

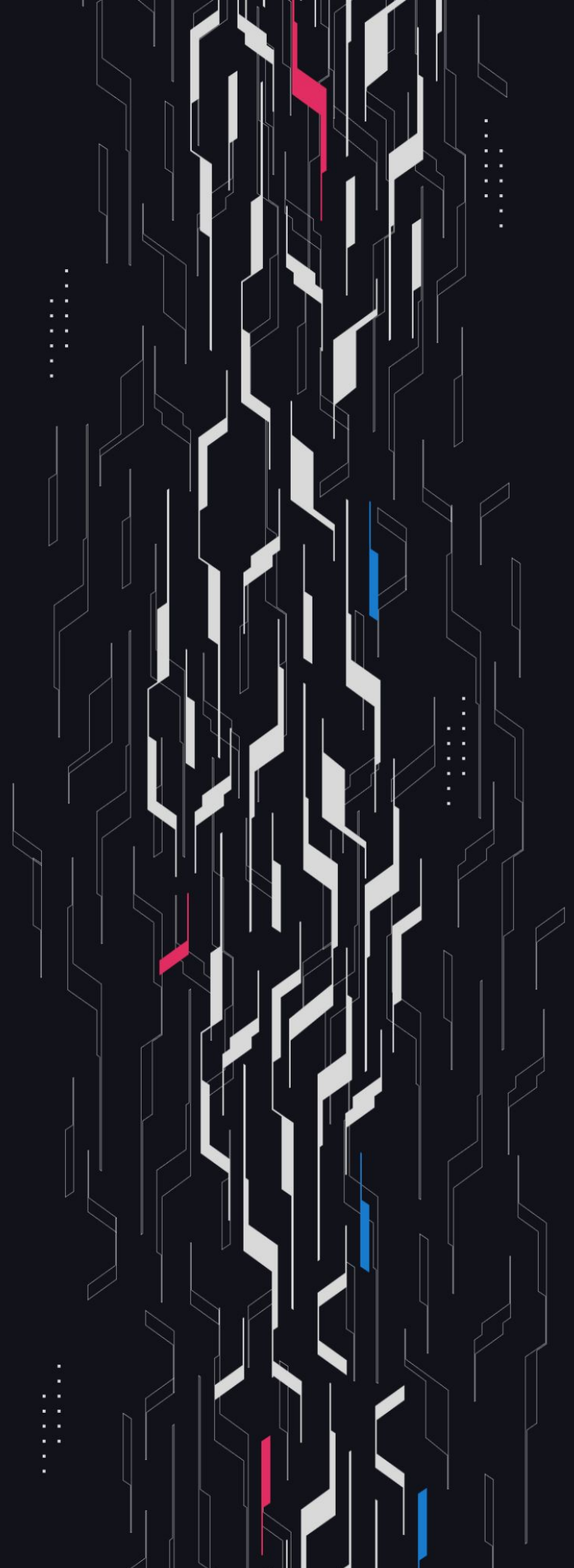
GA GUARDIAN

Olympus

Price Feed Updates

Security Assessment

May 14th, 2026



Summary

Audit Firm Guardian

Prepared By Robert Reigada, Wafflemakr

Client Firm Olympus

Final Report Date May 14, 2026

Audit Summary

Olympus engaged Guardian to review the price feed updates in scope. From March 30 through April 9, 2026, a team of 2 auditors reviewed the source code and recorded the findings and remediation results in this report.

Confidence Ranking

Given the minimal High and Critical issues detected during the main review, Guardian assigns a Confidence Ranking of 4 to the protocol. Guardian advises the protocol to consider periodic review with future changes. For detailed understanding of the Guardian Confidence Ranking, please see the rubric on the following page.

Note: Fixes to the findings uncovered in the remediation review have not been reviewed by Guardian. Guardian recommends the client either perform an amended follow-up review period to focus on the fixes to remediations findings or seek a follow-up audit from another team.

✓ Verify the authenticity of this report on Guardian's GitHub: <https://github.com/guardianaudits>

Guardian Confidence Ranking

Confidence Ranking	Definition and Recommendation	Risk Profile
5: Very High Confidence	Codebase is mature, clean, and secure. No High or Critical vulnerabilities were found. Follows modern best practices with high test coverage and thoughtful design. Recommendation: Code is highly secure at time of audit. Low risk of latent critical issues.	0 High/Critical findings and few Low/Medium severity findings.
4: High Confidence	Code is clean, well-structured, and adheres to best practices. Only 1 Significant issue was uncovered per week. Design patterns are sound, and test coverage is strong. Recommendation: Suitable for deployment after remediations; consider periodic review with changes.	0-1 High/Critical findings per engagement week and little to no Medium severity issues. Varied Low severity findings.
3: Moderate Confidence	Medium-severity and occasional High-severity issues found. Code is functional, but there are concerning areas (e.g., weak modularity, risky patterns). No critical design flaws, though some patterns could lead to issues in edge cases. Recommendation: Address issues thoroughly and consider a targeted follow-up audit depending on code changes.	1-2 High/Critical findings per engagement week.
2: Low Confidence	Code shows frequent emergence of Critical/High vulnerabilities. Audit revealed recurring anti-patterns, weak test coverage, or unclear logic. These characteristics suggest a high likelihood of latent issues. Recommendation: Post-audit development and a second audit cycle are strongly advised.	2-4 High/Critical findings per engagement week. Or additional High/Critical findings uncovered in remediation review which have not been resolved and confirmed by Guardian.
1: Very Low Confidence	Code has systemic issues. Multiple High/Critical findings (≥ 5/week), poor security posture, and design flaws that introduce compounding risks. Safety cannot be assured. Recommendation: Halt deployment and seek a comprehensive re-audit after substantial refactoring.	≥ 5 High/Critical findings and overall systemic flaws.

Table of Contents

Project Information

Project Overview 5

Audit Scope & Methodology 6

Smart Contract Risk Assessment

Findings & Resolutions 9

Addendum

Disclaimer 74

About Guardian 75

Project Overview

Project Summary

Project Name	Olympus Price Feed Updates
Codebase	github.com/OlympusDAO/olympus-v3
Commit(s)	Main Review: 1b03f97fa84538c11d42bc4dceab577081cdd8e6 Remediation: ee1b8330d39a2829c975341187795a39ee8a195b

Audit Summary

Delivery Date	May 14, 2026
Audit Methodology	Static Analysis, Manual Review

Vulnerability Summary

Vulnerability Level	Total	Pending	Declined	Acknowledged	Partially Resolved	Resolved
● Critical	0	0	0	0	0	0
● High	1	0	0	0	0	1
● Medium	15	0	0	1	2	12
● Low	25	0	0	10	1	14
● Info	19	0	0	4	0	15

Audit Scope & Methodology

src/proposals/OracleProposal.sol,240,297,26
src/interfaces/IPyth.sol,9,26,15
src/policies/price/BaseOracleFactory.sol,221,389,97
src/policies/price/ChainlinkOracleCloneable.sol,130,297,126
src/policies/price/ChainlinkOracleFactory.sol,32,71,34
src/policies/price/ERC7726OracleCloneable.sol,90,161,45
src/policies/price/ERC7726OracleFactory.sol,183,280,40
src/policies/price/MorphoOracleCloneable.sol,90,196,76
src/policies/price/MorphoOracleFactory.sol,50,110,49
src/policies/price/PriceConfig.v2.sol,137,209,37
src/modules/PRICE/IPRICE.v2.sol,95,304,170
src/modules/PRICE/OlympusPrice.v1_2.sol,78,186,75
src/modules/PRICE/OlympusPrice.v2.sol,401,851,330
src/modules/PRICE/PRICE.v2.sol,30,67,21
src/scripts/ops/batches/ConfigureOracles.sol,50,103,32
src/scripts/ops/batches/ConfigurePriceV1_2.sol,412,621,131
src/policies/interfaces/price/IChainlinkOracle.sol,7,29,15
src/policies/interfaces/price/IERC7726Oracle.sol,7,27,15
src/policies/interfaces/price/IERC7726OracleFactory.sol,19,82,43
src/policies/interfaces/price/IERC7726OraclePriceCache.sol,3,14,10
src/policies/interfaces/price/IMorphoOracle.sol,7,32,18
src/policies/interfaces/price/IOracleFactory.sol,24,103,58
src/policies/interfaces/price/IOraclePriceCache.sol,3,11,7
src/policies/interfaces/price/IPriceOracle.sol,3,11,7
src/modules/PRICE/submodules/strategies/ISimplePriceFeedStrategy.sol,9,37,21
src/modules/PRICE/submodules/strategies/SimplePriceFeedStrategy.sol,278,688,311
src/modules/PRICE/submodules/feeds/ChainlinkPriceFeeds.sol,189,365,129
src/modules/PRICE/submodules/feeds/ERC4626Price.sol,54,145,64
src/modules/PRICE/submodules/feeds/PythPriceFeeds.sol,207,440,177
src/modules/PRICE/submodules/feeds/UniswapV3Price.sol,130,309,129

Audit Scope & Methodology

Vulnerability Classifications

Severity	Impact: <i>High</i>	Impact: <i>Medium</i>	Impact: <i>Low</i>
Likelihood: <i>High</i>	● Critical	● High	● Medium
Likelihood: <i>Medium</i>	● High	● Medium	● Low
Likelihood: <i>Low</i>	● Medium	● Low	● Low

Impact

- High** Significant loss of assets in the protocol, significant harm to a group of users, or a core functionality of the protocol is disrupted.
- Medium** A small amount of funds can be lost or ancillary functionality of the protocol is affected. The user or protocol may experience reduced or delayed receipt of intended funds.
- Low** Can lead to any unexpected behavior with some of the protocol's functionalities that is notable but does not meet the criteria for a higher severity.

Likelihood

- High** The attack is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount gained or the disruption to the protocol.
- Medium** An attack vector that is only possible in uncommon cases or requires a large amount of capital to exercise relative to the amount gained or the disruption to the protocol.
- Low** Unlikely to ever occur in production.

Audit Scope & Methodology

Methodology

Guardian is the ultimate standard for Smart Contract security. An engagement with Guardian entails the following:

- Two competing teams of Guardian security researchers performing an independent review.
- A dedicated fuzzing engineer to construct a comprehensive stateful fuzzing suite for the project.
- An engagement lead security researcher coordinating the 2 teams, performing their own analysis, relaying findings to the client, and orchestrating the testing/verification efforts.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross-referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.
Comprehensive written tests as a part of a code coverage testing suite.
- Contract fuzzing for increased attack resilience.

Round One Findings & Resolutions

ID	Title	Category	Severity	Status
H-01	Public Cache Refresh Can Brick Pair Oracles	DoS	● High	Resolved
M-01	OHM Target Price Is Reset From Deployment Seed	Upgradeability	● Medium	Resolved
M-02	Single OHM Feed Outage Stalls Heartbeat	DoS	● Medium	Resolved
M-03	Chainlink Clone Serves Stale Cached Prices	Validation	● Medium	Resolved
M-04	Public Cache Writes Bypass Operator Stale Check	Unexpected Behavior	● Medium	Resolved
M-05	Public OHM Cache Commits Make Price Writable	Oracle	● Medium	Partially Resolved
M-06	Deviation Exclusion Runs Only One Filter Pass	Unexpected Behavior	● Medium	Resolved
M-07	Max-age Reads Accept Stale Moving-average Cache	Unexpected Behavior	● Medium	Resolved
M-08	getQuote Truncation Amplification	Rounding	● Medium	Resolved
M-09	Stale MA Silently Used As Target Price	Validation	● Medium	Resolved
M-10	Insufficient Pool Cardinality	DoS	● Medium	Resolved
M-11	2-price Path Bricks WETH Pricing	DoS	● Medium	Resolved
M-12	Dead OHM/sUSDS Leg Can Stall OHM Pricing	Oracle	● Medium	Partially Resolved

Round One Findings & Resolutions

ID	Title	Category	Severity	Status
M-13	Same-block Mixed Caches Can Pass Validation	Oracle	● Medium	Resolved
L-01	Oracle Validation Checks Base USD Price	Logical Error	● Low	Resolved
L-02	AveragelfDeviation Returns Min, Not First	Logical Error	● Low	Resolved
L-03	ERC4626 Oracle Rejects Valid Decimal Offsets	Compatibility	● Low	Resolved
L-04	Feed Failure Blocks Asset Reconfiguration	Configuration	● Low	Acknowledged
L-05	UniswapV3Price Accepts Counterfeit Pools	Validation	● Low	Partially Resolved
L-06	Pyth Two-feed Prices Ignore Derived Confidence	Validation	● Low	Resolved
L-07	PriceConfigv2 Cannot Update V1.2 Target Floor	Compatibility	● Low	Acknowledged
L-08	Pyth Confidence Conversion Overflow Bricks Feed	Math	● Low	Resolved
L-09	Ignored Field Alters Cache	Configuration	● Low	Resolved
L-10	Non-MA Clones Have No Cache Refresh	DoS	● Low	Resolved
L-11	No Upper Bound On minimumTargetPrice	Validation	● Low	Acknowledged
L-12	Derived Feed Threshold Exceeds Heartbeat	Configuration	● Low	Resolved

