



ZEALYNX

Web3 Security and AI Agent & MCP Audits

Mystic Finance Smart Contract Audit

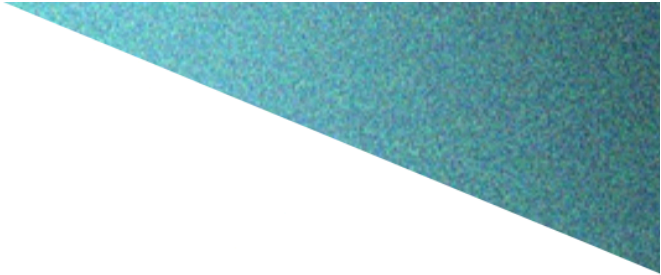
28 October 2026

Zealynx

contact@zealynx.io

Carlos Vendrell (Bloqarl)

Lead Security Auditor



Contents

1. About Zealynx

2. Disclaimer

3. Overview

3.1 Project Summary

3.2 About Mystic Finance

3.3 Audit Scope

4. Audit Methodology

5. Severity Classification

6. Executive Summary

6.1 The Key Findings

6.2 Architectural Security Observations

6.3 Security Strengths Observed

6.4 The Three Areas You Asked About

7. Audit Findings

7.1 Medium Severity Findings

7.2 Low Severity Findings

8. Informational Notes

1. About Zealynx

Zealynx, founded in January 2024 by Carlos (Bloqarl), specializes in smart contract audits, and development. Our services include comprehensive smart contract audits, application security audits, such as pentesting, and AI Audits. We are trusted by clients such as Badger DAO, Ample protocol, Lido, Inverter, Matchain, and Golden Grid.

Our team believes in transparency and actively contributes to the community by creating educational content. This includes challenges for Foundry and Rust, security tips for Solidity developers, and fuzzing materials like Foundry and Echidna/Medusa. We also publish articles on topics such as preparing for an audit and battle testing smart contracts on our website [Zealynx.io](https://zealynx.io) and our blogs.

Zealynx has achieved public recognition, including winning 1st prize in the Uniswap Hook Hackathon and a Top 5 position in the Beanstalk Audit public contest. Our ongoing commitment is to provide value through our expertise and innovative security solutions for smart contracts.

2. Disclaimer

A security review of a smart contract can't guarantee there are no vulnerabilities. It's a limited effort in terms of time, resources, and expertise to find as many vulnerabilities as possible. We can not assure 100% security post-review or guarantee the discovery of any issues with your smart contracts.

3. Overview

3.1 Project Summary

Octarine RFQ gives holders of tokenised real-world assets an exit before the issuer's redemption date. A taker sells an RWA token to a market maker at the oracle NAV, less a discount the maker bids per day until the asset redeems. Both sides sign, and each fill is bounded by the maker's signed maximum outlay and the taker's signed minimum proceeds.

3.2 About Mystic Finance

Mystic Finance builds liquidity infrastructure for tokenised RWAs. Octarine is its facility and RFQ layer, planned for Plume and Ethereum.

Redemption timing comes from a per-asset schedule: ROLLING, a fixed delay, or CYCLICAL, a calendar date the keeper maintains. Prices come from three adapter types:

- a Chainlink-style feed adapter;
- a call-based adapter that reads an issuer's own rate;
- a signer-pushed setter oracle with step and interval bounds.

Launch assets include:

- Nest assets on Plume;
- BENJI, Pareto's AA tranche, ULTRA and OUSG;
- ACRED and Pareto AA on monthly schedules.

The admin role and the oracle owners are team multisigs. The schedule keeper and the setter-oracle signers are operational bot keys.

3.3 Audit Scope

The RFQ core and the four shared oracle contracts, at commit 157c3ff4c701f2974e58c9847519741dcaade464: 466 nSLOC by your count, or 608 code lines. The fixes were reviewed on main at commit 5f7beb4. The escrow, Dutch-auction, vault, redemption-vault, instant-pool and Rincewind contracts were out of scope, except where in-scope code relies on their behaviour.

```
src/octarine-rfq/OctarineRFQ.sol
src/shared/oracles/OctarineOracleBase.sol
src/shared/oracles/OctarineChainlinkOracle.sol
src/shared/oracles/OctarineCallOracle.sol
src/shared/oracles/OctarineThirdPartySetterOracle.sol
```

4. Audit Methodology

Approach

During our security assessments, we uphold a rigorous approach to maintain high-quality standards. Our methodology encompasses thorough **Fuzz** and **Invariant Testing**, and **meticulous manual security review**.

Throughout the smart contract audit process, we prioritize the following aspects to uphold excellence:

1. **Code Quality:** We diligently evaluate the overall quality of the code, aiming to identify any potential vulnerabilities or weaknesses.

2. **Best Practices:** Our assessments emphasize adherence to established best practices, ensuring that the smart contract follows industry-accepted guidelines and standards.

3. **Documentation and Comments:** We meticulously review code documentation and comments to ensure they accurately reflect the underlying logic and expected behavior of the contract.

5. Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

6. Executive Summary

This was a **two-day Founder Security Sprint** against a frozen commit, followed by a review of the fixes. Findings were filed as issues as they were found, so remediation could begin before this report existed.

How findings were proven. Every finding is backed by a runnable Foundry test that asserts the harm in balances or prices, not just that a path exists. Each test was then subjected to a falsification step: the cited defect was corrected and the unchanged test re-run, so each proof depends on the defect rather than on the harness. Where a finding depends on how a real asset behaves, the test runs against that asset's deployed contracts on a pinned mainnet fork.

How the fixes were checked. Every recommended fix was:

- applied and run against the full test suite;
- reviewed adversarially for bypasses and regressions;
- checked alongside the other fixes for how they combine.

That review sharpened several recommendations before delivery.

Your own fixes were then reviewed commit by commit, in two rounds. Each round re-ran every proof of concept and our follow-up tests against them. The final round covered `main` at `5f7beb4`, where your suite passes 729 tests with none failing.

Scope of the effort. Two days cannot be exhaustive. Effort went first to the three areas you asked about: the new oracle types, CYCLICAL schedules, and fills around a NAV update. The signed-settlement path then received its own conservation testing.

The engagement surfaced **15 findings: 6 Medium and 9 Low**. No Critical or High issues were found. The settlement core held up well. Its signed bounds, partial-fill accounting and replay protection survived every attack we built against them. The weaknesses sit where the contract meets the outside world: in how prices are published and validated, and in how calendar redemption dates are modelled.

After the fix review:

- 9 findings are fixed;
- 2 are mitigated by documented operating rules;
- 4 are acknowledged as intended behaviour, now documented.

Two Mediums, M-03 and M-04, were regraded to Low, which leaves **4 Medium and 11 Low**.

6.1 The Key Findings

- **A reset hands the price to whichever signer pushes first.** After the oracle owner calls `invalidate()`, which your runbook reserves for very large NAV moves, the next push from any signer key is accepted with no step or interval limit. The honest keeper's correction is then held to the bounds that first push skipped, so it is refused. A single compromised bot key can fix the price the RFQ settles at for as long as it takes to notice and remove it. *Fixed: after a reset, no push is accepted until the owner seeds the price.*
- **Calendar redemptions are modelled as fixed-length cycles.** A CYCLICAL schedule rolls its anchor forward by a fixed `cycleSeconds` when the keeper misses an update, but months are 28–31 days long. Near a month-end, a missed update leaves fills priced on the wrong side of the redemption date. In the worked case, a sale 12 hours before redemption is charged 29.5 days of discount. *Mitigated: the keeper bound now admits month-length changes, and a documented keeper runbook keeps the anchor on the calendar.*
- **The schedule has no notion of the issuer's request cutoff.** A maker that buys after the filing deadline is paid a full cycle later than the contract assumes. Under the operation you described, the keeper cannot move the anchor to the next payout date in one call. A late or missed move overpays by about the bid rate $\times$ the cycle, for example 3% at 10 bps/day on a monthly asset. *Fixed: a per-asset redemption lead now prices fills after the filing deadline to the next payout.*
- **The call-based adapter cannot refuse a price its own source has flagged invalid.** On the real Pareto contract, a borrower default leaves `priceAA()` at its pre-default value while `defaulted()` turns true, and fills keep settling at the pre-default price. On ULTRA, an issuer unit change in May 2025 would have re-priced a fixed adapter $\times 100$. *Fixed: an adapter can now pin a guard call such as `defaulted()`, and a per-asset halt stops every RFQ route that prices the asset.*

6.2 Architectural Security Observations

Four patterns account for most of the findings.

