chain – it would just be silently divided by 1e10 and come out ~1e10x too low (effectively zero after truncation).

That near-zero mid then feeds any pool using this oracle, which would price the token pair wildly wrong.

Impact

Latent, no funds at risk under current feeds – Chainlink Data Streams reports in use are 18 decimals, which this code handles correctly. The exposure is conditional: if an 8-decimal feed is ever configured (or Chainlink adds one), the reported mid is ~1e10x too

small and pools consuming it would be severely mispriced, opening the door to draining liquidity at the wrong price.

Recommendation

Make the decimals assumption explicit instead of a silent constant. Since the report itself doesn't declare decimals, store the expected report decimals per feed at registration and compute the scale from that, or explicitly restrict supported feeds to 18-decimal and reject/guard anything else, so an unexpected 8-decimal feed fails loudly rather than producing a near-zero price.

Issue L-31: Codebook256 table contains values outside the documented [0, 10000] bound [ACKNOWLEDGED]

Source: <https://github.com/sherlock-audit/2026-05-metric-may-22nd/issues/127>

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Codebook256.TABLE is documented as mapping each 8 bit index to a spread value in [0, 10000] bps with no duplicates. Decoding the table shows indices 251 to 254 decode to 10078, 10228, 10378, and 10528, all above 10000. Index 255 decodes to exactly 10000.

Detail

`decode(uint8 index)` only validates the index range, it never validates the returned value against `ORACLE_BPS.MAX_VALUE = 10_000` is declared in `Codebook256.sol` but never referenced anywhere in the codebase. `CompressedOracle.getOracleData` decodes both `spread0` and `spread1` from this table with no clamp. `spread0` is implicitly caught downstream, `PriceProvider.sol` and `ProtectedPriceProvider.sol` treat it as stalled using `spread >= ORACLE_BPS (ORACLE_BPS = 10_000)`, so index 255 (value 10000, within the documented range) is also treated as stalled, alongside indices 251 to 254 (genuinely out of range). `spread1` is not caught anywhere, `PriceProvider._getBidAndAskPrice` reads `(mid, spread, , refTime) = price(...)`, discarding `spread1` entirely, so it is never validated by anything in the codebase. No duplicate values exist elsewhere in the table.

Impact

Compressed oracle writes are namespace isolated per creator, so this can only affect a creator's own feed, never another creator's. `spread0` landing on codebook index 251 through 255 safely degrades to a stalled read. `spread1` landing on 251 to 254 is returned unclamped (values up to 10528) to any consumer that calls `getOracleData/price` directly rather than through `PriceProvider`.

Code Snippet

<https://github.com/Oracle-Based-Pool/smart-contracts-poc/blob/edbc5f0e/contracts/oracles/utils/Codebook256.sol#L7> <https://github.com/Oracle-Based-Pool/smart-contracts-poc/blob/edbc5f0e/contracts/oracles/utils/Codebook256.sol#L11>

Tool Used

Manual Review

Recommendation

Clamp both decoded values against MAX_VALUE/ORACLE_BPS in `_decodeCodebookIndex` or `getOracleData`, or regenerate TABLE so every entry stays within [0, 10000].

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.