Round One Findings & Resolutions

ID	Title	Category	Severity	Status
L-13	Wrong Oracle Source Passes Validation	Validation	● Low	Resolved
L-14	CURRENT Reports Block Time For Degraded Inputs	Oracle	● Low	Acknowledged
L-15	Runtime Writes Persist Degraded Fallback Prices	Warning	● Low	Acknowledged
L-16	Batch Can Activate A Factory From Another Kernel	Validation	● Low	Resolved
L-17	Correlated Chainlink Routes Overstate Quorum	Warning	● Low	Acknowledged
L-18	Factory Re-enable Can Expose Stale Oracle Clones	Warning	● Low	Resolved
L-19	Disabling MA Leaves Stale Asset Metadata	Unexpected Behavior	● Low	Resolved
I-01	ERC4626 Oracle Uses Idealized Share Value	Warning	● Info	Resolved
I-02	ConfigureOracles Uses Missing Env Keys	Configuration	● Info	Resolved
I-03	OracleProposal Missing Factory Activation	Configuration	● Info	Resolved
I-04	DeactivatePolicy Does Not Stop Clones	Warning	● Info	Resolved
I-05	Double storeObservation Corrupts MA	Logical Error	● Info	Resolved
I-06	Static Price Bounds Block Deployment	Configuration	● Info	Resolved

Round One Findings & Resolutions

ID	Title	Category	Severity	Status
I-07	Variant.LAST Semantic Changed By Refactor	Logical Error	● Info	Resolved
I-08	updateAsset Locked By Stale MA	DoS	● Info	Resolved
I-09	Split PRICE Upgrade Leaves OHM Unapproved	Upgradeability	● Info	Resolved
I-10	Clone Assumes ERC20 Decimals For All Assets	Warning	● Info	Resolved
I-11	Health Checks Ignore Disabled Oracle State	Warning	● Info	Resolved
I-12	ERC4626 Adapter Assumes Trusted Share Rate	Oracle	● Info	Resolved
I-13	ConfigureOracles Cannot Be Safely Retried	Warning	● Info	Resolved
I-14	Unchecked Casts Can Rewrite PRICE Batch Args	Warning	● Info	Resolved

H-01 | Public Cache Refresh Can Brick Pair Oracles

Category	Severity	Location	Status
DoS	● High	PriceConfig.v2.sol	Resolved

Description

`PriceConfigv2.cachePrice()` is permissionless once the policy is enabled, but the pair adapters read `Variant.LAST` for each asset and require the two cache timestamps to match exactly. `ChainlinkOracleCloneable.latestRoundData()` reverts on any mismatch, and the same pattern is repeated in `MorphoOracleCloneable.price()` and `ERC7726OracleCloneable.getQuote()`. That makes the pair oracles incompatible with one-sided cache updates. A caller can refresh OHM without refreshing USDS, which moves only one `LAST.cachedAt` forward and causes every OHM/USDS adapter read to revert with an inconsistent-timestamp error until both assets are cached again at the same timestamp. This is not limited to the public policy path. The generic ERC7726 clone can also be used as a public cache writer because `ERC7726OracleFactory.cachePrices()` does not bind `base_` and `quote_` to any immutable pair, so a caller can route `cachePrices(asset, asset)` through the deployed clone and refresh a single asset that way as well. The result is a cheap, repeatable DoS on the newly deployed oracle surfaces. Any integration that depends on the OHM/USDS Chainlink, Morpho, or ERC7726 adapters can be forced to revert until a repair transaction re-synchronizes the two cache entries.

Recommendation

Do not combine exact timestamp equality with permissionless single-asset cache writes. The safest fix is to make the pair adapters validate freshness per asset instead of requiring `baseTimestamp == quoteTimestamp`, and then use `min(baseTimestamp, quoteTimestamp)` as the reported timestamp. If exact equality must be preserved, remove the public single-asset cache path for assets used by pair oracles and add pair validation to `ERC7726OracleFactory.cachePrices()` so the generic clone cannot refresh arbitrary assets.

Resolution

Olympus: Resolved.

M-01 | OHM Target Price Is Reset From Deployment Seed

Category	Severity	Location	Status
Upgradeability	● Medium	ConfigurePriceV1_2.sol	Resolved

Description

`ConfigurePriceV1_2._configureOhm()` does not initialize OHM's moving-average state from the live PRICE v1.1 module. Instead, it reads a single `ohmInitialPrice` from the batch arguments, fills all 21 entries of the seven-day observation window with that value, and stores the seeded array as the initial OHM moving-average history for PRICE v1.2. That means the upgrade does not preserve the live target-price state. It replaces it with an operator-chosen deployment input. The only guard on that input is a broad sanity range check between `17e18` and `22e18`, which is not tied to the actual live OHM target at the time of execution. This directly affects the compatibility path exposed by `OlympusPricev1_2.getTargetPrice()` returns `max(movingAverage, minimumTargetPrice)`, so any seeded moving average above the live target immediately becomes the new protocol target price even though no new market observations justified the change. On a mainnet fork at block `24,792,420`, the live PRICE v1.1 target price was `16.522383181232041915e18`. Replaying the PR rollout with the batch logic from this PR set the upgraded PRICE v1.2 target price to exactly `20e18`. The impact is not limited to read-only oracle drift. `Operator` consumes `PRICE.getTargetPrice()` in both `_updateRangePrices()` and `_addObservation()`. As a result, the seeded value propagates into RANGE pricing and regeneration logic as soon as the upgrade is live. The protocol starts operating from an inflated OHM target instead of the real pre-upgrade state. The issue also persists longer than a single block. The batch seeds a full seven-day window using 21 synthetic observations. Even if later heartbeats store real prices, the inflated moving average decays only as those seeded observations age out. For roughly one moving-average window, the protocol can remain anchored to deployment-time input rather than market-derived history.

Recommendation

Do not bootstrap OHM's moving average from a free-form deployment argument. The upgrade should migrate the live PRICE v1.1 state into PRICE v1.2, or reconstruct the initial observation window from live on-chain data at execution time. If full migration is too heavy, the minimum safe fallback is to derive the seed from the old module's live OHM target or recent stored observations and reject any proposed seed that deviates materially from that value. The rollout should also include a post-upgrade check that compares the new `getTargetPrice()` against the pre-upgrade value and reverts if the drift exceeds a narrow tolerance.

Resolution

Olympus: Resolved.

M-02 | Single OHM Feed Outage Stalls Heartbeat

Category	Severity	Location	Status
DoS	● Medium	ConfigurePriceV1_2.sol	Resolved

Description

The PR configures OHM to use `SimplePriceFeedStrategy.getAveragePrice()` in strict mode across exactly two live feeds: the `OHM/WETH` Uniswap V3 TWAP and the `OHM/sUSDS` Uniswap V3 TWAP. That combination leaves no tolerance for a single feed outage. In strict mode, `getAveragePrice()` reverts when fewer than two non-zero prices are available. So if either OHM path fails, reverts, or returns zero, OHM no longer has a current price even though the other path is still healthy. This becomes a protocol-availability issue because `Heart.beat()` begins by calling `PRICE.updateMovingAverage()`. Updating the moving average for OHM requires a fresh current OHM price. When one OHM feed path is missing, `PRICE` reverts with `PRICE_StrategyFailed(OHM, SimpleStrategy_PriceCountInvalid(1,2))`, and the heartbeat aborts before rebases or periodic tasks execute. The impact is broader than OHM spot-price availability. A stalled heartbeat blocks the protocol’s scheduled maintenance flow, including rebases, keeper rewards, and downstream periodic-task execution. In effect, one failed OHM path can halt the whole heartbeat loop until the missing path is restored.

Recommendation

- Do not run OHM in strict mode with only two feeds. Safer options are:
- Add a third independent OHM source so one failure still leaves two prices.
 - Disable strict mode so OHM can continue using the surviving path during a single-feed outage.
 - If heartbeat continuity is the priority, add a bounded fallback to the cached price or moving average when one live path is unavailable.

Resolution

Olympus: Resolved.

M-03 | Chainlink Clone Serves Stale Cached Prices

Category	Severity	Location	Status
Validation	● Medium	ChainlinkOracleCloneable.sol	Resolved

Description

`ChainlinkOracleCloneable.latestRoundData()` reads `PRICE.getPrice(..., Variant.LAST)` for both assets and never enforces `maxAge()` before returning the answer. Once a value is cached, the adapter keeps serving that cached ratio even after the configured freshness window has expired. This differs from the Morpho and ERC7726 adapters in the same PR. Those adapters explicitly revert when the shared cached timestamp is older than `maxAge()`. The Chainlink clone does not. The test suite confirms the behavior: after warping past `maxAge`, `latestRoundData()` still returns the old cached answer instead of rejecting it. That turns a short-lived bad cache into a longer-lived integration risk. If OHM is cached while one of its Uniswap TWAP inputs is manipulated, the OHM/USDS Chainlink clone can continue serving that stale manipulated answer until some caller refreshes the cache. Integrations that treat the adapter like a normal Chainlink feed but do not independently enforce a heartbeat may keep consuming an outdated price long after the intended freshness window has passed. A bad cached round can survive beyond the configured `maxAge` boundary.

Recommendation

Make `latestRoundData()` enforce the same freshness rule as the other clone adapters. The smallest safe change is to reuse the existing staleness check and revert when the shared cache timestamp is older than `maxAge()`. If Chainlink compatibility requires stale rounds to remain readable, then this adapter should not be used as a safety boundary on its own and the documentation and deployment scripts should require downstream consumers to reject old `updatedAt` values explicitly.

Resolution

Olympus: Resolved.

M-04 | Public Cache Writes Bypass Operator Stale Check

Category	Severity	Location	Status
Unexpected Behavior	● Medium	OlympusPrice.v1_2.sol	Resolved

Description

`OlympusPricev1_2.lastObservationTime()` no longer returns the timestamp of the last moving-average observation. It returns the timestamp of OHM `Variant.LAST`, which is the generic cache entry. That cache can be updated without storing a new observation. `PriceConfigv2.cachePrice()` is public and calls `PRICE.cachePrice(asset_)`, so any caller can refresh OHM's cached timestamp without advancing the moving-average ring buffer. This breaks a real legacy assumption in `Operator`. `Operator._onlyWhileActive()` treats `PRICE.lastObservationTime()` as the freshness check for whether RANGE data is still based on a recent observation. Under v1.2, a public `cachePrice(OHM)` call can make that check look fresh even though no new observation was stored and `Heart.beat()` never ran. The stale-data guard can therefore be bypassed, leaving swaps and wall activity enabled against stale RANGE state.

Recommendation

Keep `lastObservationTime()` backward-compatible and return the actual OHM observation timestamp, not the generic cache timestamp. The simplest fix is to read OHM's stored moving-average state directly. If that compatibility cannot be preserved, `Operator` should stop using `PRICE.lastObservationTime()` as its stale gate and instead read an explicit observation timestamp that public cache writes cannot refresh.

Resolution

Olympus: Resolved.

M-05 | Public OHM Cache Commits Make Price Writable

Category	Severity	Location	Status
Oracle	● Medium	ConfigurePriceV1_2.sol	Partially Resolved

Description

The OHM rollout is configured so the live OHM price comes only from two Uniswap V3 TWAP paths, OHM/WETH and OHM/sUSDS, resolved with `SimplePriceFeedStrategy.getAveragePrice()` in strict mode. The batch also seeds a seven-day moving average, but sets `storeMovingAverage = true` and `useMovingAverage = false`, so that moving average is stored and updated without being used in the live OHM pricing path. That matters because `OlympusPricev2.cachePrice()` snapshots `_getCurrentPrice()` into `Variant.LAST` and the public policy surface exposes that write through `PriceConfigv2.cachePrice()`. The clone adapters also expose public cache entrypoints that route back to the same cache write path. In other words, once OHM is configured this way, anyone can choose when the current OHM DEX-TWAP path is committed into the canonical cached value used by downstream adapters.