Protections that can be configured or reset away. Every bound on the setter oracle exists for the case where a signer key misbehaves, and each has a path that switches it off:

- the constructor accepts a zero interval or a step of 100%;
- `invalidate()` frees the next push from every bound;
- with only two signers, one compromised key stalls recovery for a full signer timelock.

Feed health is decided by the adapter's word. The RFQ trusts the `lastUpdated` an adapter reports, and reads every failure as "feed down" and quietly falls back to the reference-asset route. That is how a call-based source with no clock is always fresh, and how a gas-starved read becomes indistinguishable from an outage. It is also how a defaulted source keeps pricing: nothing the source publishes about its own health reaches the RFQ.

Signed bounds derived from the inputs they protect against. `maxMakerAmount` and `minReceivedAmount` are set from a quote of the contract's own pricing. When that pricing is wrong, the bounds encode the error instead of catching it. Calendar drift, the request cutoff and a frozen NAV all get through this way. You told us makers derive their caps from their own independent NAV, which helps against a bad price but not against a wrong horizon.

Redemption semantics the contract cannot see. Correct CYCLICAL pricing depends on keeper actions at precise moments:

- publishing each calendar date;
- moving the anchor before each request cutoff.

The contract neither knows those moments nor enforces them, and its keeper shift bound blocks one of the moves the runbook needs.

6.3 Security Strengths Observed

- **The assumptions section is written adversarially.** It names what is accepted, why, and where the acceptance stops. Several findings came from testing those stated rationales, which was possible only because they were written down.
- **Two-signature consent is sound under splitting.** The request and the bid are signed separately, and the bounds are pro-rated per fill, with the taker's floor rounded up. We verified across a stateful fuzz of about 154,000 calls that neither side's signed bound leaks over any split of an order into partial fills, whoever submits them.
- **Measured balance deltas.** The taker leg measures what actually arrived, so fee-on-transfer and rebasing taker tokens settle on delivered amounts, clamped so a rebase cannot inflate the maker's bill.
- **Decimals are folded once, in the adapters, and the scaling is exact.** We checked it against a rational model for every combination of 0–24 token and source decimals, and through real fills in both the direct and triangulated routes.
- **The test suite anticipates its own optimizer.** Your test base documents that `via_ir` can reuse a cached `block.timestamp`, and anchors every warp to an absolute time. Most teams find this hazard only after a false-green test.
- **Source reads are gas-capped and bounded in returndata,** so a hostile source cannot exhaust the caller.

6.4 The Three Areas You Asked About

1. The new oracle types: are the bounds and validation sufficient?

At the frozen commit, not yet, in eight specific places. Each had a small fix.

- The setter oracle's bounds can be configured off: [L-04].
- A reset hands the next price to the first signer: [M-01].
- One compromised signer can take every update slot, so honest signers cannot correct it: [L-07].
- A call-based source with no update time is always fresh. This applies to Pareto's AA price; ULTRA can be bounded through its epoch timestamp: [M-04].
- No adapter can act on a source's own validity signals, such as a default flag or a unit change: [M-06].
- The submitter's gas limit can push a costly direct read onto the reference route: [L-02].
- The Chainlink-style adapter cannot price Chronicle feeds: [L-03].
- The same adapter cannot read the nanosecond Stork feeds that Morpho uses for Nest assets: [L-09].

After the fix review:

- [L-02] is mitigated by a documented gas ceiling.
- [M-04] is kept by design and regraded to Low.
- The other six are fixed.

What holds, verified by execution:

- **Adapters on real sources.** Configured correctly, each adapter reads the real BENJI, Pareto, ULTRA and OUSG contracts exactly on a mainnet fork.
- **Feed validation.** The Chainlink adapter's round, answer and timestamp validation is sound. A negative or future answer fails closed.
- **The setter oracle's controls.** Its step bound, interval, signer timelock and immediate removal hold across their whole admitted range.

Signers with equal, independent power remain a residual risk after the [L-07] fix: two compromised keys can still walk the price. The robust answer is a quorum or median.

2. Cyclical schedule mode: have you covered the edge cases?

The arithmetic is correct. We verified it against an independent model of your interface, across the full admitted domain, at 50,000 runs per property. That covered:

- the countdown and the wrap;
- the keeper shift bound on every mode transition;
- the RWA→RWA net horizon.

The model has two material gaps, both about calendars:

- fixed-length cycles against calendar months: [M-02], now mitigated by a documented keeper runbook;
- no notion of a request cutoff: [M-05], now fixed with a per-asset redemption lead.

3. What happens when fills land just before or after a NAV update?

The fill logic handles the boundary correctly: fills settle at the right price up to the staleness edge, and revert one second later. The exposure is in how a NAV move gets published.

- **Large moves are walked in steps.** A move larger than one step is walked in steps a `minUpdateInterval` apart, and every intermediate step is a live, fresh price. In a 20% mark-down walked at 5% per day, every live quote overpays by 18.75% at the first step. The makers' caps do not bind on a mark-down.
 - Halt the asset for the walk with `setHalted(asset, true)`, added in the fix for [M-06]. Unrouting one pair is not enough: a registered direct pair, such as the asset against another RWA, keeps pricing on its own oracle.
- **Very large moves use a reset.** The `invalidate()` path for those was the reset race in [M-01], now fixed: the owner seeds the post-reset price.
- **Frozen issuer rates.** A rate that stops updating is served as current on a call-based source with no clock: [M-04].
- **NAV strike on the redemption date.** When the strike coincides with the redemption date, the horizon jumps at the same moment as the price. Signed bounds and expiries are the only protection, as your Assumption 3 states.

Summary of Findings:

Vulnerability	Severity	Status
[M-01] After invalidate(), the first push from any signer sets an unbounded price that honest signers cannot correct	Medium	Fixed
[M-02] CYCLICAL schedules roll by a fixed cycle, so after a missed keeper update fills near each month-end are charged the wrong side of the redemption date	Medium	Mitigated
[M-05] The schedule has no notion of a redemption request cutoff, so fills after the filing deadline are charged for a payout the maker can no longer reach	Medium	Fixed
[M-06] OctarineCallOracle cannot refuse a price its own source flags as invalid: a defaulted Pareto tranche keeps settling at its pre-default price, and ULTRA's x100 unit change re-prices the route	Medium	Fixed
[M-03] Revoking a delegated order signer only pauses it: registering the same key again revives every unexpired order it signed	Low	Acknowledged
[M-04] CallOracle without a timestamp call reports every read as fresh, so the RFQ's mandatory staleness bound never applies to issuer-set rates	Low	Acknowledged
[L-02] The submitter's gas limit can starve a healthy direct CallOracle read, so the fill is priced on the triangulated route	Low	Mitigated
[L-03] The Chainlink adapter rejects every Chronicle answer (startedAt is always 0), so Chronicle-priced assets cannot be filled or are silently repriced	Low	Fixed
[L-04] Setter oracle accepts guard values that switch off its bounds, so one signer key can move the price without limit in a single block	Low	Fixed
[L-05] The maker leg's measured receipt is not clamped, so a token that credits balances mid-transfer lets a fill settle below the taker's signed floor	Low	Fixed

Vulnerability	Severity	Status
[L-06] A relayed fill publishes the taker's signed request, so a competing maker can front-run the winning bid and fill at the taker's floor	Low	Acknowledged
[L-07] One compromised setter signer can take every update slot, so honest signers cannot correct a walked price and extra signers add no protection	Low	Fixed
[L-08] A delegated signer's cancelSalt under its own address succeeds and emits SaltCancelled, but the orders it signed stay fillable	Low	Acknowledged
[L-09] The Chainlink adapter rejects every answer from the Stork feeds Morpho uses for Nest assets on Plume, because Stork reports timestamps in nanoseconds	Low	Fixed
[L-10] The filler of an open quote cannot set a minimum, so a schedule or price move between its simulation and settlement is charged to it in full	Low	Fixed

Statuses reflect the fix review of main at 5f7beb4 (the same code as remediations-1 at 6ef4d9b):

- **Fixed:** removed in code.
- **Mitigated:** reduced by a documented operating or deployment rule that the code does not enforce.
- **Acknowledged:** the behaviour is kept by design and is now documented.

M-03 and M-04 were graded Medium at delivery and regraded to Low in the fix review. They keep their IDs so they match the fix commits. There is no L-01: it was regraded to Medium during the sprint and renumbered M-04.