`UniswapV3Price.getTokenTWAP()` already warns that Uniswap V3 TWAPs can be manipulated with multi-block MEV. The chosen OHM strategy adds no divergence filter. With exactly two live OHM feeds, `getAveragePrice()` only checks that both are non-zero and then averages them. So if an attacker can bias either OHM pool for the configured TWAP window, the manipulated leg becomes half of the resolved OHM price instead of being excluded as an outlier. The main impact is that the OHM oracle path is not only manipulable, it is writable. An attacker can first bias the OHM TWAP inputs and then immediately call a public cache entrypoint to promote that manipulated live value into `LAST`. The OHM/USDS Chainlink, Morpho and ERC7726 adapters all read cached `LAST` prices, so the committed value becomes the quote surface seen by lending integrations. For Morpho and ERC7726 this persists until the cache ages out or someone refreshes it. For the Chainlink clone, standard reads do not enforce `maxAge`, so a stale manipulated round can survive even longer if the consumer does not reject it on `updatedAt`. This can misprice OHM collateral or debt in downstream integrations and enable bad borrows, bad liquidations, or insolvency if the manipulated cached price is accepted. Current deployment artifacts in this repository only show this OHM price path on Ethereum mainnet. Using live mainnet pool state from April 2, 2026 at block 24794064, the cheapest manipulation route is the OHM/WETH pool. A one-path manipulation held for the full 30-minute TWAP window needs roughly 46.9 WETH (about \$96.6k) to make the final OHM oracle read about 5% high, or 91.7 WETH (about \$189k) to make it read about 10% high. The OHM/sUSDS pool is materially more expensive at roughly \$444k and \$868k for those same outcomes. If the attacker can only hold the distortion for 10 minutes of the 30-minute window, the OHM/WETH path rises to roughly \$304k for a 5% final skew and roughly \$623k for a 10% final skew.

These figures show that the issue is not a cheap arbitrary-write primitive today. The expensive step is economically holding the OHM pool off-market long enough for the TWAP to move, after which the cache commit is trivial. In practice, exploitability depends on whether the value extractable from a downstream lending integration exceeds the capital, arbitrage leakage and operational risk needed to sustain that 30-minute distortion.

Recommendation

Do not expose permissionless cache commits for a lending-facing OHM price path that is derived only from DEX TWAPs. The safest fix is to harden the OHM price source and the cache-commit surface together. Use the stored moving average in the live OHM path, add a non-DEX OHM source, or switch the resolver to a deviation-filtering strategy such as `getAveragePriceExcludingDeviations()` so a single manipulated pool cannot be committed directly into `LAST`. If the protocol wants to keep a DEX-heavy OHM path, the permissionless cache entrypoints should not be allowed to write that value directly into the lending-facing cached surface.

Resolution

Olympus: Partially Resolved.

M-06 | Deviation Exclusion Runs Only One Filter Pass

Category	Severity	Location	Status
Unexpected Behavior	● Medium	SimplePriceFeedStrategy.sol	Resolved

Description

`getAveragePriceExcludingDeviations()` computes one median benchmark from the full input set, calls `_filterByDeviation()` once, and immediately averages the survivors. It never recomputes the benchmark after removing outliers. `getMedianPriceExcludingDeviations()` repeats the same pattern at `SimplePriceFeedStrategy.sol:620-650`. This is weaker than the selector's documented consensus behavior. A more extreme outlier can pull the initial median far enough that a second weaker outlier survives the first pass even though it no longer fits the benchmark formed by the remaining prices. For example, at a `200` bps threshold the set `[10000, 10000, 10400, 11000]` leaves `[10000, 10000, 10400]` after one pass and `getAveragePriceExcludingDeviations()` returns `10133`. Recomputing the median on the survivors gives `10000`, so an iterative filter would drop `10400` and return `10000` instead. This matters because the repository's v1.2 rollout config uses `getAveragePriceExcludingDeviations()` for WETH with four feeds and a `200` bps threshold. Therefore one bad feed can still bias the resolved price if another, more extreme outlier is present in the same sample. The issue does not require the surviving outlier to pass the tighter post-filter consensus. It only needs to survive the first benchmark.

Recommendation

Recompute the benchmark after each exclusion round and continue filtering until the survivor set stops changing or the minimum required count is no longer met. The smallest safe change is to wrap the current median-based filter in a loop and reuse the reduced array as the next input. If iterative consensus is not intended, the selector comments and interface docs should be corrected so integrators do not rely on a stronger guarantee than the code provides. [\[https://github.com/OlympusDAO/olympus-v3/pull/238\]](https://github.com/OlympusDAO/olympus-v3/pull/238)[\[https://github.com/OlympusDAO/olympus-v3/pull/238\]](https://github.com/OlympusDAO/olympus-v3/pull/238)

Resolution

Olympus: Resolved.

M-07 | Max-age Reads Accept Stale Moving-average Cache

Category	Severity	Location	Status
Unexpected Behavior	● Medium	OlympusPrice.v2.sol	Resolved

Description

`getPrice(asset_, maxAge_)` treats cache freshness as if it were full price freshness. That is not true for assets whose strategy uses the stored moving average. When `cachePrice()` or `storeObservation()` writes `Variant.LAST`, the cached value can include the moving average. `_getCurrentPrice()` checks `lastObservationTime` only at the moment the cache entry is created. After that, `getPrice(asset_, maxAge_)` returns the cached value whenever `cachedAt` is within `maxAge_`. It never checks whether the moving average inside that cached price has since gone stale. As a result, a caller can cache an MA-backed asset shortly before `observationFrequency` expires, wait until the moving average is stale, and still read the old inclusive price through `getPrice(asset_, maxAge_)` as long as the cache timestamp is recent enough. I confirmed this with a targeted local repro: after caching `ONEMA` while its moving average was fresh, `Variant.CURRENT` reverted with `PRICE_MovingAverageStale` after the observation window elapsed, but `getPrice(ONEMA, 2 * observationFrequency)` still returned the cached inclusive value. This weakens the freshness guarantee that callers are likely to infer from `maxAge_`. The result is not just an old spot price. It can be a price that still embeds an expired moving average. Any integration that relies on `getPrice(asset_, maxAge_)` or `getPriceIn(..., maxAge_)` as its freshness boundary can accept MA-backed prices longer than the module's moving-average staleness rule intends. A realistic case looks like this:

- `observationFrequency = 8 hours`
 - an oracle or integration uses `maxAge = 1 hour`
 - at 15:59, while the moving average is still fresh, someone calls `cachePrice(asset_)`
 - the 16:00 heartbeat is missed
 - after 16:00, a fresh read through `Variant.CURRENT` would revert with `PRICE_MovingAverageStale`
 - but until 16:59, `getPrice(asset_, 1 hours)` can still return the cached inclusive value because it only checks the cache timestamp, not whether the embedded moving average has expired
- That means the caller can receive a price that appears fresh under the `maxAge_` rule even though the moving-average component inside it is already stale.

Recommendation

When an asset uses the stored moving average, `getPrice(asset_, maxAge_)` should reject `Variant.LAST` once the underlying moving average is stale, even if `cachedAt` is still within `maxAge_`. The smallest safe change is to add a second check before returning the cached value:

```
if (
  asset.useMovingAverage &&
  asset.lastObservationTime + _observationFrequency <= block.timestamp
) {
  (uint256 price, , ) = _getCurrentPrice(asset_, true);
  return price;
}
```

`_getPriceStale()` and the `getPriceIn(..., maxAge_)` helpers should enforce the same rule so all max-age reads treat moving-average freshness consistently.

Resolution

Olympus: Resolved.

M-08 | getQuote Truncation Amplification

Category	Severity	Location	Status
Rounding	● Medium	ERC7726OracleCloneable.sol:102-105	Resolved

Description

ERC7726OracleCloneable._getQuoteInternal() splits the price conversion into two sequential FullMath.mulDiv calls. Step 1 divides by quotePriceUsd, truncating the intermediate result. Step 2 multiplies that truncated value by quoteTokenScale / baseTokenScale, amplifying the truncation error.

```
uint256 intermediate = inAmount_.mulDiv(basePriceUsd, quotePriceUsd); // truncates
outAmount_ = intermediate.mulDiv(quoteTokenScale, baseTokenScale); // amplifies
```

When the quote token has more decimals than the base token, the amplification factor is quoteTokenScale / baseTokenScale. For base=6 dec and quote=18 dec, the factor is 1e12. For base=0 dec and quote=18 dec, the factor is 1e18. In the worst case (base=0 dec, quote=18 dec, price ratio 1:3), step 1 computes $1 * 1e18 / 3e18 = 0$ via floor division. Step 2 cannot recover from a zero intermediate and returns 0 – a 100% loss for a valid non-zero input.

Recommendation

Fuse the two mulDiv calls so that quoteTokenScale is multiplied before dividing by quotePriceUsd, eliminating the amplification path:

```
outAmount_ = FullMath.mulDiv(
    inAmount_.mulDiv(basePriceUsd, baseTokenScale),
    quoteTokenScale,
    quotePriceUsd
);
```

This ensures truncation only occurs once at the end, after the full numerator is assembled.

Resolution

Olympus: Resolved.

M-09 | Stale MA Silently Used As Target Price

Category	Severity	Location	Status
Validation	● Medium	OlympusPrice.v2.sol:159-163	Resolved

Description

`getTargetPrice()` calls `getPrice(OHM, Variant.MOVINGAVERAGE)`, which returns `cumulativeObs / numObservations` with no staleness check on `lastObservationTime`. In contrast, `_getCurrentPrice()` explicitly checks `lastObservationTime + observationFrequency <= block.timestamp` and reverts with `PRICE_MovingAverageStale` when the MA is outdated. After a heartbeat gap (e.g., caused by a feed outage), `getCurrentPrice()` correctly reverts, but `getTargetPrice()` silently returns the stale MA value. `Operator.beat()` consumes `getTargetPrice()` to set RBS range prices via `_updateRangePrices()`, so the protocol resumes operating against a target that no longer reflects market conditions. This is distinct from the single-feed-outage heartbeat stall. This finding covers what happens after resumption: stale MA data is consumed without any error signal.

Recommendation

Add a staleness check to the `Variant.MOVINGAVERAGE` path in `OlympusPrice.v2.sol` (lines 159-163), mirroring the guard in `_getCurrentPrice()`:

```
if (asset.lastObservationTime + _observationFrequency <= block.timestamp)
    revert PRICE_MovingAverageStale(asset_, asset.lastObservationTime);
```

Alternatively, add the check directly in `getTargetPrice()` before returning the MA value.

Resolution

Olympus: Resolved.

M-10 | Insufficient Pool Cardinality

Category	Severity	Location	Status
DoS	● Medium	src/modules/PRICE/submodules/feeds/UniswapV3Price.sol:151	Resolved

Description

UniswapV3Price.getTokenTWAP() calls pool.observe([1800, 0]) to read a 30-minute TWAP. UniV3 stores observations in a circular buffer and writes at most one entry per block. If swap activity on a pool sustains one swap per block for ~26 minutes, all 128 buffer slots get overwritten with observations from the last 25.6 minutes (128 × 12s). The 30-minute lookback then has no observation old enough and observe() reverts with OLD. Both OHM pools have observationCardinality = 128 on mainnet. The pool's cardinality is an external property that Olympus does not control and that can become insufficient at any time as trading patterns change.

Recommendation

Validate at configuration time that pool.observationCardinality >= observationWindowSeconds / 12 for the target pool. Consider also monitoring cardinality as an operational concern.

Resolution

Olympus: Resolved.

M-11 | 2-price Path Bricks WETH Pricing

Category	Severity	Location	Status
DoS	● Medium	SimplePriceFeedStrategy.sol:478-504	Resolved

Description

`getAveragePriceExcludingDeviations` has two code paths: for exactly 2 non-zero prices it uses their average as the deviation benchmark (inline, lines 478-504); for 3+ prices it uses the median via `_filterByDeviation`. The average benchmark is strictly more sensitive than the median. When two prices diverge, each deviates equally from their midpoint, so a spread exceeding $2 \times deviationBps$ excludes both prices simultaneously and reverts. WETH is configured with 4 feeds at 200 bps (2%). If exactly 2 feeds fail (e.g., Pyth stale + RedStone offline), the 2 survivors enter the inline path. A spread >4% between them — realistic when the derived ETH/BTC×BTC/USD feed lags behind direct Chainlink ETH/USD during volatile markets — causes both to be excluded and WETH pricing reverts with `SimpleStrategy_PriceCountInvalid(0, 2)`. The same spread survives the 3-price median path because the median anchors on a middle value that doesn't shift symmetrically. This creates a non-monotonic DoS: losing 2-of-4 feeds is worse than losing 1-of-4. At the fork block ETH price of ~\$1962, the bricking threshold is a ~\$78 spread between the two survivors.

Recommendation

Use median as the benchmark for the 2-price case as well, or unify all input counts into a single filtering path via `_filterByDeviation`. Alternatively, increase WETH's `deviationBps` to tolerate realistic inter-feed divergence during volatile markets.

Resolution

Olympus: Resolved.

M-12 | Dead OHM/sUSDS Leg Can Stall OHM Pricing

Category	Severity	Location	Status
Oracle	● Medium	ConfigurePriceV1_2.sol	Partially Resolved

Description

`configurePriceV1_2()` hard-codes OHM to use a strict two-feed average over `OHM/WETH` and `OHM/sUSDS`. The second feed calls `UniswapV3Price.getTokenTWAP()`, which accepts the pool as long as `observe()` succeeds and the quote token can be priced. It does not reject a pool with zero in-range liquidity or a boundary tick. Consequently, a dead `OHM/sUSDS` feed is still treated as valid until the strategy sees its `0` result. That state already occurred on mainnet for pool `0x0858e2B0F9D75f7300B38D64482aC2C8DF06a755`. At block `24,862,000` on April 12, 2026, `slot0.tick = -887272`, `liquidity() = 0` and `observe([1800, 0])` still succeeded even though the newest initialized observation was about `4.31` days old. This happens because Uniswap V3 `observe()` counterfactually extends the last observation using the current tick and liquidity. With `OHM.decimals() = 9` and `sUSDS.decimals() = 18`, a TWAP tick of `-887272` makes `OracleLibrary.getQuoteAtTick()` return less than `1 wei` of `sUSDS` for `1 OHM`, so the `OHM/sUSDS` leg floors to `0`. This was not a theoretical edge case. The pool died in tx `0xeaa0a57e04a9cb086d55a59d0eb7a29912e1957677c498b0664b4afe01dc20ab` at block `24,831,090` on April 7, 2026. That swap sold about `0.3913027 OHM` for about `5.872525384784232 sUSDS` and moved the pool onto the minimum tick with zero active liquidity. The pool stayed there until tx `0x55c00482f91fa7bfc4036343e9e02c731149eaf7a4fdf68be81699a84df53e97` at block `24,877,959` on April 14, 2026, where a swap of about `138.09596400040314 sUSDS` for about `9.146450539 OHM` moved price back into range and restored active liquidity. This shows the configured pool is thin enough to go economically dead under normal trading. Because the rollout sets `ohmStrictMode = true`, the final OHM strategy reverts once only one non-zero price remains. On a replayed fork this exact case produces `OHM/sUSDS = 0`, `OHM/WETH > 0` and `SimpleStrategy.PriceCountInvalid(1, 2)`. Consequently, OHM configuration fails if the upgrade is executed during the dead window. This dependency remains live after the upgrade. Any fresh OHM recomputation, including `getPrice(OHM)` once the cache is stale, `getPrice(OHM, CURRENT)`, `cachePrice(OHM)`, `storeObservation(OHM)` and `storeObservations()`, will revert while the pool remains out of range. Since `Heart.beat()` begins by updating PRICE moving averages, the same condition can stall heartbeat execution before the rebase and other periodic tasks run.

Recommendation

Do not make this `OHM/sUSDS` pool a mandatory OHM oracle input. Consider removing the strict 2-of-2 dependence for OHM by setting `ohmStrictMode` to `false` until a third independent source is added.

Resolution

Olympus: Partially Resolved.

M-13 | Same-block Mixed Caches Can Pass Validation

Category	Severity	Location	Status
Oracle	● Medium	ChainlinkOracleCloneable.sol	Resolved

Description

ChainlinkOracleCloneable builds the quote from two independent PRICE.getPrice(..., Variant.LAST) reads and only rejects the result when the cached timestamps differ. That check does not prove both legs came from one coherent observation. PriceConfig2.cachePrice() is public and PRICE.cachePrice() writes cachedAt = block.timestamp. Therefore a block builder can include one transaction that refreshes the base asset early in a block and another that refreshes the quote asset later in the same block after the market moved. Both cache entries then share the same timestamp even though they represent different states. The adapter will return the mixed ratio and mark it as a valid Chainlink round by setting roundId, updatedAt and answeredInRound to that timestamp. Integrators that only check answer > 0, freshness and answeredInRound == roundId will accept an economically wrong price. For example, imagine the wrapper starts a block with coherent cached prices of OHM = 24.00 USD and USDS = 1.0000 USD, so the correct pair price is 24.0000 USDS per OHM. A public caller refreshes only the OHM leg early in the block. Later in the same block, after OHM has sold off and USDS has drifted slightly off peg, the new coherent live state is OHM = 23.10 USD and USDS = 0.9980 USD, so the correct pair price at that point is about 23.146292585170340681. Another public refresh updates only the USDS leg. The wrapper now reads OHM = 24.00 USD @ T and USDS = 0.9980 USD @ T, sees matching timestamps, and returns about 24.048096192384769539e18 as a fresh valid round. That value did not correspond to either coherent state in the block: the pair was 24.0000 before the move and about 23.1463 after it.

Recommendation

Do not treat timestamp equality as proof that the pair snapshot is coherent. The safest fix is to refresh both assets through one atomic pair snapshot and have the adapter read that shared record. If the protocol keeps per-asset caches, pair adapters should track a factory-controlled round counter or pair observation id that changes only when both legs are refreshed together. Public single-asset refreshes should not be able to create apparently valid pair rounds for assets used by the Chainlink wrapper.  (M-13/same-block-mixed-caches-chainlink-oracle.png)

Resolution

Olympus: Resolved.

L-01 | Oracle Validation Checks Base USD Price

Category	Severity	Location	Status
Logical Error	● Low	DeployOracles.sol	Resolved

Description

`DeployOracles.validateOraclePrice()` says it validates the deployed base/quote oracle against the configured `minPrice` and `maxPrice` bounds, but it never reads the oracle price or derives the pair price. After checking that the oracle exists and is enabled, it calls `IPRICEv2(_priceModule).getPrice(_baseToken)`, which is only the base token's USD price. That means the validation compares the wrong value whenever the quote token is not exactly one USD. A correct pair oracle can be rejected because the script is looking at the base token's USD price instead of the base/quote price. The reverse is also true: a wrong pair oracle can pass if the base token's USD price happens to sit inside the configured bounds. The scripted test harness reproduces both cases with a simple setup where the base token is worth `2e18` USD and the quote token is worth `4e18` USD. The true pair price is `0.5e18`, but `validateOraclePrice()` still passes for `1.5e18-2.5e18` bounds and reverts for the correct `0.4e18-0.6e18` bounds. This is not just a logging mistake as `BatchScriptV2` runs post-batch validation during simulation before it proposes or executes the batch, so the bug can block a correct deployment or give operators false confidence that a mispriced deployment was validated successfully.

Recommendation

Validate the deployed oracle's pair price instead of `PRICE.getPrice(baseToken)`. The safest fix is to query the oracle returned by the factory and compare its quote-denominated output against `minPrice` and `maxPrice`. If the script needs to stay generic and cannot read every oracle type directly, derive the expected pair price from the `PRICE` module by dividing the base token USD price by the quote token USD price and validate that value instead. The args file and docs should also make it explicit that the bounds are pair-price bounds, not base-token USD bounds.

Resolution

Olympus: Resolved.

L-02 | AveragelfDeviation Returns Min, Not First

Category	Severity	Location	Status
Logical Error	● Low	SimplePriceFeedStrategy.sol	Resolved

Description

`getAveragePriceIfDeviation()` says it returns the first non-zero price when no deviation is detected, but it sorts `nonZeroPrices` before that branch and then returns `nonZeroPrices[0]`. `QuickSort.sort()` mutates the array in place, so `nonZeroPrices[0]` is no longer the first configured feed. It is the minimum surviving price. That silently changes the selector's behavior from feed-priority semantics to min-price semantics whenever the inputs are close enough to avoid the deviation path. Any asset that relies on this selector to prefer a specific feed ordering will resolve to the lowest feed instead.

Recommendation

Cache the original first non-zero value before sorting and return that saved value on the no-deviation branch. `getMedianPriceIfDeviation()` already follows this pattern with `firstNonZeroPrice`.

Resolution

Olympus: Resolved.

L-03 | ERC4626 Oracle Rejects Valid Decimal Offsets

Category	Severity	Location	Status
Compatibility	● Low	ERC4626Price.sol	Resolved

Description

ERC4626Price reverts unless the share token and the underlying token report the same decimals. ERC-4626 does not require that invariant. A compliant vault can use different share decimals or a decimal offset and still return correct conversions through `convertToAssets()`. This hard check therefore rejects valid vaults that the adapter should be able to price. Once such a vault is configured, price resolution fails even though the vault itself may be well formed. Consequently, the listed asset becomes unreadable through PRICE, and any downstream oracle or dependent asset that relies on the same feed can fail as well. The current `sUSDS` deployment appears to use matching 18-decimal tokens, but the generic adapter remains incompatible with a valid class of ERC4626 vaults.

Recommendation

Consider removing the equality check and normalize across both scales instead. Query `convertToAssets(10 shareDecimals)` for one whole share, then divide by `10 underlyingDecimals` when translating the underlying USD price into a whole-share USD price.

Resolution

Olympus: Resolved.

L-04 | Feed Failure Blocks Asset Reconfiguration

Category	Severity	Location	Status
Configuration	● Low	OlympusPrice.v2.sol:581	Acknowledged

Description

`addAsset()` and `updateAsset()` both validate the final configuration with `_getCurrentPrice(asset_, true)` and then revert if `successAllFeeds` is false. This requires every configured feed in the final configured feed set to succeed during reconfiguration, even when the runtime strategy can still resolve a valid price from the remaining feeds. For example, WETH is configured with 4 feeds. If RedStone becomes stale while the other 3 feeds are still live, runtime WETH pricing continues to work because the strategy drops the stale leg. However, `updateAsset()` still reverts with `PRICE_PriceFeedCallFailed(WETH)` if the final updated feed set still includes that stale RedStone leg. This creates a reconfiguration failure mode during deployment and maintenance: a single stale or failing feed can cause validation to revert even though the asset remains priceable in normal runtime paths.

Recommendation

Do not require `successAllFeeds == true` for every `addAsset()` / `updateAsset()` path. Allow configuration to succeed when the final feed set and strategy can still resolve a valid aggregate price, even if excluded failed feeds would not.

Resolution

Olympus: Acknowledged.

L-05 | UniswapV3Price Accepts Counterfeit Pools

Category	Severity	Location	Status
Validation	● Low	UniswapV3Price.sol	Partially Resolved

Description

UniswapV3Price does not verify that the configured pool belongs to the canonical Uniswap V3 factory. `_checkPoolAndTokenParams()` only checks that the target responds to `slot0()`, `token0()`, and `token1()`, and that `lookupToken_` is one of the reported pool tokens. `getTokenTWAP()` then trusts `observe()` on the same address to supply the TWAP inputs. Consequently, any contract that mimics the Uniswap V3 pool surface can be accepted as a valid feed if it reports the expected token addresses and returns plausible oracle data. `PRICE.addAsset()` would still approve the configuration because its validation only asks the feed for one successful price. The rollout scripts make this easier to misconfigure because they read raw pool addresses from arguments and pass them straight through to `UniswapV3Price.UniswapV3Params` without checking the pool factory first. We checked the two currently configured Ethereum mainnet OHM pools on April 3, 2026, and both report the canonical Uniswap V3 factory `0x1F98431c8aD98523631AE4a59f267346ea31F984`. The contract itself still leaves that invariant unenforced, so a bad governance input or a compromised deployment script can turn the feed into an arbitrary price source while it still looks structurally valid.

Recommendation

Verify the pool against the canonical Uniswap V3 factory before accepting it as a feed. The safest fix is to read `factory()` from the configured pool, compare it to the expected factory for the deployment, and reject any mismatch. If governance always knows the intended token pair and fee tier, an even stronger fix is to derive the expected pool address from the canonical factory and compare it to the supplied address during configuration.

Resolution

Olympus: Partially Resolved.

L-06 | Pyth Two-feed Prices Ignore Derived Confidence

Category	Severity	Location	Status
Validation	● Low	PythPriceFeeds.sol	Resolved

Description

`getTwoFeedPriceDiv()` and `getTwoFeedPriceMul()` validate the confidence interval of each input feed independently through `_getFeedPrice()`, then combine the midpoint prices into a synthetic result. They never derive or validate the confidence interval of the synthetic price itself. That means the contract enforces the wrong uncertainty bound for composed prices. A two-leg cross-rate can have materially wider uncertainty than either leg on its own, especially during stressed markets. For example, if both legs are allowed to pass with roughly 1% relative uncertainty, the derived product or ratio can be close to 2% wide while still passing both per-feed checks. Pyth's own cross-rate guidance warns that confidence intervals are not derived automatically and must be handled manually when a protocol needs them. The current Olympus rollout does not use these two-feed Pyth helpers, so this is not an immediate production issue. It is still a real adapter bug because the contract advertises generic Pyth cross-rate helpers while enforcing only per-leg confidence caps, not a cap on the derived output that callers actually consume.

Recommendation

Derive and validate confidence on the synthetic result instead of only validating each leg independently. Consider computing a conservative confidence bound for the product or ratio and compare that derived bound against a dedicated output-level threshold.