7. Audit Findings

7.1 Medium Severity Findings

[M-01] After invalidate(), the first push from any signer sets an unbounded price that honest signers cannot correct

Affected files

src/shared/oracles/OctarineThirdPartySetterOracle.sol#L74-L92

Description

pushPrice bounds a push only when a price is already stored (L79-L83):

```
uint256 prev = price;
if (prev != 0) {
    uint64 nextAt = updatedAt + minUpdateInterval;
    if (block.timestamp < nextAt) revert TooSoon(nextAt);
    if (_deviates(newPrice, prev, maxChangeBps)) revert
    ChangeTooLarge(prev, newPrice);
}
```

invalidate() deletes price (L89-L92). So after a reset, and before the very first push, **any** signer may set any price with no step or interval limit, as the L74 natspec says. Every later push is then bounded against whatever that first push set.

That inverts the purpose of the bounds ("so a signer key cannot walk the price far in one sitting", L11-L15) at the one moment the owner uses them. invalidate() is the reset path for a genuine move larger than maxChangeBps, such as a NAV mark-down or a distribution, or for a bad price. Whichever signer pushes first after it chooses the new anchor without limit. The honest correction that follows is held to the same bounds that rogue push skipped, so it is refused until it has been walked back maxChangeBps at a time, one minUpdateInterval apart, or until the owner resets again and the race repeats.

Vulnerable Scenario:

1. The oracle runs a conservative configuration: 5% per push, one push per hour, two signers.
2. The issuer marks NAV down from 1.00 to 0.92. That exceeds 5%, so the owner calls invalidate() to let the keeper publish 0.92.
3. A second signer key, compromised, pushes 1.03 first. It is accepted, unbounded.

4. The keeper's 0.92 reverts `ChangeTooLarge`, in that block and an hour later. The oracle keeps serving 1.03.
5. The attacker fills a maker's open quote at 1.03 for RWA the issuer values at 0.92.

Impact

A single signer key that pushes first after a reset fixes the price the RFQ settles at. The price it chooses is not bounded by the oracle, and it cannot be corrected by the other signers within the oracle's own bounds. Live orders and open quotes are exposed up to their signed caps until the owner identifies and removes the key and resets again. In the PoC the maker pays 10,269.10 USDC for 10,000 RWA that the issuer's NAV values at 9,172.40 USDC, +12%, and the keeper's correct price is refused twice. Most of that gap is the 8% markdown the rogue key keeps from being published; the maker's own 3% headroom over the pre-markdown price is the rest. The loss therefore grows with the size of the NAV move the reset was meant to publish.

This is the path your runbook takes for very large NAV moves: the owner invalidates, and a signer pushes the new price. The first push after `invalidate()` sets the anchor, so on every such move the honest keeper is racing every other signer key for that push.

Proof of Concept

A runnable Foundry proof of concept for this finding is in the corresponding issue on the project tracker.

Recommendations

Do not let a single signer set the price that the bounds are then measured from. There are two ways to do that, and they differ in who holds price power.

Option A: a second signer confirms the reset price. Owner keeps no price power.

With no live price, which means the first price after deployment or after `invalidate()`, a push only proposes the value. It goes live when a different signer pushes the same value. Signers keep sole price authority, and the owner keeps only signer management behind `signerTimeLock`, as today.

```
@@ -23,4 +23,9 @@ contract OctarineThirdPartySetterOracle is
OctarineOracleBase, Ownable2Step {
    uint64 public updatedAt;

+    /// @notice With no live price (first push, or after
+    {invalidate}), a push is only proposed; it goes live
```

```

+    ///          when a different signer pushes the same value.
+    uint256 public pendingPrice;
+    address public pendingSigner;
+
+    mapping(address => bool) public isSigner;
+    mapping(address => uint64) public signerSubmittedAt;
@@ -28,4 +33,5 @@ contract OctarineThirdPartySetterOracle is
OctarineOracleBase, Ownable2Step {
+    event PricePushed(address indexed signer, uint256 price);
+    event PriceInvalidated();
+    event PriceProposed(address indexed signer, uint256 price);
+    event SignerSubmitted(address indexed signer, uint64
validAt);
+    event SignerRevoked(address indexed signer);
@@ -77,5 +83,14 @@ contract OctarineThirdPartySetterOracle is
OctarineOracleBase, Ownable2Step {
+    if (newPrice == 0) revert ZeroPrice();
+    uint256 prev = price;
-    if (prev != 0) {
+    if (prev == 0) {
+        if (pendingPrice != newPrice || pendingSigner ==
address(0) || pendingSigner == msg.sender) {
+            pendingPrice = newPrice;
+            pendingSigner = msg.sender;
+            emit PriceProposed(msg.sender, newPrice);
+            return;
+        }
+        delete pendingPrice;
+        delete pendingSigner;
+    } else {
+        uint64 nextAt = updatedAt + minUpdateInterval;
+        if (block.timestamp < nextAt) revert TooSoon(nextAt);
@@ -89,4 +104,6 @@ contract OctarineThirdPartySetterOracle is
OctarineOracleBase, Ownable2Step {
+    function invalidate() external onlyOwner {
+        delete price;
+        delete pendingPrice;
+        delete pendingSigner;

```

```

        emit PriceInvalidated();
    }
    @@ -115,4 +132,8 @@ contract OctarineThirdPartySetterOracle is
    OctarineOracleBase, Ownable2Step {
        function removeSigner(address signer) external onlyOwner {
            isSigner[signer] = false;
+           if (pendingSigner == signer) {
+               delete pendingPrice;
+               delete pendingSigner;
+           }
            delete signerSubmittedAt[signer];
            emit SignerSet(signer, false);
        }
    }

```

Costs:

- **Deploy every setter oracle with at least three signers.** The honest keepers must publish identical values. They read the same issuer NAV, so that is natural, but it must be exact.
- **A compromised signer can stall a reset, but no longer set the price.** By overwriting the proposal it keeps fills on the asset reverting until the owner removes it. Setting the price would take two compromised keys.
- **Recovery needs a third signer.** With only two signers, removing the compromised one leaves a single signer, and one signer can never confirm its own proposal. The asset then stays unpriced until a replacement signer clears `signerTimelock`. In our test (not included here), 24 hourly honest pushes after the removal all stayed proposals, and pricing returned only when a new signer cleared the timelock, 259,200 s later. With three signers, the two honest keys confirm in the next block.
- In the in-scope suites, 15 tests fail. Every one bootstraps with a single signer push, or asserts the single-push behaviour this removes, and needs its fixture updated.

Option B: the owner publishes the reset price. Simpler, but the owner gains price power.

An owner-only `seedPrice` sets the first price and the price after `invalidate()`, and `pushPrice` refuses when no price is live:

```

@@ -38,4 +38,5 @@ contract OctarineThirdPartySetterOracle is
OctarineOracleBase, Ownable2Step {
    error NoPendingSigner();
    error TimelockNotElapsed(uint64 validAt);
}

```

```

+     error NotSeeded();

    constructor(
@@ -77,9 +78,17 @@ contract OctarineThirdPartySetterOracle is
OctarineOracleBase, Ownable2Step {
        if (newPrice == 0) revert ZeroPrice();
        uint256 prev = price;
-        if (prev != 0) {
-            uint64 nextAt = updatedAt + minUpdateInterval;
-            if (block.timestamp < nextAt) revert TooSoon(nextAt);
-            if (_deviates(newPrice, prev, maxChangeBps)) revert
ChangeTooLarge(prev, newPrice);
-        }
+        if (prev == 0) revert NotSeeded();
+        uint64 nextAt = updatedAt + minUpdateInterval;
+        if (block.timestamp < nextAt) revert TooSoon(nextAt);
+        if (_deviates(newPrice, prev, maxChangeBps)) revert
ChangeTooLarge(prev, newPrice);
+        price = newPrice;
+        updatedAt = uint64(block.timestamp);
+        emit PricePushed(msg.sender, newPrice);
+    }
+
+    /// @notice Set the price outright: the first price, or the
anchor after {invalidate}. Signers resume
+    ///         bounded pushes from it.
+    function seedPrice(uint256 newPrice) external onlyOwner {
+        if (newPrice == 0) revert ZeroPrice();
+        price = newPrice;
+        updatedAt = uint64(block.timestamp);

```

Costs:

- **Today the owner cannot set a price at all.** To get a price in, it must add a signer and wait out signerTimelock, and that delay is what limits a stolen owner key. With seedPrice, a stolen owner key can call invalidate() and seedPrice() in one block and set any price instantly. If you choose this option, restrict seedPrice to when no price is live, and

consider putting it behind the same `signerTimeLock`. The asset is then unpriced for that delay.

- A large NAV move waits on the owner rather than a keeper, which is slower if the owner is a multisig.
- In the in-scope suites, 17 tests fail. Every one fails with `NotSeeded()` at a signer's bootstrap push, and two of them assert the exemption this removes.

Either option also closes the same exemption at deployment, where the first push is otherwise unbounded for every signer. With either applied, `testFuzz_resetHandsAnchorToFirstSigner` passes 5,000 runs and `test_firstPushAfterResetIsStickyAgainstHonestCorrection` fails, as described above.