Resolution

Olympus: Resolved.

L-07 | PriceConfigv2 Cannot Update V1.2 Target Floor

Category	Severity	Location	Status
Compatibility	● Low	PriceConfig.v2.sol	Acknowledged

Description

PriceConfigv2 explicitly accepts a PRICE module at version 1.2 or higher, but its permission set only covers the v2 asset, submodule, observation, and cache entrypoints. It never requests or exposes `changeMinimumTargetPrice()`, even though OlympusPricev1_2 still keeps `minimumTargetPrice` as live state and still uses it inside `getTargetPrice()`. That leaves the v1.2 compatibility floor stranded unless governance also keeps the deprecated OlympusPriceConfig policy deployed and active. The repository already shows that this is not guaranteed. The checked-in `src/scripts/deploy/savedDeployments/price_v1_2_deploy.json` deployment sequence includes OlympusPriceConfigV2, but does not include the legacy OlympusPriceConfig policy that is the only policy in this codebase which can call `changeMinimumTargetPrice()`. This matters because Operator still consumes `PRICE.getTargetPrice()` when updating RANGE prices and regeneration observations. If the protocol treats PriceConfigv2 as the replacement admin surface for a v1.2 module, the OHM target floor becomes fixed at its deployment-time seed. The protocol then loses the ability to adjust that floor as backing assumptions change.

Recommendation

Do not advertise PriceConfigv2 as a complete admin policy for OlympusPricev1_2 unless it can manage every live v1.2 control. Consider adding a v1.2-only wrapper for `changeMinimumTargetPrice()` and request that permission when the bound module is OlympusPricev1_2. If the intent is to keep using the legacy OlympusPriceConfig for that one control, make that dependency explicit in deployment scripts and rollout checks and fail deployment if the old policy is not present and active.

Resolution

Olympus: Acknowledged.

L-08 | Pyth Confidence Conversion Overflow Bricks Feed

Category	Severity	Location	Status
Math	● Low	PythPriceFeeds.sol:286-291	Resolved

Description

`PythPriceFeeds._getFeedPrice()` converts `maxConfidence` from output-decimal scale to Pyth scale via `SafeCast.encodeUInt64()`. The conversion computes `maxConfidence * 10^|expo| / 10^outputDecimals`. When `|expo| >= 22` and `maxConfidence` is a standard 1% threshold (`1e16` at 18 decimals), the result exceeds `type(uint64).max` (~1.8e19) and `SafeCast` reverts. This permanently bricks the feed for that asset – every `getOneFeedPrice` call reverts regardless of actual price data. The revert occurs during the admin-supplied `maxConfidence` conversion, before any comparison with the feed's actual confidence value. The code allows exponents down to -30 (`MAX_NEGATIVE_EXPONENT`), so configurations with `expo` in `[-22, -29]` combined with standard confidence thresholds are valid but permanently broken at runtime. The boundary is precise: `expo=-21` with `maxConfidence=1e16` produces `1e19` (fits), `expo=-22` produces `1e20` (overflows).

Recommendation

Validate at configuration time that `maxConfidence * 10^|expo| / 10^outputDecimals` fits in `uint64`. Alternatively, widen the accumulator to `uint256` for the comparison and clamp to `type(uint64).max` instead of reverting.

Resolution

Olympus: Resolved.

L-09 | Ignored Field Alters Cache

Category	Severity	Location	Status
Configuration	● Low	src/modules/PRICE/IPRICE.v2.sol#L267	Resolved

Description

IPRICEv2.UpdateAssetParams documents useMovingAverage as a field that is only read when updateStrategy=true. OlympusPrice.v2.updateAsset() initially follows that contract by resolving finalUseMA from the existing asset state whenever updateStrategy=false. However, the later cache-selection branch does not use finalUseMA. It reads raw params_.useMovingAverage instead. As a result, a caller can submit an update that claims not to touch strategy configuration, yet still change whether a single supplied observation becomes the cached Variant.LAST price/timestamp. Two otherwise identical updates can produce different cached state solely because of a field the interface says should be ignored.

Recommendation

Use the resolved final state (finalUseMA) instead of raw params_.useMovingAverage in the cache-selection logic. That makes the implementation match the documented API contract and ensures ignored fields truly have no effect unless their corresponding update flag is enabled.

Resolution

Olympus: Resolved.

L-10 | Non-MA Clones Have No Cache Refresh

Category	Severity	Location	Status
DoS	● Low	OlympusPrice.v2.sol:469-477	Resolved

Description

`heart.beat()` calls `storeObservations()` which only processes assets with `storeMovingAverage=true` (only OHM in the current config). WETH, USDS, and sUSDS have `storeMovingAverage=false`, so their `Variant.LAST` caches are set once at `addAsset` and never refreshed by the heartbeat. All three oracle clone types (Chainlink, Morpho, ERC7726) read `Variant.LAST` exclusively. After the initial cache ages past `maxAge`, Morpho and ERC7726 clones revert (DoS), while the Chainlink clone silently serves stale data indefinitely. On a mainnet fork after one heartbeat: OHM cache advances to `block.timestamp`, but USDS/sUSDS/WETH cache timestamps remain frozen at their initial values. After 3 heartbeats (~24h), the non-MA caches are over 81,000 seconds stale. The only refresh mechanism is `PriceConfig.cachePrice(asset)`, which is permissionless but requires an off-chain keeper. There is no on-chain, heartbeat-integrated cache path for non-MA assets.

Recommendation

Extend `storeObservations()` or `updateMovingAverage()` to also call `cachePrice()` for all approved assets, not just MA-tracking ones. Alternatively, document the keeper requirement explicitly and ensure a keeper is deployed before oracle clones go live.

Resolution

Olympus: Resolved.

L-11 | No Upper Bound On minimumTargetPrice

Category	Severity	Location	Status
Validation	● Low	OlympusPrice.v1_2.sol:147-150	Acknowledged

Description

`changeMinimumTargetPrice()` accepts any `uint256` with no upper bound. Setting it to an extreme value (e.g., `type(uint256).max`) makes `getTargetPrice()` always return that value, which propagates into `Operator._updateRangePrices()` and `_addObservation()`, distorting RBS wall/cushion pricing and regeneration logic. This is an admin-only function gated by the `permissioned` modifier, but there is no sanity guard preventing a governance misconfiguration from bricking the RBS system.

Recommendation

Cap `minimumTargetPrice` to a reasonable maximum (e.g., 2x the current moving average) or validate it against the current MA value before accepting.

Resolution

Olympus: Acknowledged.

L-12 | Derived Feed Threshold Exceeds Heartbeat

Category	Severity	Location	Status
Configuration	● Low	ConfigurePriceV1_2.sol:395	Resolved

Description

`_configureWeth()` sets `wethUpdateThreshold=3600` (1h) for all 4 WETH feeds including the derived ETH/BTC × BTC/USD feed. The Chainlink ETH/BTC aggregator uses a deviation-triggered heartbeat (updates on >1% moves, up to 24h max). On-chain analysis of the last 20 ETH/BTC rounds at block 24582001 shows 16/20 round gaps exceed the 1h threshold (gaps of 3612–3648s). Each gap creates a window where `ChainlinkPriceFeeds._getFeedPrice()` reverts with `Chainlink_FeedRoundStale`, silently dropping the derived feed. In quiet markets where ETH/BTC moves less, gaps can extend to hours. The `TwoFeedParams` struct already supports per-leg thresholds, but the config applies the same value to both.

Recommendation

Set `firstUpdateThreshold` to 86400 for the ETH/BTC leg to match its actual heartbeat. Keep `secondUpdateThreshold` at 3600 for BTC/USD.

Resolution

Olympus: Resolved.

L-13 | Wrong Oracle Source Passes Validation

Category	Severity	Location	Status
Validation	● Low	OlympusPrice.v2.sol:579-581	Resolved

Description

`addAsset()` and `updateAsset()` validate feed configurations by calling `_getCurrentPrice(asset_, true)` and checking that the aggregate price is non-zero. This proves the configured feeds are callable, but does not prove each feed is semantically correct for the asset. For example, wiring a BTC/USD Pyth feed ID into WETH pricing still passes `addAsset` because the Pyth contract returns a valid non-zero price for any registered feed ID. The deviation-excluding strategy then drops the ~\$80k BTC outlier and resolves WETH from the remaining 3 Chainlink feeds. The asset looks healthy even though one source is wrong. The mistake may only surface later if other feeds fail and the wrong feed survives filtering — particularly dangerous with the 2-price path (F-18) where the average benchmark replaces the median, potentially accepting a wrong-source price that would have been excluded with more feeds. At the fork block (24582001), this was confirmed: configuring WETH with BTC/USD instead of ETH/USD passes `addAsset` and returns a correct-looking price from the remaining feeds.

Recommendation

Strengthen configuration validation so each feed is checked against asset-specific expectations instead of only validating the final aggregate result. At minimum, add explicit source-level assertions in deployment/config scripts for assets that rely on multi-feed masking.

Resolution

Olympus: Resolved.

L-14 | CURRENT Reports Block Time For Degraded Inputs

Category	Severity	Location	Status
Oracle	● Low	OlympusPrice.v2.sol	Acknowledged

Description

`_getCurrentPrice()` always returns `uint48(block.timestamp)` as the quote timestamp. It does this even when the result depends on an older moving-average sample or on a reduced set of surviving feeds after one or more feed calls failed. This gives callers a freshness signal that is stronger than the underlying data. A consumer that trusts the returned timestamp can treat a degraded or partially historical value as if every input was observed at the current block. The module does compute `successAllFeeds`, but the public price getters discard that signal and still expose the current block time.

Recommendation

Do not use `block.timestamp` as the generic quote timestamp. Return the oldest freshness input used to build the value, or expose separate fields for cache time, last observation time and feed-success status so callers can enforce the freshness model they actually need.

Resolution

Olympus: Acknowledged.

L-15 | Runtime Writes Persist Degraded Fallback Prices

Category	Severity	Location	Status
Warning	● Low	OlympusPrice.v2.sol	Acknowledged

Description

`cachePrice()` and `_storeObservation()` write whatever `_getCurrentPrice()` or `_aggregate()` returns, even when one or more configured feeds failed. `_getFeedPrices()` does track this condition through `successAllFeeds`, but the runtime write functions ignore it and still update `LAST`, `obs` and `cumulativeObs`. The same module rejects partial feed failure during `addAsset()` and `updateAsset()`. At runtime, however, a multi-feed asset can silently degrade into a single surviving feed or a moving-average fallback if the strategy is configured in best-effort mode. The degraded result is then persisted as shared state and presented as the latest cached or observed price for later readers.

Recommendation

Make the runtime write functions enforce the same feed-success requirement as configuration changes. `cachePrice()` should revert when `successAllFeeds` is false. `_storeObservation()` should do the same before mutating observations or cache, unless the protocol explicitly opts an asset into degraded-write behavior through a separate configuration flag.

Resolution

Olympus: Acknowledged.

L-16 | Batch Can Activate A Factory From Another Kernel

Category	Severity	Location	Status
Validation	● Low	ConfigureOracles.sol	Resolved

Description

`ConfigureOracles` trusts the three factory addresses loaded from `src/scripts/env.json` and immediately schedules `ActivatePolicy` for each of them. It never checks that a factory was deployed against the same `Kernel` address that the batch is targeting. This matters because policy activation calls `configureDependencies()` and each factory resolves `PRICE` and `ROLES` through its own stored `kernel`. If the `env` file points at a factory from another deployment, the batch can still mark that contract active in the selected kernel and `validateOraclesConfigured()` will still pass. The factory will then keep reading modules from the other kernel. Later `enable()` or `createOracle()` calls can revert when they hit the wrong `ROLES` or `PRICE` module, or worse, create clones that read prices from the wrong deployment. `OracleProposal` makes this easier to miss because it uses a separate address registry for the same factories and does not cross-check that both stages point at the same contracts.

Recommendation

Before adding an activation to the batch, assert that each factory is bound to the target kernel. After activation, assert that the resolved `PRICE` module is the one associated with that same deployment. The batch and proposal should also share one address source, or explicitly compare both registries and revert on any mismatch.

Resolution

Olympus: Resolved.

L-17 | Correlated Chainlink Routes Overstate Quorum

Category	Severity	Location	Status
Warning	● Low	ConfigurePriceV1_2.sol	Acknowledged

Description

The v1.2 batch config gives `USDS` and `WETH` the same deviation strategy, `getAveragePriceExcludingDeviations(..., revertOnInsufficientCount=true)`, but the configured feed sets are not independent vote sets.