Remediation

Fixed in `beefe0b` (Option B). After `invalidate()`, and at deployment, every push reverts `NotSeeded` until the owner calls `seedPrice`, so no signer can set the price the bounds are measured from. Replaying the scenario, the rogue key's 1.03 is refused and the maker pays the fair 9,172.40 USDC instead of 10,269.10.

The cost of Option B remains. `seedPrice` is not behind `signerTimeLock`, so a stolen owner key can call `invalidate()` and `seedPrice()` in one block and set any price. The setter oracle's owner is therefore trusted with the price; the README does not yet say so. A reset also leaves each signer's interval running, so a signer whose interval is open can move a fresh seed by one step.

[M-02] CYCLICAL schedules roll by a fixed cycle, so after a missed keeper update fills near each month-end are charged the wrong side of the redemption date

Affected files

src/octarine-rfq/OctarineRFQ.sol#L110-L124

Description

CYCLICAL is documented as a calendar schedule: "redemption lands on a calendar date" (IOctarineRFQ.sol L41-L42). The README gives "Fixed calendar date (e.g. the 30th)", and in practice that means month-ends. Once the anchor passes, `_secondsFor` rolls it forward by a fixed `cycleSeconds` (L113-L118):

```
uint256 cycle = s.cycleSeconds;
uint256 anchor = s.nextRedemptionAt;
if (block.timestamp >= anchor) {
    uint256 periods = ((block.timestamp - anchor) / cycle) + 1;
    anchor += periods * cycle;
}
```

Calendar months are 28 to 31 days long, so no fixed `cycleSeconds` lands on them. The interface treats the roll as a safe fallback: "a missed bot update degrades to a stale-but-sane schedule rather than a broken one" (IOctarineRFQ.sol L200-L202). It is not sane near a redemption date. The horizon is a sawtooth that drops to one second and jumps back to a full cycle at the anchor, so an anchor one day early turns a one-day date error into a horizon error of almost a full cycle:

- **Rolled anchor early (cycle shorter than the month):** between the rolled anchor and the real date, the horizon has already wrapped to about `cycleSeconds` while redemption is hours away. The taker is charged a month of discount for a wait of hours.
- **Rolled anchor late (cycle longer than the month):** between the real date and the rolled anchor, the horizon is near zero while the real wait is a whole new cycle. The maker pays almost no discount for a month's wait.

Neither side's signed bound helps. `minReceivedAmount` and `maxMakerAmount` are set from `quoteRFQOrder`, which returns the same wrong horizon.

Vulnerable Scenario:

1. An asset redeems on the last day of each month. The keeper sets `nextRedemptionAt` = 31 Oct 2026 with `cycleSeconds` = 30 days and then misses its next update.
2. The roll gives 30 Nov (correct), then 30 Dec, one day before the real 31 Dec.
3. At 30 Dec 12:00 UTC redemption is 12 hours away, but `secondsToRedemption` returns 29.5 days.
4. A taker sells 10,000 RWA at 10.00 bps/day, signing a floor 0.5% under the contract's quote. The fill charges 29.5 days of discount.

Impact

Inside each mismatch window, every fill prices the wrong side of the redemption date. With a 30-day cycle, after one missed keeper update the windows in the next five months are 30–31 Dec, 29–31 Jan and 30–31 Mar. The loss per fill is up to `makerBpsPerDay` × `cycle`: the taker overpays the discount, or the maker underpays it. In the PoC a taker selling 10,000 RWA at 10.00 bps/day receives 9,705 USDC instead of the 9,995 USDC the same rate gives over the real 12-hour horizon, a loss of 290 USDC (2.9%). The windows fall on predictable dates, so a maker can bid into them deliberately.

Proof of Concept

A runnable Foundry proof of concept for this finding is in the corresponding issue on the project tracker.

Recommendations

Do not extrapolate a calendar schedule by a fixed cycle. Once the anchor passes, refuse to price until the keeper publishes the next redemption date, and let the keeper publish it after the anchor has passed:

```
@@ -110,12 +110,10 @@ contract OctarineRFQ is IOctarineRFQ,
    AccessControl, ReentrancyGuard, Pausable,
        function _secondsFor(RedemptionSchedule memory s) internal
    view returns (uint32) {
        if (s.mode == ScheduleMode.ROLLING) return
    s.rollingSeconds;

-    uint256 cycle = s.cycleSeconds;
+    // A calendar schedule cannot be extrapolated by a fixed
+    cycle: once the anchor passes, refuse
+    // to price until the keeper publishes the next
+    redemption date.
```

```

        uint256 anchor = s.nextRedemptionAt;
-       if (block.timestamp >= anchor) {
-           uint256 periods = ((block.timestamp - anchor) /
cycle) + 1;
-           anchor += periods * cycle;
-       }
+       if (block.timestamp >= anchor) revert InvalidSchedule();

        // Exact to the second: the last hours before redemption
price as hours, not as a whole day.
        uint256 remaining = anchor - block.timestamp;
@@ -131,9 +129,11 @@ contract OctarineRFQ is IOctarineRFQ,
AccessControl, ReentrancyGuard, Pausable,
    _validateSchedule(s);

        RedemptionSchedule memory prev = _schedules[asset];
+       // An expired CYCLICAL schedule has no current horizon:
measure no shift, but keep the cycle-length check.
+       bool expired = prev.mode == ScheduleMode.CYCLICAL &&
block.timestamp >= prev.nextRedemptionAt;
        if (!isAdmin && prev.configured &&
maxScheduleShiftSeconds != 0) {
-           uint256 before_ = _secondsFor(prev);
            uint256 after_ = _secondsFor(s);
+           uint256 before_ = expired ? after_ :
_secondsFor(prev);
            uint256 shift = before_ > after_ ? before_ - after_ :
after_ - before_;

            uint256 pAmp = prev.mode == ScheduleMode.ROLLING ?
before_ : prev.cycleSeconds;
@@ -165,7 +165,7 @@ contract OctarineRFQ is IOctarineRFQ,
AccessControl, ReentrancyGuard, Pausable,
            if (s.cycleSeconds < MIN_HORIZON_SECONDS ||
s.cycleSeconds > MAX_REDEMPTION_SECONDS) {
                revert InvalidSchedule();
            }
-           if (s.nextRedemptionAt == 0) revert

```

```

InvalidSchedule();
+         if (s.nextRedemptionAt <= block.timestamp) revert
InvalidSchedule();
        // Until the first anchor passes, the horizon is not
yet capped by `cycleSeconds`.
        if (s.nextRedemptionAt > block.timestamp +
MAX_REDEMPTION_SECONDS) revert InvalidSchedule();
    }

```

1. **_secondsFor** reverts once a CYCLICAL anchor has passed, so fills are refused rather than mispriced.
2. **setSchedule** no longer reads the expired schedule's horizon. Without this, the bound itself calls `_secondsFor(prev)`, which now reverts, and the keeper could not publish the next date at all; only the admin could. An expired schedule has no current horizon to measure a shift from, so the shift is measured from the new date itself and comes to zero. The cycle-length check still applies, so a keeper cannot switch mode or change the cycle length beyond the bound in the same call.
3. **_validateSchedule** requires `nextRedemptionAt` to be in the future, since a past anchor would only make pricing revert.

With the fix, a late keeper publishes the real next date and fills resume on the true horizon. Recovery also works after several missed months, and ROLLING schedules are unaffected. All of these were checked by test.

Costs:

- **Liveness now depends on the keeper after each redemption date.** A missed update pauses fills for that asset instead of mispricing them. The keeper already has to update after every month whose length differs from `cycleSeconds` for the schedule to be correct at all.
- **Fills revert at every redemption date until the keeper publishes the next one.** While the current date is live the keeper cannot publish the next: an update an hour early reverts `ScheduleShiftTooLarge`. So even a punctual keeper leaves a window at each date, including on pairs where the asset is the maker's leg. Queuing the next date in advance (below) is the only full remedy.
- **The keeper's first date after an expiry is not shift-bounded.** Only its cycle length is.
- `cycleSeconds` no longer affects pricing; it only sizes the keeper's cycle-length check.
- **Tests that need updating.** With the fix, 11 tests fail, all asserting the automatic roll. Four are in `test/unit/OctarineRFQSchedule.t.sol`:

```

test_cyclical_wrapsBackToCycleLength,      test_cyclical_subDayCycle,
test_cyclical_autoRollsAcrossManyMissedCycles      and
testFuzz_cyclicalNeverExceedsCycle.      Seven      are      in
test/audit/ScheduleAmountVerification.t.sol:
test_cyclical_autoRollAcrossMissedCyclesPricesIdentically,
test_cyclical_fractionalDayCycle,
test_cyclical_quoteLegWrappingCollapsesTheDiscount,
test_cyclical_subDayCycle,
test_cyclical_wrapAtTheAnchorIsExactToTheSecond,
testFuzz_cyclicalAmountTracksTheRemainingSeconds,      and
test_deadBand_cyclicalOnlyBitesAfterTheWrap      (which      now      hits
InvalidSchedule() before the DiscountExceedsPrincipal it expects). Every other
test passes (695 passed, 1 skipped). They need the same update: publish the next date
instead of relying on the roll.

```

If some assets genuinely redeem on a fixed-length cycle, keep the roll for those behind an explicit per-schedule opt-in rather than for every CYCLICAL schedule. Alternatively, the keeper can publish the next several redemption dates at once, so that one missed update does not pause fills.

Remediation

Mitigated in 8ba441e and 15bbb56. The roll is unchanged; the fix is operational.

- The keeper bound now defaults to 7 days. That admits the 31 → 28 → 31 day changes the 2-day bound refused.
- The README now has a keeper runbook. Every update republishes the anchor together with the coming cycle's length. It runs early in each cycle, before payout – redemptionLead. A missed or reverted update raises an alert, and the admin path is kept ready.

We followed that runbook over a year of month-end payouts: every update was accepted and the horizon was exact. One missed early-cycle update costs at most the month-length difference, 10 to 30 USDC per 10,000 RWA at 10 bps/day. A keeper that runs close to the payout and misses once still causes the full loss (290 USDC per 10,000 RWA), which only the admin can then repair.

[M-05] The schedule has no notion of a redemption request cutoff, so fills after the filing deadline are charged for a payout the maker can no longer reach

Affected files

src/octarine-rfq/OctarineRFQ.sol#L110-L124

src/octarine-rfq/OctarineRFQ.sol#L127-L146

src/octarine-rfq/interfaces/IOctarineRFQ.sol#L199-L204

Description

For a CYCLICAL asset, `_secondsFor` charges the discount over the time until `nextRedemptionAt`, and the contract describes that anchor as the moment the buyer's RWA redeems: "Seconds until asset redeems ... an asset one hour from redemption prices at an hour of spread, not at a whole day of it" (IOctarineRFQ.sol L199-L204).

You told us the anchor will be the payout date, "because the payout is when the maker actually receives cash". You also told us that before the issuer's request cutoff passes, the keeper moves the anchor to the next payout date. The two CYCLICAL assets planned are Pareto AA (filing takes one block) and ACRED (filing takes up to a day).

The contract has no notion of that cutoff:

- **Who is caught.** A maker that buys the RWA too late to file before the cutoff, i.e. after the cutoff minus its filing time, cannot get into that payout's batch. It is paid at the next payout, a full cycle later.
- **What it is charged.** Until the anchor moves, the contract charges that maker for the wait to the current payout.
- **When the price is right.** Only if the anchor moves to the next payout at least the filing time before the cutoff, every cycle.

Two things make that move hard to land on time:

- **The keeper cannot make it.** Moving the anchor from one payout date to the next is a full-cycle shift. The keeper's shift bound defaults to 2 days, so a `SCHEDULE_KEEPER_ROLE` call reverts `ScheduleShiftTooLarge` (L127-L146). Only the admin can make the move in one call; the keeper would need about fifteen batched calls.
- **Every cycle depends on it.** If the move lands late, every fill between the filing deadline and the move is mispriced. If it does not land before the cutoff, every fill until the payout is.

The taker picks the moment of the fill: it submits the fill itself, or anyone holding both signatures does when `sender` is unset. Orders from the app run up to a month, so an order on a monthly

asset almost always spans a cutoff.

Vulnerable Scenario (an ACRED-like asset: request cutoff 5 days before payout, filing within a day):

1. A maker signs a 10.00 bps/day bid for 1,000 RWA, 29 days before the payout (quote 971 USDC, cap 1,000 USDC), with a one-month expiry.
2. Two days before the cutoff, the keeper's move to the next payout reverts. The admin makes it an hour before the cutoff.
3. Twelve hours before the cutoff, a taker fills. The maker pays 994.50 USDC, i.e. 5.5 days of discount. Its request, filed within a day, lands after the cutoff and is paid at the next payout.
4. Right after the move, the contract quotes the same lot at 964.96 USDC.

Impact

The maker pays for a payout it can no longer reach, and holds the RWA for another full cycle without being paid for the wait.

- **Size per struck fill.** About $\text{rate} \times \text{cycle}$: 0.6% of notional at 2 bps/day, 3% at 10 and 6% at 20, on a monthly cycle.
- **No principal loss.** The maker keeps an asset worth its NAV, but carries a cycle of exposure for free.
- **The cap does not stop it.** Late in a cycle, honest fills already cost close to the full NAV, so a cap that allows them also allows the strike.
- **How often.** It repeats every cycle in which the move is late or missed. While the keeper cannot make the move, every cycle depends on a manual admin transaction landing inside the window.

Proof of Concept

A runnable Foundry proof of concept for this finding is in the corresponding issue on the project tracker.

Recommendations

Tell the contract where the filing deadline is, instead of relying on when the anchor is moved. Add an admin-held, per-asset **redemption lead**: the gap between the issuer's request cutoff and the payout, plus the time a buyer needs to file (6 days in the scenario above). The schedule then counts as expired from anchor – lead:

- **Inside that window** fills are refused rather than mispriced.

- **The keeper** can then publish the next payout date in one call. An expired schedule's next date is not shift-bounded, but the cycle-length check still applies.

The diff applies on top of the fix recommended for [M-02], including the cycle-length correction described there:

```

--- a/src/octarine-rfq/OctarineRFQ.sol
+++ b/src/octarine-rfq/OctarineRFQ.sol
@@ -54,6 +54,10 @@

////////////////////////////////////*/

    mapping(address asset => RedemptionSchedule) private
    _schedules;
+   /// @notice CYCLICAL: seconds before the anchor (payout)
+   after which a buyer can no longer get into that
+   ///          payout's batch – the issuer's request cutoff plus
+   the time to file. Admin-set, so a keeper
+   ///          update cannot drop it.
+   mapping(address asset => uint32) public redemptionLead;

    mapping(address base => mapping(address quote =>
IOctarineOracle)) public oracles;
    mapping(address base => mapping(address quote => uint64))
public oracleMaxStaleness;
@@ -95,7 +99,7 @@
    function secondsToRedemption(address asset) public view
returns (uint32) {
    RedemptionSchedule memory s = _schedules[asset];
    if (!s.configured) revert ScheduleNotConfigured(asset);
-   return _secondsFor(s);
+   return _secondsFor(s, redemptionLead[asset]);
}

    function scheduleOf(address asset) external view returns
(RedemptionSchedule memory) {
@@ -104,16 +108,22 @@

```

```

    function _optionalSeconds(address asset) internal view
returns (uint32) {
    RedemptionSchedule memory s = _schedules[asset];
-    return s.configured ? _secondsFor(s) : 0;
+    return s.configured ? _secondsFor(s,
redemptionLead[asset]) : 0;
    }

-    function _secondsFor(RedemptionSchedule memory s) internal
view returns (uint32) {
+    function setRedemptionLead(address asset, uint32 lead)
external onlyRole(DEFAULT_ADMIN_ROLE) {
+        RedemptionSchedule memory s = _schedules[asset];
+        if (s.mode == ScheduleMode.CYCLICAL && s.configured &&
lead >= s.cycleSeconds) revert InvalidSchedule();
+        redemptionLead[asset] = lead;
+    }
+
+    function _secondsFor(RedemptionSchedule memory s, uint256
lead) internal view returns (uint32) {
        if (s.mode == ScheduleMode.ROLLING) return
s.rollingSeconds;

        // A calendar schedule cannot be extrapolated by a fixed
cycle: once the anchor passes, refuse
        // to price until the keeper publishes the next
redemption date.
        uint256 anchor = s.nextRedemptionAt;
-        if (block.timestamp >= anchor) revert InvalidSchedule();
+        if (block.timestamp + lead >= anchor) revert
InvalidSchedule();

        // Exact to the second: the last hours before redemption
price as hours, not as a whole day.
        uint256 remaining = anchor - block.timestamp;
@@ -126,14 +136,15 @@
        bool isAdmin = hasRole(DEFAULT_ADMIN_ROLE, msg.sender);
        if (!isAdmin && !hasRole(SCHEDULE_KEEPER_ROLE,