`USDS` is configured from:

- Chainlink `USDS / USD`
- Chainlink `DAI / USD`
- Pyth `USDS / USD`

`WETH` is configured from:

- Chainlink `ETH / USD`
- RedStone `ETH / USD`
- Pyth `ETH / USD`
- derived Chainlink `ETH / BTC x BTC / USD`

The strategy counts raw feed slots, not provider groups. Once there are 3 or more non-zero prices it sorts the array, chooses the median as the benchmark and filters by deviation from that benchmark. That means correlated Chainlink-family legs can control the benchmark even though they do not add a new independent trust domain.

Using the live rollout config and a direct fork execution of `SimplePriceFeedStrategy`:

- `getAveragePriceExcludingDeviations([1.05e18, 1.05e18, 1.00e18], 100 bps, strict)` returns `1.05e18`
- the two Chainlink-family `USDS` slots outvote the honest minority and the bad subset is accepted
- `getAveragePriceExcludingDeviations([2200e18, 1800e18, 1800e18, 2200e18], 200 bps, strict)` reverts with `SimpleStrategy_PriceCountInvalid(0, 2)`
- the two Chainlink-family `WETH` slots pull the median benchmark to `2000e18`, which excludes every quote and bricks strict pricing under a 2-vs-2 split

Because `sUSDS` is a single ERC4626 wrapper over `USDS`, the accepted `USDS` subset propagates directly into `sUSDS` pricing as well. The practical result is that the rollout overstates quorum quality. A single provider family malfunction can:

- make `USDS` accept a manipulated majority subset
- push `sUSDS` with it
- or make `WETH` revert even while two honest independent feeds still agree

Recommendation

Do not count correlated provider legs as separate quorum votes. The smallest safe change is to replace duplicate Chainlink-family slots with genuinely independent sources, or group feeds by provider and benchmark across groups instead of raw feed count.

For this rollout specifically:

- remove the Chainlink `DAI / USD` proxy from the `USDS` quorum unless it is treated as a fallback-only signal
- do not count `ETH / BTC x BTC / USD` as an independent `WETH` vote alongside direct Chainlink `ETH / USD`
- document an explicit minimum number of independent providers required for `USDS`, `sUSDS` and `WETH`

Resolution

Olympus: Acknowledged.

L-18 | Factory Re-enable Can Expose Stale Oracle Clones

Category	Severity	Location	Status
Warning	● Low	BaseOracleFactory.sol, ERC7726OracleFactory.sol	Resolved

Description

Factory-wide `enable( " " )` only calls `_enable(enableData_)`, flips `isEnabled` and emits `Enabled`. Neither `BaseOracleFactory` nor `ERC7726OracleFactory` overrides `_enable()` to refresh stale or mismatched `PRICE` cache entries before clones become live again. This matters because the clone wrappers trust the factory-level enabled flag and then read shared `PRICE Variant.LAST` state. For `BaseOracleFactory`, this affects both `MorphoOracleCloneable.price()` and `ChainlinkOracleCloneable.latestRoundData()`. For `ERC7726OracleFactory`, this affects `ERC7726OracleCloneable.getQuote()`. If the factory stays disabled longer than a clone's `maxAge`, or if one leg was refreshed before the pause, the clones come back enabled but still revert until someone separately repairs the cache.

Recommendation

Override `_enable()` in the factory layer and refresh each oracle pair that will become live again before `isEnabled` flips to `true`. If iterating every deployed oracle is too expensive, decode a bounded list of oracle addresses or token pairs from `enableData_` and recache them during the re-enable transaction. The key requirement is that factory-wide re-enable must not expose clones that still depend on stale or mismatched `LAST` timestamps.

Resolution

Olympus: Resolved.

L-19 | Disabling MA Leaves Stale Asset Metadata

Category	Severity	Location	Status
Unexpected Behavior	● Low	OlympusPrice.v2.sol	Resolved

Description

`_updateAssetMovingAverage()` clears `obs` and `cumulativeObs` when `storeMovingAverage_` is set to `false`, but it does not clear `movingAverageDuration`, `nextObsIndex`, `numObservations`, or `lastObservationTime`. Those fields are only reassigned inside the `storeMovingAverage_ == true` branch. As a result, an asset that previously stored a moving average can report stale moving-average metadata through `getAssetData()` after moving-average storage has been disabled. The pricing functions do key off `storeMovingAverage`, so this does not immediately corrupt on-chain price resolution. It does leave inconsistent lifecycle state for monitoring, migrations, and admin tooling that reads the raw asset struct.

Recommendation

Zero the remaining moving-average fields when storage is disabled. Clearing `movingAverageDuration`, `nextObsIndex`, `numObservations`, and `lastObservationTime` alongside `obs` and `cumulativeObs` keeps `getAssetData()` consistent with the asset's active configuration.

Resolution

Olympus: Resolved.

I-01 | ERC4626 Oracle Uses Idealized Share Value

Category	Severity	Location	Status
Warning	● Info	ERC4626Price.sol	Resolved

Description

ERC4626Price values one vault share with `convertToAssets(10 ** shareDecimals)`. ERC-4626 defines `convertToAssets()` as an idealized average-user conversion. It excludes fees and does not account for slippage or other execution conditions. Consequently, this adapter can overstate share value for vaults that charge withdrawal fees, gate redemptions, or otherwise make actual exits worse than the idealized rate. That matters because PRICE treats this value as the asset's USD price. Any later integration that relies on this adapter for collateral valuation or as an input to another oracle can inherit an optimistic price. The current `sUSDS` deployment does not appear to charge fees, but the submodule is generic and the issue can be triggered in future ERC4626 listings.

Recommendation

Merely informative. Ensure this behaviour is documented.

Resolution

Olympus: Resolved.

I-02 | ConfigureOracles Uses Missing Env Keys

Category	Severity	Location	Status
Configuration	● Info	ConfigureOracles.sol	Resolved

Description

ConfigureOracles loads the kernel and all three factory addresses through `_envAddressNotZero()` using the keys `olympus.policies.ChainlinkOracleFactory`, `olympus.policies.MorphoOracleFactory` and `olympus.policies.ERC7726OracleFactory`. That is not compatible with the checked-in address registry. The repository's `src/scripts/env.json` does not define any of those policy keys on mainnet, sepolia, or goerli, so `_envAddressNotZero()` turns the lookup into an immediate revert instead of building the batch. The same rollout stack has a second naming mismatch in `BatchScriptV2._validateHeartBeat()`, which requires `olympus.policies.OlympusPriceConfigV2` even though the registry only defines `OlympusPriceConfig`. Consequently, this script cannot be simulated, signed, or proposed from the repository state as checked in.

Recommendation

Make the rollout scripts consume the canonical keys that already exist in `src/scripts/env.json`, or add the missing keys to the registry and keep the naming consistent across `ConfigureOracles`, `DeployOracles`, `ConfigurePriceV1_2`, and `BatchScriptV2`. The safest fix is to add a small preflight check that asserts every required env key exists before simulation begins, with an error that prints the missing key names directly.

Resolution

Olympus: Resolved.

I-03 | OracleProposal Missing Factory Activation

Category	Severity	Location	Status
Configuration	● Info	OracleProposal.sol:164-180	Resolved

Description

OracleProposal._build() calls IEnabler.enable() and createOracle() on the oracle factories but never pushes Kernel.executeAction(Actions.ActivatePolicy) for them. Without Kernel activation, configureDependencies() never runs, so ROLES and PRICE references are address(0). The enable() call reverts because it queries ROLES.hasRole() on a zero address. The proposal depends on a separate prior batch (ConfigureOracles) to activate the factories first, but this dependency is not documented or enforced. The proposal's _validate() also does not check Kernel.isPolicyActive().

Recommendation

Either include ActivatePolicy actions in the proposal so it is self-contained, or add a pre-condition check that verifies the factories are already Kernel-active before attempting to enable them.

Resolution

Olympus: Resolved.

I-04 | DeactivatePolicy Does Not Stop Clones

Category	Severity	Location	Status
Warning	● Info	BaseOracleFactory.sol:360-364	Resolved

Description

Oracle clones check liveness via `factory.isOracleEnabled()`, which includes the factory's `PolicyEnabler.isEnabled` flag. `Kernel.executeAction(DeactivatePolicy, factory)` revokes the factory's kernel permissions (PRICE module access, role grants), but it does not call `factory.disable()`. Because `isEnabled` remains true, all existing oracle clones continue to serve cached price reads after deactivation. The actual kill switch for oracle consumption is `factory.disable("")`, which flips `isEnabled = false` and causes every clone's `_checkEnabled()` to revert. This matters for emergency response: if a compromised or faulty price feed is detected and the team deactivates the factory via Kernel governance, downstream Chainlink/Morpho/ERC7726 oracle clones will keep returning prices to external consumers. The Chainlink clone is especially affected because it lacks per-read `maxAge` enforcement (unlike Morpho and ERC7726 clones), so it will serve indefinitely stale cached data.

Recommendation

Document in the operational runbook that `DeactivatePolicy` removes module permissions but does not halt oracle clone reads. The emergency stop action is `factory.disable("")`. Consider adding a `Kernel`-triggered callback or overriding `onDeactivate()` to automatically disable the factory when the policy is deactivated.

Resolution

Olympus: Resolved.

I-05 | Double storeObservation Corrupts MA

Category	Severity	Location	Status
Logical Error	● Info	OlympusPrice.v2.sol:394-446	Resolved

Description

`_storeObservation()` has no same-block guard. A permissioned caller can call `storeObservation()` twice in the same block, advancing the circular buffer index twice and replacing two historical observations with the same spot price. The `lastObservationTime` is set to `block.timestamp` on both calls, so the second call overwrites the observation stored by the first. With a 3-observation buffer, a double-call produces a 17% MA deviation from the expected value. The corrupted MA feeds into `getTargetPrice()` and downstream RBS range calculations via `Operator._updateRangePrices()`. The function documents that it does not enforce a minimum frequency between observations, pushing scheduling responsibility to the caller. However, the lack of even a same-block guard means a single transaction can corrupt the MA without detection.

Recommendation

Add `if (asset.lastObservationTime == uint48(block.timestamp)) revert` to prevent same-block double writes. The caller (Heart) already enforces per-epoch timing, but the module itself should not rely on external scheduling for correctness.

Resolution

Olympus: Resolved.

I-06 | Static Price Bounds Block Deployment

Category	Severity	Location	Status
Configuration	● Info	src/scripts/ops/batches/ConfigurePriceV1_2.sol:51-58	Resolved

Description

`ConfigurePriceV1_2` hardcodes price validation bounds as constants (ETH: 1500–2100, OHM: 17–22, USDS: 0.99–1.01, sUSDS: 1.06–1.10). These are enforced in `validatePricesAreSane()` (line 602), which runs as post-batch validation via `_setPostBatchValidateSelector`. If market prices move outside these ranges before the governance batch executes, the entire PRICE v1.2 upgrade reverts — even when the PRICE configuration itself is correct. The script already contains a `TODO` comment at line 48 acknowledging the bounds need adjustment at deployment time, but the values are compile-time constants rather than configurable parameters. A volatile market day or delayed governance execution window could force a recompile and redeployment of the batch script.

Recommendation

Make the price validation bounds configurable at execution time by loading them from the args JSON file (alongside other deployment parameters) instead of hardcoding them as constants. Alternatively, derive bounds dynamically from live PRICE module state with a configurable tolerance percentage.

Resolution

Olympus: Resolved.

I-07 | Variant.LAST Semantic Changed By Refactor

Category	Severity	Location	Status
Logical Error	● Info	src/modules/PRICE/OlympusPrice.v2.so l:438-445	Resolved

Description

The `_storeObservation()` refactor changed what `Variant.LAST` returns for assets with `useMovingAverage=true`. Previously, the cache stored the raw feed-aggregated price (MA excluded). Now it stores the MA-inclusive aggregated price computed via `_aggregate(_getInclusivePrices(feedPrices))`. Additionally, the cache uses the post-update MA (`cumulativeObs` is updated at line 430 before the cache is computed at line 439), creating a forward bias where the new observation feeds into its own cached price. With `Feed1=$10`, `Feed2=$12`, `MA=$10`: old `LAST=$11` (raw avg), new `LAST=$10.833` (MA-inclusive with post-update `MA=$10.5`). Oracle clones reading `Variant.LAST` silently receive a different value than before the refactor. Current deployment is unaffected because OHM has `useMovingAverage=false`, but any future asset configured with `useMovingAverage=true` will exhibit this semantic change.

Recommendation

If the old behavior (raw feed price in cache) was intended, change `_storeObservation` to always cache `obsPrice` (the MA-excluded aggregate) regardless of `useMovingAverage`. If the new behavior is intentional, document the semantic change clearly so oracle clone consumers understand what `Variant.LAST` now represents.

Resolution

Olympus: Resolved.