```

```

msg.sender)) revert Unauthorized();
    if (asset == address(0)) revert ZeroAddress();
-    _validateSchedule(s);
+    uint256 lead = redemptionLead[asset];
+    _validateSchedule(s, lead);

    RedemptionSchedule memory prev = _schedules[asset];
    // An expired CYCLICAL schedule has no current horizon:
measure no shift, but keep the cycle-length check.
-    bool expired = prev.mode == ScheduleMode.CYCLICAL &&
block.timestamp >= prev.nextRedemptionAt;
+    bool expired = prev.mode == ScheduleMode.CYCLICAL &&
block.timestamp + lead >= prev.nextRedemptionAt;
    if (!isAdmin && prev.configured &&
maxScheduleShiftSeconds != 0) {
-        uint256 after_ = _secondsFor(s);
-        uint256 before_ = expired ? after_ :
_secondsFor(prev);
+        uint256 after_ = _secondsFor(s, lead);
+        uint256 before_ = expired ? after_ :
_secondsFor(prev, lead);
        uint256 shift = before_ > after_ ? before_ - after_ :
after_ - before_;

        uint256 pAmp = prev.mode == ScheduleMode.ROLLING ?
before_ : prev.cycleSeconds;
@@ -153,7 +164,7 @@
        emit ScheduleUpdated(asset, s.mode, s.rollingSeconds,
s.nextRedemptionAt, s.cycleSeconds);
    }

-    function _validateSchedule(RedemptionSchedule calldata s)
internal view {
+    function _validateSchedule(RedemptionSchedule calldata s,
uint256 lead) internal view {
        if (s.maxBpsPerDay == 0) revert InvalidSchedule();

        if (s.mode == ScheduleMode.ROLLING) {

```

```

@@ -165,7 +176,8 @@
        if (s.cycleSeconds < MIN_HORIZON_SECONDS ||
s.cycleSeconds > MAX_REDEMPTION_SECONDS) {
            revert InvalidSchedule();
        }
-        if (s.nextRedemptionAt <= block.timestamp) revert
InvalidSchedule();
+        if (lead >= s.cycleSeconds) revert InvalidSchedule();
+        if (s.nextRedemptionAt <= block.timestamp + lead)
revert InvalidSchedule();
        // Until the first anchor passes, the horizon is not
yet capped by `cycleSeconds`.
        if (s.nextRedemptionAt > block.timestamp +
MAX_REDEMPTION_SECONDS) revert InvalidSchedule();
    }

```

The lead is held by the admin, outside the schedule struct, so a routine keeper update cannot drop it.

We checked this with the lead set to 6 days:

- A fill inside the window is refused with `InvalidSchedule`.
- Once the window opens, the keeper publishes the next payout in one call, and a fill is then priced to the next payout at the fair amount.
- A fill before the window is still priced to the current payout, which the maker makes.

With the [M-02] fix and the lead applied, the client test suite gives the same result as the [M-02] fix alone. With the lead at its default of zero, it adds no failures of its own.

Costs:

- **The lead must be set for each CYCLICAL asset.** Until it is, the asset behaves as it does under the [M-02] fix alone.
- **A refusal window every cycle.** Fills are refused between the filing deadline and the keeper's next update. That is the same per-date refusal the [M-02] fix already introduces, moved earlier by the lead.

Remediation

Fixed in ad5a529 and bdb4c1e. Each CYCLICAL asset now carries a `redemptionLead`. Once payout – `redemptionLead` passes, fills are priced to the next payout. With the lead

set to the issuer's cutoff plus a day of filing time, the maker pays the fair 964.50 USDC instead of 994.50.

bdb4c1e adds the follow-up hardening:

- only the admin sets the lead, and a keeper update that changes it reverts `Unauthorized`;
- `ScheduleUpdated` carries the lead;
- `setSchedule` refuses a lead on a ROLLING schedule, an anchor at or below the lead, and a `cycleSeconds + redemptionLead` above `MAX_REDEMPTION_SECONDS`.

Two operating points remain, both documented in the README:

- The lead must be set for every CYCLICAL asset. At zero, the original behaviour returns.
- The lead's value decides who bears a filing-time error, so set it from the issuer's real cutoff.

[M-06] OctarineCallOracle cannot refuse a price its own source flags as invalid: a defaulted Pareto tranche keeps settling at its pre-default price, and ULTRA's x100 unit change re-prices the route

Affected files

src/shared/oracles/OctarineCallOracle.sol#L54-L61

src/shared/oracles/OctarineCallOracle.sol#L13

Description

OctarineCallOracle prices a pair off one view call: word 0 of the return data, scaled by an immutable sourceDecimals (L54-L61). Its only notion of health is the optional timestamp call. Nothing else about the source can stop a read.

Two of the sources this adapter names for launch publish exactly the signals a consumer needs before trusting the number. The adapter cannot read either.

Pareto (priceAA ()): the tranche keeps its pre-default price after a default.

- When the borrower fails to pay at an epoch's end, the Pareto CDO records a hard default: defaulted () turns true, and the vault pauses and stops the epoch.
- It does **not** touch priceAA (); the loss is only booked later, by finalizeDefault.
- The AA token stays transferable.
- So after a default the adapter keeps returning the pre-default price as fresh. A timestamp call would not help: Pareto exposes no update time, and nothing newer exists to be stale against.

ULTRA (lastSetMintExchangeRate ()): the unit is an issuer parameter, and it has already changed ×100.

- ULTRA's rate means lastSetMintExchangeRate / BPS_DENOMINATOR (), and BPS_DENOMINATOR is a storage variable the issuer's admin can set.
- On Ethereum, after an upgrade at block 22437985 (8 May 2025), BPS_DENOMINATOR read 0 until block 22438191 and 1e6 from then on (it had been 1e4). The rate was republished at the new scale at block 22444277: 10071 became 1007100, both \$1.0071.
- An adapter deployed with sourceDecimals = 4 priced \$1.0071 before the change and \$100.71 after it, with no signal on-chain.

The design treats a bad oracle price as a staleness problem ("a stale RWA price misprices the entire fill"). For these two sources the price is not stale but wrong, and the source says so on-chain.

Impact

- **Pareto.** Once a default is recorded (and while it is not yet reflected in `priceAA()`), any taker holding a live maker bid can sell AA at the pre-default price.
 - The maker's signed cap does not bind, because the price never moved.
 - The maker's loss is the tranche's eventual impairment $\times$ the size filled, and it is unknown until the default is finalised.
 - The taker picks the moment: it fills itself, or anyone holding both signatures does when sender is unset. A taker can back-run the epoch-closing transaction before the maker sees the default and cancels.
- **ULTRA.** After a unit change, the route misprices $\times 100$ in one direction or the other.
 - Each counterparty's own signed bound limits the loss: `maxMakerAmount` or `minReceivedAmount`. With a realistic cap the fill reverts, so the route is dead until redeployed.
 - Any consumer of the same adapter without a per-order bound is exposed to the full $\times 100$.

Proof of Concept

A runnable Foundry proof of concept for this finding is in the corresponding issue on the project tracker.

Recommendations

Let each adapter pin one guard call whose answer must match, and treat any other answer, or a failed call, as no price:

```
@@ -14,9 +14,12 @@
    bool public immutable inverse;
    uint32 public immutable gasCap;
    address public immutable timestampTarget;
+   address public immutable guardTarget;
+   uint256 public immutable expectedGuard;

    bytes public callData;
    bytes public timestampCallData;
+   bytes public guardCallData;

    error SourceUnavailable();
    error TimestampUnavailable();
```

```

@@ -32,8 +35,14 @@
    bool inverse_,
    uint32 gasCap_,
    address timestampTarget_,
-   bytes memory timestampCallData_
+   bytes memory timestampCallData_,
+   address guardTarget_,
+   bytes memory guardCallData_,
+   uint256 expectedGuard_
) OctarineOracleBase(base_, quote_) {
+   guardTarget = guardTarget_;
+   guardCallData = guardCallData_;
+   expectedGuard = expectedGuard_;
    if (target_ == address(0) || callData_.length < 4 ||
gasCap_ == 0) {
        revert InvalidGuards();
    }
@@ -53,6 +62,10 @@

    function getPrice(address _base, address _quote) public view
returns (uint256 price, uint256 lastUpdated) {
    _checkPair(_base, _quote);
+   if (guardTarget != address(0)) {
+       (bool ok, uint256 g,) = _read(guardTarget,
guardCallData, gasCap, 1);
+       if (!ok || g != expectedGuard) return (0, 0);
+   }
    if (timestampTarget == address(0)) return
(_scale(_readSource(), sourceDecimals, inverse), block.timestamp);

    lastUpdated = _readTimestamp();

```

- **Pareto:** deploy with defaulted() as the guard, expected 0. Once a default is recorded, the route reads as down and fills revertNoPriceAvailable.
- **ULTRA:** deploy with BPS_DENOMINATOR() as the guard, expected the value at deployment. After a unit change the route reads as down until a new adapter is deployed at the new scale.