I-08 | updateAsset Locked By Stale MA

Category	Severity	Location	Status
DoS	● Info	OlympusPrice.v2.sol:881	Resolved

Description

updateAsset() concludes with `_getCurrentPrice(asset_, true)` for validation. The `true` argument forces MA inclusion, which checks `lastObservationTime + observationFrequency <= block.timestamp` and reverts with `PRICE_MovingAverageStale` if the heartbeat was missed. This blocks ALL governance reconfiguration for assets with `useMovingAverage=true` — even feed-only updates unrelated to the MA. The lockout occurs during exactly the degraded state when reconfiguration is most needed. A workaround exists: passing `updateStrategy=true, useMovingAverage=false` disables the MA before validation, but this changes pricing behavior as a side effect. Current deployment is unaffected (OHM has `useMovingAverage=false`).

Recommendation

Pass `includeMovingAverage_=false` in the validation call at line 881, or add a separate validation path that skips the MA staleness check during asset reconfiguration.

Resolution

Olympus: Resolved.

I-09 | Split PRICE Upgrade Leaves OHM Unapproved

Category	Severity	Location	Status
Upgradeability	● Info	OlympusPrice.v1_2.sol	Resolved

Description

OlympusPricev1_2 is presented as a backward-compatible PRICEv1 replacement, but its constructor only stores OHM, observationFrequency and minimumTargetPrice. It does not register OHM as a v2 asset or migrate any of the old module's live pricing state.

That becomes dangerous during an actual module upgrade because Kernel._upgradeModule() swaps the module immediately and then rebinds dependent policies to the new address. If governance upgrades PRICE in one transaction and configures OHM in a later transaction, every legacy OHM lookup on the new module reverts with PRICE_AssetNotApproved(OHM) in the gap between those steps.

Because of this issue, a mis-sequenced rollout can take the live price module offline until a repair transaction installs submodules and adds OHM.

For example:

- Transaction 1 on Monday: the DAO upgrades PRICE from v1.1 to v1.2
- Transaction 2 on Tuesday: the DAO plans to install submodules and configure OHM, WETH, USDS, and sUSDS
- Between Monday and Tuesday: Heart.beat() runs, or a keeper tries to call it
- Heart asks the new PRICE module for OHM pricing
- The new PRICE module reverts because OHM is not an approved asset yet
- The beat fails, so rebases and periodic maintenance stall until the follow-up configuration transaction lands

Recommendation

Do not rely on a multi-transaction rollout for this upgrade. The module upgrade, policy activation, submodule installation and OHM asset configuration should execute atomically in one governance batch. If that guarantee is not acceptable, add migration or initialization logic so the upgraded module registers OHM and restores the minimum state needed for legacy PRICEv1 reads before dependent policies are rebound to it.

Resolution

Olympus: Resolved.

I-10 | Clone Assumes ERC20 Decimals For All Assets

Category	Severity	Location	Status
Warning	● Info	ERC7726OracleCloneable.sol	Resolved

Description

`_getQuoteInternal` calls `IERC20(base_).decimals()` and `IERC20(quote_).decimals()` for every quote. This makes the adapter depend on both assets implementing the ERC20 metadata interface. That assumption is narrower than the ERC-7726 interface it exposes. ERC-7726 allows special non-ERC20 addresses such as the ETH sentinel and ISO-4217 codes. Those values do not implement `decimals()`, so the clone will revert even if the PRICE module can provide a cached value for them. Therefore the contract is only generic across ERC20-like contract addresses.

Recommendation

Either constrain the clone explicitly to ERC20-like assets in its documentation and factory validation, or add explicit handling for non-ERC20 ERC-7726 assets and source their decimals from a registry instead of assuming `IERC20.decimals()`.

Resolution

Olympus: Resolved.

I-11 | Health Checks Ignore Disabled Oracle State

Category	Severity	Location	Status
Warning	● Info	ERC7726OracleCloneable.sol	Resolved

Description

getQuote and getQuotes call _checkEnabled() before reading cached prices, but isStale and timestamp skip that check. A disabled clone can therefore report a fresh timestamp and false from isStale even though every quote attempt will revert with ERC7726Oracle_NotEnabled. This can mislead integrators that treat isStale or timestamp as a preflight signal for quote availability. The cache may be healthy while the oracle is still unusable.

Recommendation

Apply _checkEnabled() in isStale and timestamp so all read methods reflect the same enabled-state contract. If these helpers are meant to report cache status only, document that clearly and avoid presenting them as a quote-readiness check.

Resolution

Olympus: Resolved.

I-12 | ERC4626 Adapter Assumes Trusted Share Rate

Category	Severity	Location	Status
Oracle	● Info	ERC4626Price.sol	Resolved

Description

ERC4626Price prices a vault share by reading raw `convertToAssets()` output and multiplying that conversion rate by the underlying asset's USD price. The adapter does not add any smoothing, sanity checks, or manipulation resistance of its own. Consequently, it is only safe when the wrapped vault's share-to-asset conversion is already trusted on oracle timescales. The configured vault is `sUSDS` and direct `USDS` donation does not change `sUSDS.totalAssets()`, `convertToAssets()`, or downstream OHM pricing. `sUSDS` still grows over time, but that is the intended yield behavior the adapter is meant to report. This adapter becomes risky if it is later pointed at a vault whose share rate can be moved directly through donations, harvest timing, rebasing side effects, or other externally steerable accounting changes. In that case the reported vault price would inherit that vault-specific behavior.

Recommendation

Document that `ERC4626Price` is only suitable for vaults whose share conversion is already trusted and restrict deployments accordingly. If broader ERC4626 support is required, add vault-specific bounds or use a share-rate source that applies its own smoothing or sanity checks instead of trusting raw `convertToAssets()` output.

Resolution

Olympus: Resolved.

I-13 | ConfigureOracles Cannot Be Safely Retried

Category	Severity	Location	Status
Warning	● Info	ConfigureOracles.sol	Resolved

Description

`configureOracles` always queues three `ActivatePolicy` actions. It does not check whether one or more factories are already active before building the batch. This makes the script non-idempotent. `Kernel` reverts when `ActivatePolicy` is called for an already active policy and `BatchScriptV2` aborts the whole batch on the first failed call. Consequently, if an operator needs to rerun the batch after a partial rollout, the retry can fail before it reaches the remaining inactive factories. A recovery run can therefore brick itself instead of finishing the missing activations.

Recommendation

Query `Kernel.isPolicyActive()` for each factory before adding it to the batch. Only enqueue inactive factories and log when a factory is skipped because it is already active.

Resolution

Olympus: Resolved.

I-14 | Unchecked Casts Can Rewrite PRICE Batch Args

Category	Severity	Location	Status
Warning	● Info	ConfigurePriceV1_2.sol	Resolved

Description

ConfigurePriceV1_2 reads numeric batch arguments as `uint256` and then narrows them with plain Solidity casts before encoding them into PRICE components. For example, `_configureUSDS()` casts `usdsDeviationBps` to `uint16` and `usdsUpdateThreshold` to `uint48`. The same pattern is repeated for `_ohmObservationWindow` in `configurePriceV1_2()` and for the WETH settings in `_configureWeth()`. These casts do not validate bounds. They truncate modulo the destination width. Consequently, a malformed or mistyped JSON value can be turned into a different but still-valid oracle parameter instead of reverting. For example, `65537` basis points becomes `1,248 + 3600` becomes `3600` and `232 + 1800` becomes `1800`. The target submodules will decode those rewritten values normally, so the batch can install configuration that does not match the operator's input.

Recommendation

Reject out-of-range arguments before any narrowing cast. The simplest fix is to compare each raw `uint256` against `type(uint16).max`, `type(uint32).max`, or `type(uint48).max` as appropriate and revert on overflow, then cast only after that check passes.

Resolution

Olympus: Resolved.

Remediation Findings & Resolutions

ID	Title	Category	Severity	Status
M-01	PRICE Accepts MA Strategies That Later Fail	Validation	● Medium	Resolved
M-02	Queued MA Updates Execute With Stale Timestamps	Unexpected Behavior	● Medium	Acknowledged
L-01	Morpho Oracles Keep Stale Decimal Scales	Unexpected Behavior	● Low	Resolved
L-02	Price Cache Accepts Stale MA Snapshots	Unexpected Behavior	● Low	Acknowledged
L-03	Feed Failures Block Price Reconfiguration	DoS	● Low	Acknowledged
L-04	Pyth Confidence Bypass	Configuration	● Low	Resolved
L-05	Unsafe Fallback Pricing	Oracle	● Low	Acknowledged
L-06	Queued Update Not Implementation-bound	Configuration	● Low	Acknowledged
I-01	OHM Filter Can Halt Heart After Feed Outage	Warning	● Info	Acknowledged
I-02	Synthetic MA Check Can Miss Price Outage	Warning	● Info	Resolved
I-03	OHM Rollout Materials Use Stale Settings	Documentation	● Info	Acknowledged
I-04	WETH Feed Expectation Is Stale	Warning	● Info	Acknowledged
I-05	Submodule Upgrades Lack Ordering	Unexpected Behavior	● Info	Acknowledged

M-01 | PRICE Accepts MA Strategies That Later Fail

Category	Severity	Location	Status
Validation	● Medium	OlympusPrice.v2.sol	Resolved

Description

`_validateAssetConfiguration` only counts the final number of configured price sources as `feedCount + 1` when `useMovingAverage` is true. It does not prove that the configured strategy can aggregate every runtime input count that the module will use. This matters because the same strategy is used for two different calculations. `getPrice(..., CURRENT)` can include the moving average, but `storeObservation()` intentionally excludes it when writing the next observation. `addAsset()` validates only the MA-inclusive calculation. Therefore a configuration with two feeds plus a moving average can pass with `SimplePriceFeedStrategy.getMedianPrice`, which requires at least three inputs, then every later `storeObservation()` reverts because it sends only the two raw feed prices to the same median strategy. `updateAsset()` has the inverse problem. It validates the final configuration with `_getCurrentPrice(asset_, false)`, so a strategy-only update can enable `useMovingAverage` without ever testing the MA-inclusive current price. A single-feed asset with stored observations can be updated to use the median strategy. The update succeeds because the MA-excluded validation returns the single raw feed directly, then `getPrice(asset, CURRENT)` reverts because the median strategy receives only two inputs.

Recommendation

Validate every calculation that can be used after the configuration is stored. For assets that store a moving average and use it in the strategy, check both the MA-inclusive current price and the MA-excluded observation value before accepting `addAsset()` or `updateAsset()`. If the design intentionally allows one strategy to handle different input counts, add an explicit strategy capability check or separate strategies for current-price reads and observation writes.

Resolution

Olympus: Resolved.

M-02 | Queued MA Updates Execute With Stale Timestamps

Category	Severity	Location	Status
Unexpected Behavior	● Medium	OlympusPrice.v2.sol	Acknowledged

Description

`updateAsset()` writes a replacement moving-average state from `params_.lastObservationTime`, then validates the resulting asset with `_getCurrentPrice(asset_, false)`. Passing `false` skips the moving-average freshness check even when the asset still has `useMovingAverage = true`. This becomes unsafe through `PriceConfigv2.queueUpdateAsset()`. The queued payload contains a fixed `lastObservationTime`, but `TimelockQueue` makes the action executable only after the timelock delay. With the configured 1 day minimum delay and an 8 hour PRICE observation frequency, a timestamp that was fresh when queued is already stale when execution is first allowed. Any caller can then execute the queued action and store stale moving-average metadata. Fresh `CURRENT` reads for the asset immediately revert with `PRICE_MovingAverageStale` until a permissioned observation update repairs the state.

Recommendation

Reject queued moving-average updates that would be stale at execution, or revalidate the MA-inclusive current price after applying a moving-average update when `useMovingAverage` remains enabled. A safer design is to avoid queueing absolute observation timestamps for MA state and instead refresh the observation at execution time from live feeds.

Resolution

Olympus: Acknowledged.

L-01 | Morpho Oracles Keep Stale Decimal Scales

Category	Severity	Location	Status
Unexpected Behavior	● Low	MorphoOracleFactory.sol	Resolved

Description

MorphoOracleFactory._encodeOracleData snapshots collateral and loan decimals from the current priceCache and turns them into an immutable scaleFactor. The deployed clone then keeps using that old scale while reading prices from the factory's current priceCache. This is unsafe when the decimal source changes for an existing oracle. BaseOracleFactory.setPriceCache can rotate the cache policy, and PriceCache.setNonContractAssetMetadata can update decimals for non-contract assets. After either change, the pair cache can be refreshed with the new metadata, but the Morpho clone still scales the returned ratio with the old decimals. For example, if a loan asset was created with 18 decimals and later has 6 decimals in the active cache, the correct Morpho scale is smaller by 1e12. The old clone would keep returning values scaled with the larger factor. If that oracle is used by a Morpho market, collateral or debt can be materially mispriced. This requires a privileged cache rotation or metadata update, but those are supported maintenance operations rather than key-compromise assumptions.

Recommendation

Treat decimals for assets used by enabled Morpho oracles as immutable. Before rotating priceCache, validate that every affected enabled oracle still resolves the same collateral and loan decimals. If any pair differs, disable and recreate the oracle instead of keeping it live. For non-contract metadata, prevent decimal changes while active Morpho oracles reference the asset, or require the operator to disable and replace all affected oracles as part of the metadata update. If the oracle must survive cache rotations, store the deployment-time collateral and loan decimals and make price() revert when the active cache reports different values.

Resolution

Olympus: Resolved.

L-02 | Price Cache Accepts Stale MA Snapshots

Category	Severity	Location	Status
Unexpected Behavior	● Low	PriceCache.sol	Acknowledged

Description

PriceCache checks cached pair freshness only by comparing `cachedPrice.updatedAt` against `maxAge`. When `_cachePrice()` stores a non-unit asset, it reads `PRICE.getPrice(asset, Variant.CURRENT)`. For an asset configured with `useMovingAverage = true`, that current price can include the stored moving average. After the moving-average deadline passes, direct `PRICE.getPrice(asset, Variant.CURRENT)` reverts because the moving average is stale. The already-cached pair snapshot can still be treated as fresh by PriceCache until `updatedAt + maxAge` expires, because the cache does not know the moving-average deadline embedded in the stored value. For example, assume an asset has one live feed at `\$10`, a stored moving average of `\$20`, and `useMovingAverage = true`. While the moving average is still fresh, `PRICE.getPrice(asset, CURRENT)` returns `\$15` and a public cache call stores that `\$15` pair snapshot. When the moving average expires one hour later, direct PRICE current reads revert, but an adapter configured with a two-hour `maxAge` can still read the cached `\$15` value for another hour. Consequently, Morpho, ERC-7726, or Chainlink-style adapters can keep serving an MA-inclusive cached value after the underlying moving-average component is no longer valid for fresh PRICE reads.

Recommendation

Make cache freshness account for the data used inside the cached price. One option is to expose from PRICE whether a current value included a moving average and the timestamp at which that component expires. PriceCache should then mark the pair stale when either the pair `maxAge` expires or any embedded moving-average component is stale.

Resolution

Olympus: Acknowledged.

L-03 | Feed Failures Block Price Reconfiguration

Category	Severity	Location	Status
DoS	● Low	OlympusPrice.v2.sol	Acknowledged

Description

`addAsset()` validates a new asset by reading the current price and then requiring every configured feed to have returned a nonzero value. `updateAsset()` repeats the same check after applying a configuration change. This happens even when the configured strategy can compute a valid price from the remaining feeds. Consequently, an authorized operator can be blocked from adding or updating an asset while one retained feed is temporarily failing. This can happen even if the final feed set and strategy would still produce a valid nonzero aggregate price at runtime. For example, a strict average strategy can ignore one zero feed and resolve from two live feeds, but `addAsset()` and `updateAsset()` still revert with `PRICE_PriceFeedCallFailed` because `successAllFeeds` is false.

Recommendation

Do not require `successAllFeeds` during configuration when the strategy has already returned a valid aggregate. Let the configured strategy decide whether enough valid feeds remain, and expose degraded-feed status separately for operator visibility.

Resolution

Olympus: Acknowledged.

L-04 | Pyth Confidence Bypass

Category	Severity	Location	Status
Configuration	● Low	src/modules/PRICE/submodules/feeds/ PythPriceFeeds.sol:317-318	Resolved

Description

PythPriceFeeds lets admins configure `maxConfidence` in output-decimal scale, then converts it to Pyth's raw exponent scale before validating `priceData.conf`. If the converted value exceeds `uint64.max`, the code clamps the limit to `uint64.max`. Because Pyth's raw confidence field is also `uint64`, the clamped limit accepts every possible raw confidence value. This silently disables the intended confidence guard for any feed/configuration that crosses the clamp boundary. Current deployment values for normal `expo = -8` USD feeds do not appear to clamp, so the issue is a configuration-validation gap rather than an active deployment exploit.

Recommendation

Reject configurations whose converted confidence limit exceeds `uint64.max`, or require an explicit unbounded-confidence sentinel.

Resolution

Olympus: Resolved.

L-05 | Unsafe Fallback Pricing

Category	Severity	Location	Status
Oracle	● Low	src/modules/PRICE/submodules/feeds/UniswapV3Price.sol:260	Acknowledged

Description

OlympusPricev2 catches failed feed calls and leaves those feed prices as zero. The configured strategy then decides whether the remaining prices are enough. In best-effort mode, SimplePriceFeedStrategy can return a single surviving source. This becomes dangerous if a manipulable source such as UniswapV3Price.getTokenPrice() is included in the feed set. That selector reads the current pool tick from slot0(), so it is a same-block spot price. If other feeds fail or are filtered out, a multi-feed configuration can collapse to one manipulable spot source. Current deployment args use strict mode and the OHM Uniswap feeds use TWAP selectors, so this is an unsafe allowed configuration rather than a current-config exploit.

Recommendation

Disallow spot selectors in production price feeds, and require strict mode or an explicit minimum valid-source count for assets with economic consumers.

Resolution

Olympus: Acknowledged.

L-06 | Queued Update Not Implementation-bound

Category	Severity	Location	Status
Configuration	● Low	src/policies/price/PriceConfig.v2.sol:603	Acknowledged

Description

`PriceConfigv2.queueUpdateAsset()` queues `UpdateAssetParams`, whose feed and strategy components identify submodules by `SubKeycode`. The queued payload does not bind those keycodes to the implementation addresses installed at queue time. If a referenced feed or strategy submodule is upgraded before the queued asset update executes, the same reviewed payload executes against the new implementation. This is inconsistent with `queueExecOnSubmodule()`, which stores the current submodule implementation in the queued payload and rejects execution if that implementation changes before execution.

Recommendation

Include the expected implementation address for every feed and strategy submodule used by the final asset configuration in the queued `updateAsset` payload, and reject execution if any referenced submodule implementation changed since queue time.

Resolution

Olympus: Acknowledged.

I-01 | OHM Filter Can Halt Heart After Feed Outage

Category	Severity	Location	Status
Warning	● Info	ConfigurePriceV1_2.sol	Acknowledged

Description

The OHM price configuration uses `getAveragePriceExcludingDeviations` with a 200 bps tolerance and strict mode enabled. It currently reads from three sources: OHM/WETH, OHM/sUSDS and Chainlink OHM/ETH * ETH/USD. If one source is unavailable or returns zero, the strategy is left with exactly two non-zero prices. In the two-price case, `SimplePriceFeedStrategy` uses the average of the two prices as the deviation benchmark. With a 2% tolerance, two surviving prices that differ by a little more than 4% are each more than 2% away from their midpoint. The strategy then excludes both prices and reverts with `SimpleStrategy_PriceCountInvalid(0, 2)`. This leaves a narrow heartbeat liveness dependency in the OHM configuration. A single unavailable OHM source, combined with unusually large divergence between the two remaining sources, can make `PRICE.updateMovingAverage()` revert. `Heart.beat()` calls that function before triggering the rebase and before executing periodic tasks, so this would stop heartbeat execution until the failed source recovers or the two surviving prices move back inside the allowed range. Recent mainnet history suggests this condition is unlikely. During the known OHM/sUSDS zero-quote window from block 24,831,090 through 24,877,959, sampled every 300 blocks, none of the 155 comparable samples crossed the two-price threshold. The largest OHM/WETH-vs-Chainlink divergence in that window was 183 bps. A broader daily-scale sample from block 24,000,000 through 25,021,851 also found no threshold crossings, with a maximum divergence of 190 bps across 142 comparable samples. Since the strict two-price case requires a little over 400 bps divergence relative to the lower surviving price, the checked history supports treating this as a hardening and monitoring item.

Recommendation

Merely informative. Consider changing the strict two-price fallback only if the protocol wants the heartbeat to tolerate both a source outage and unusually high inter-source divergence.

Resolution

Olympus: Acknowledged.

I-02 | Synthetic MA Check Can Miss Price Outage

Category	Severity	Location	Status
Warning	● Info	OlympusPrice.v2.sol	Resolved

Description

`_validateMovingAverageStrategy` checks MA-backed configurations with a value that normal current-price reads do not use. It aggregates the raw feed prices into `obsPrice`, then appends `obsPrice` as a synthetic moving average. `getPrice(asset, CURRENT)` appends the stored moving average from `asset.cumulativeObs / asset.numObservations` instead. This distinction matters for assets configured with `useMovingAverage = true` and a strict deviation-filtered strategy. Validation can accept the raw feeds plus the synthetic value, while the same raw feeds plus the stored moving average would revert. In that case the configuration is accepted, but immediate `CURRENT` reads remain unavailable until a fresh observation is stored or the asset is changed. The current v1.2 batch does not configure any asset this way. USDS, sUSDS and WETH are added with `storeMovingAverage = false` and `useMovingAverage = false`. OHM stores a moving average, but it is added with `useMovingAverage = false`, so the moving average is not supplied to the strategy. Therefore this is not a current deployment breakage. It is a hardening issue for future or manual MA-backed configurations that expect validation to prove immediate current-price availability.

Recommendation

If successful `addAsset` or `updateAsset` calls should guarantee immediate `CURRENT` price availability, validate the strategy with the stored moving average that runtime reads will use. If the synthetic value is deliberately intended as a recovery check when the stored moving average is stale or out of consensus, document that narrower guarantee and require a fresh observation before any consumer depends on the asset's `CURRENT` price.

Resolution

Olympus: Resolved.

I-03 | OHM Rollout Materials Use Stale Settings

Category	Severity	Location	Status
Documentation	● Info	price.md	Acknowledged

Description

The PRICE rollout documentation still describes OHM as using `getAveragePrice()` in strict mode and lists 1,800-second Uniswap observation windows. The executable batch now configures OHM with `getAveragePriceExcludingDeviations`, `ohmDeviationBps = 200`, strict mode and 1500-second windows for both OHM/WETH and OHM/sUSDS. The args file also keeps `ohmObservationWindow = 1800` even though the script reads the split `ohmWethObservationWindow` and `ohmSusdsObservationWindow` values instead. This does not change deployed behavior, but it makes the rollout materials disagree with the configuration that reviewers and signers are expected to approve.

Recommendation

Update the OHM documentation and batch args so they describe the same configuration that the script executes. Remove the unused `ohmObservationWindow` key, or wire it deliberately if a single shared window is intended.

Resolution

Olympus: Acknowledged.

I-04 | WETH Feed Expectation Is Stale

Category	Severity	Location	Status
Warning	● Info	ConfigurePriceV1_2.json	Acknowledged

Description

The v1.2 batch args still set `wethExpectedPrice` to `1800e18` with a `1667` bps tolerance. `ConfigurePriceV1_2` applies that same expectation envelope to every WETH feed before the asset is added and `PriceConfigv2` rejects any feed value outside the resulting bound. That bound is roughly 1500 to 2100 USD. A live Chainlink ETH/USD read on May 4, 2026 returned `2348.96366170`, which is above the configured upper bound. Therefore the batch can fail during WETH feed validation before the PRICE upgrade is configured.

Recommendation

Refresh the WETH expected price and tolerance immediately before proposing or executing the batch. Prefer deriving the expectation from live feed reads in a preflight step, then writing the exact values into the signed args.

Resolution

Olympus: Acknowledged.

I-05 | Submodule Upgrades Lack Ordering

Category	Severity	Location	Status
Unexpected Behavior	● Info	src/policies/price/PriceConfig.v2.sol:654	Acknowledged

Description

`PriceConfigv2.queueUpgradeSubmodule()` queues only the replacement submodule address. Multiple pending upgrades for the same subkeycode are not ordered or bound to the implementation that was current when each action was queued. As a result, an older queued upgrade can execute after a newer upgrade and replace the newer implementation during the older action's execution window. This is not an authorization bypass because both upgrades were timelocked and emergency can cancel stale actions, but it can confuse operational review and rollback expectations.

Recommendation

Bind queued submodule upgrades to the expected current implementation, or use a per-subkeycode nonce so stale queued upgrades revert.

Resolution

Olympus: Acknowledged.

Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Guardian to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Guardian’s position is that each company and individual are responsible for their own due diligence and continuous security. Guardian’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Guardian is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract’s safety and security. Therefore, Guardian does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

About Guardian

Founded in 2022 by DeFi experts, Guardian is a leading audit firm in the DeFi smart contract space. With every audit report, Guardian upholds best-in-class security while achieving our mission to relentlessly secure DeFi.

To learn more, visit <https://guardianaudits.com>

To view our audit portfolio, visit <https://github.com/guardianaudits>

To book an audit, message <https://t.me/guardianaudits>