- **Construction:** if a guard target is set, also reject guard calldata shorter than 4 bytes in the constructor.

Both cases were fix-checked on the same forks with a guarded copy of the adapter:

- Pareto: priced before the default, NoPriceAvailable after it.
- ULTRA: no price after the switch, so the $\times 100$ fill cannot settle.

Three conditions for the guard to hold:

1. **Guard every route that prices the asset.** A tripped guard makes its route read as down, and the RFQ then uses the reference-asset route instead. If the asset also has an unguarded route, the fallback still prices it at the pre-default price. In our test, a guarded direct AA→USDT pair with an unguarded AA→USDC leg settled 1,111.79 USDT through the fallback after a recorded default.
2. **Guard the setter oracle too.** [M-04] recommends moving Pareto to OctarineThirdPartySetterOracle, which has no guard. A keeper re-pushing priceAA() after a default keeps it priced; in our test a fill 3 h after the default still settled 1,111.79 USDC. Give the setter oracle the same optional guard. The diff applies on top of the publication-time change recommended in [M-04]:

```
@@ -20,6 +20,13 @@
    uint32 public immutable minUpdateInterval;
    uint32 public immutable signerTimelock;

+    /// @notice Optional health call pinned at deployment (e.g.
+    Pareto `defaulted()` expecting 0). Any other
+    ///          answer, or a failed call, reads as no price, as
+    in {OctarineCallOracle}.
+    address public immutable guardTarget;
+    uint256 public immutable expectedGuard;
+    bytes public guardCallData;
+    uint32 internal constant GUARD_GAS = 100_000;
+
    uint256 public price;
    /// @notice When the SOURCE published `price` (signer-
    attested). Staleness is measured from this, not from
    ///          the push, so re-pushing an unchanged rate cannot
    keep a frozen source fresh.
```

```

@@ -58,7 +65,10 @@
    uint32 maxAge_,
    uint16 maxChangeBps_,
    uint32 minUpdateInterval_,
-   uint32 signerTimelock_
+   uint32 signerTimelock_,
+   address guardTarget_,
+   bytes memory guardCallData_,
+   uint256 expectedGuard_
) OctarineOracleBase(base_, quote_) Ownable(owner_) {
    if (
        maxAge_ == 0 || maxChangeBps_ == 0 || maxChangeBps_ >
MAX_CHANGE_BPS_CEILING || minUpdateInterval_ == 0
@@ -70,6 +80,10 @@
        maxChangeBps = maxChangeBps_;
        minUpdateInterval = minUpdateInterval_;
        signerTimelock = signerTimelock_;
+   guardTarget = guardTarget_;
+   guardCallData = guardCallData_;
+   expectedGuard = expectedGuard_;
+   if (guardTarget_ != address(0) && (guardCallData_.length
< 4 || !_guardOk())) revert InvalidGuards();
        for (uint256 i; i < signers.length; ++i) {
            isSigner[signers[i]] = true;
            emit SignerSet(signers[i], true);
@@ -78,10 +92,16 @@

    function getPrice(address _base, address _quote) external
view returns (uint256, uint256) {
        _checkPair(_base, _quote);
-       if (price == 0 || block.timestamp - updatedAt > maxAge)
return (0, updatedAt);
+       if (price == 0 || block.timestamp - updatedAt > maxAge ||
!_guardOk()) return (0, updatedAt);
        return (_scale(price, 18, false), updatedAt);
    }

+   function _guardOk() internal view returns (bool) {

```

```

+         if (guardTarget == address(0)) return true;
+         (bool ok, uint256 g,) = _read(guardTarget, guardCallData,
GUARD_GAS, 1);
+         return ok && g == expectedGuard;
+     }
+
    /// @notice The last pushed price however old, for marking
    positions only.
    function latestPriceUnchecked(address _base, address _quote)
    external view returns (uint256, uint256) {
        _checkPair(_base, _quote);

```

3. **Validate the guard at deployment.** An adapter whose guard call fails, or returns anything but the expected value, should refuse to deploy; otherwise the route is dead from day one:

```

@@ -65,6 +65,11 @@ contract OctarineCallOracle is
OctarineOracleBase {
    if (_readTimestamp() == 0) revert
TimestampUnavailable();
    }
    if (_readSource() == 0) revert SourceUnavailable();
+    // A guard that does not read as healthy at deployment
+    would leave the route dead from day one.
+    if (guardTarget_ != address(0)) {
+        (bool ok, uint256 g,) = _read(guardTarget_,
guardCallData_, gasCap_, 1);
+        if (!ok || g != expectedGuard_) revert
InvalidGuards();
+    }
    }

    function getPrice(address _base, address _quote) public view
    returns (uint256 price, uint256 lastUpdated) {

```

With all three applied, a recorded default leaves the asset unpriced on every route. We checked this with these changes and the other recommended oracle fixes applied together.

You told us you unroute an asset on an issuer event. The guard closes the window until the unroute lands: a taker can fill in the block after the default.

Two cautions about that runbook, which apply to any issuer event, not only a default:

- **Unroute every pair that involves the asset, in both directions.** `registerOracle(asset, USDC, address(0), 0)` stops fills against USDC, but a registered direct pair, such as the asset against another RWA, keeps pricing on its own oracle. In our test, after the USDC leg was unrouted during a 20% mark-down walked in 5% steps, an open quote on the direct pair filled at the first step: the maker paid 464.08 B for RWA worth 390.80 B at the published NAV (18.75% more).
- **Invalidating a direct pair's oracle does not leave the pair unpriced.** You described `invalidate()` followed by a new push as leaving the route "unpriced rather than mispriced in between". For a direct pair, the RFQ reads the invalidated oracle as down and prices the pair through the reference-asset route instead. In our test, ten minutes after `invalidate()` on a direct pair's setter oracle, and before the reference leg re-printed, a fill settled at twice the asset's new value (488.50 B for RWA worth 244.25 B).

Pausing the RFQ is the one control that stops every route at once. To stop a single asset, add a per-asset halt that `priceOf` checks before any route.

Costs and composition:

- **Constructor parameters.** Each adapter's constructor gains three parameters. The six calls in `test/unit/OctarineCallOracle.t.sol` and the four setter-oracle constructions in the tests need them.
- The change composes with the `live` flag recommended for [M-04]. Both add constructor parameters; applied together by hand, `src` compiles.

Separate from [M-04]. [M-04] is about sources with no clock. A clock would not catch either case here.

After an issuer unit change like ULTRA's, deploy the replacement adapter only once the issuer has republished the rate at the new scale. ULTRA's scale had changed for about 21 hours before its rate was republished.

Remediation

Fixed in `aace386`, documented in `b77231b`. `OctarineCallOracle` now takes a guard call whose answer must match, and an unhealthy guard is refused at deployment. `setHalted`, held by `PAUSER_ROLE`, stops every RFQ route that prices an asset.

On mainnet forks:

- Pareto, with `defaulted()` as the guard, reads as no price after the real default.
- ULTRA, with `BPS_DENOMINATOR()` as the guard, reads as no price after the unit change.

The README states that the halt covers RFQ pricing only, and that every route pricing a guarded asset needs the guard.

Two notes remain open:

- Halting the reference asset does not stop pairs that triangulate through it, although the `halted` natspec says no route may price a halted asset.
- `OctarineThirdPartySetterOracle` has no guard, so for an asset priced through it the halt is the only stop.

7.2 Low Severity Findings

[M-03] Revoking a delegated order signer only pauses it: registering the same key again revives every unexpired order it signed

Affected files

src/octarine-rfq/OctarineRFQ.sol#L278-L282

src/octarine-rfq/OctarineRFQ.sol#L292-L304

src/octarine-rfq/interfaces/IOctarineRFQ.sol#L254-L257

Description

A maker in cold custody lets a hot key sign its quotes with `registerAllowedOrderSigner(key, true)`. Every place the protocol describes revoking that key says it voids the key's outstanding orders:

- `IOctarineRFQ.sol` L256: "Revocation is immediate and invalidates every outstanding order that signer produced."
- `IOctarineRFQ.sol` L112-L113: revoking a delegated signer "remains the bulk kill-switch for one key's whole book."
- `src/octarine-rfq/README.md`, assumption 3: "revoking that signer voids every order it produced."
- `src/octarine-rfq/DESIGN.md`: "revocation invalidates every outstanding order that signer produced, which is the intended kill-switch."

The code does not void anything. A delegated signature is checked against a live flag at fill time (`_verifySignature`, L301):

```
&& (recovered == signer || (allowDelegate && orderSigners[signer][recovered]))
```

`registerAllowedOrderSigner` only writes that flag (L280):

```
orderSigners[msg.sender][signer] = allowed;
```

Revoking sets it to `false`. Nothing records that the key was revoked, and an order carries no signer generation. Registering the same key again sets the flag back to `true`, and every order

that key signed before the revocation becomes fillable again, unless it has expired or its salt was cancelled.

A kill-switch invites switching back on. A maker that pulls its book this way, for an issuer event, a pricing review or an infrastructure incident, and later re-enables its quoting key to resume, revives the whole old book at the old rates. A taker can watch for `OrderSignerRegistered(maker, key, true)` and fill in the next block. The taker sends the fill itself, or anyone holding both signatures does when sender is unset, which is how the app sends fills. Orders from the app carry expiries of 24 hours to one month, so the window in which a re-enable revives old orders is long.

Vulnerable Scenario:

1. A maker's hot key signs a 10.00 bps/day quote for 1,000 RWA; the taker signs its request.
2. The maker revokes the key. The order now reverts with `InvalidSignature`.
3. Hours later the maker re-enables the same key and quotes 15.00 bps/day.
4. The taker fills the old order. The maker pays 970 USDC for a lot its live book prices at 955 USDC.

Impact

The maker trades on orders it retracted, at terms it no longer offers, across every unexpired order the key signed. Each order is bounded by its own `maxMakerAmount` and `takerAmount`. Those were signed under the old conditions, though, so they do not bound the gap between the old terms and the maker's current view. In the PoC the gap is 15 USDC per 1,000 RWA (1.5%). If the maker pulled its book because of an issuer event the oracle has not priced yet, and re-enables the key before the oracle catches up, the loss is the NAV gap, up to each order's cap.

Proof of Concept

A runnable Foundry proof of concept for this finding is in the corresponding issue on the project tracker.

Recommendations

Option A (recommended): make revocation permanent for that key.

```
@@ -67,6 +67,7 @@ contract OctarineRFQ is IOctarineRFQ,
AccessControl, ReentrancyGuard, Pausable,
    mapping(bytes32 orderHash => uint128) public rfqFilled;
    mapping(bytes32 requestHash => uint128) public requestFilled;
    mapping(address maker => mapping(address signer => bool))
```

```

public orderSigners;
+   mapping(address maker => mapping(address signer => bool))
public orderSignerRevoked;
    mapping(address signer => mapping(uint256 salt => bool))
public saltCancelled;

/*//////////////////////////////////////
@@ -277,6 +278,8 @@ contract IOctarineRFQ is IOctarineRFQ,
AccessControl, ReentrancyGuard, Pausable,
    /// @inheritdoc IOctarineRFQ
    function registerAllowedOrderSigner(address signer, bool
allowed) external {
        if (signer == address(0)) revert ZeroAddress();
+       if (allowed && orderSignerRevoked[msg.sender][signer])
revert Unauthorized();
+       if (!allowed) orderSignerRevoked[msg.sender][signer] =
true;

        orderSigners[msg.sender][signer] = allowed;
        emit OrderSignerRegistered(msg.sender, signer, allowed);
    }

```

The maker rotates to a fresh key instead of re-enabling the old one. The fresh key fills normally (lifecycle test above). Option A costs one storage write per revocation, and a key revoked by mistake cannot come back, so the maker registers a new one. The full test suite passes with this change (706 passed, 1 skipped).

Option B: if pause-and-resume is the intended behaviour, say so in the four places above. Tell makers to cancelSalt their outstanding salts, or wait past the longest expiry they accepted, before re-enabling a key.

Remediation

Acknowledged, and regraded to **Low** (Medium at delivery). Re-enabling a key still revives its orders, by design. In 70b34f4 the client applied Option B. All four places now describe revocation as a pause, not a cancellation:

- the registerAllowedOrderSigner natspec;
- the RFQOrder natspec;

- README assumption 3;
- DESIGN.md.

They also tell makers to cancel outstanding salts or wait out the longest expiry before re-enabling a key, and to register a new key to retire one for good. A maker that follows the documentation is no longer exposed, which is why the finding is now graded Low.

[M-04] CallOracle without a timestamp call reports every read as fresh, so the RFQ's mandatory staleness bound never applies to issuer-set rates

Affected files

src/shared/oracles/OctarineCallOracle.sol#L54-L61

src/octarine-rfq/OctarineRFQ.sol#L194-L203

Description

OctarineRFQ treats staleness as mandatory on every route. `registerOracle` refuses a zero staleness (L210), and `_tryOracle` documents that a registered feed yields "0 if it is absent, reverting, zero or stale" (L194-L203). It checks `block.timestamp <= updatedAt + oracleMaxStaleness[base]` [quote] against the `lastUpdated` the adapter returns.

`OctarineCallOracle` without a timestamp call returns `block.timestamp` as `lastUpdated` (L56):

```
if (timestampTarget == address(0)) return (_scale(_readSource(),
sourceDecimals, inverse), block.timestamp);
```

The read is therefore reported as current on every call, and the RFQ's staleness bound can never fire for that route, whatever value it was registered with. The constructor documents an empty timestamp call as being "for a live source". But the contract's own list of supported sources (L7-L10) includes rates an issuer sets and leaves until it sets them again, and at least one of them cannot be given a timestamp call at all. BENJI's Ethereum token (0x3ddc84940ab509c11b20b76b466933f40b750dc9, a verified EIP-1967 proxy to MoneyMarketFund_V6) exposes `lastKnownPrice()` and no update-time getter of any kind.

If an issuer stops maintaining such a rate, for example after migrating to a new contract, suspending the fund, or simply missing an update, fills keep settling on the last value indefinitely. Both counterparties' signed bounds are derived from that same stale price, so neither bound limits the error. The loss to whichever side the stale price disfavours is the full gap between the frozen rate and the asset's current value.

Proof of Concept

A runnable Foundry proof of concept for this finding is in the corresponding issue on the project tracker.

Recommendations

Make "always fresh" an explicit, recorded decision, rather than the default whenever the timestamp call is left empty:

```
@@ -15,4 +15,7 @@ contract OctarineCallOracle is
OctarineOracleBase {
    uint32 public immutable gasCap;
    address public immutable timestampTarget;
+    /// @notice Set only for a source that computes its price on
+    /// every read (e.g. ERC-4626
+    /// `convertToAssets`). An issuer-set rate is not
+    /// live, even if it has no update-time getter.
+    bool public immutable live;

    bytes public callData;
@@ -33,6 +36,9 @@ contract OctarineCallOracle is
OctarineOracleBase {
    uint32 gasCap_,
    address timestampTarget_,
-    bytes memory timestampCallData_
+    bytes memory timestampCallData_,
+    bool live_
    ) OctarineOracleBase(base_, quote_) {
+    if ((timestampCallData_.length == 0) != live_) revert
InvalidGuards();
+    live = live_;
    if (target_ == address(0) || callData_.length < 4 ||
gasCap_ == 0) {
        revert InvalidGuards();
    }
```

The flag records the choice, but it cannot stop someone declaring an issuer-set rate "live". For an issuer-set rate with no on-chain update time, such as Pareto's AA price, this adapter has no way to detect staleness. Where a source publishes a related time, such as ULTRA's epoch start, point timestampTarget at it; it bounds an outage of the issuer's process rather than the age of the rate itself. Otherwise price the rate through OctarineThirdPartySetterOracle, with two conditions:

- **Age the price from the issuer's publication, not from the push.** The setter oracle measures age from the moment of the push. A keeper that re-pushes an unchanged rate to keep the route alive therefore keeps a frozen issuer rate fresh indefinitely. In our test (not included here), 30 days after the issuer's last publication, the oracle's age was 0 s and a fill still settled; the maker paid 1,004.08 USDC, 56.93 more than at a 0.95 mark. Close this in one of two ways:
 - **Keeper policy:** push only when the issuer publishes a new rate, never on a timer, and set maxAge above the issuer's longest normal gap (about 5 days for ULTRA and OUSG, about 35 days for Pareto's epochs).
 - **On-chain:** have the signer attest the publication time. Staleness is then measured from it, and the update interval from the push.

The diff applies on top of the fixes recommended for [M-01] (Option A) and [L-04]. Apply it together with [L-07], keeping pushedAt per signer: the guarantee in its natspec depends on that. With one interval shared by all signers, a compromised signer that takes every slot with an old observedAt gets every honest refresh refused with TooSoon, and the live price ages out after maxAge; in our test a fill then paid 1,018.50 instead of 970.00. With pushedAt per signer, no refresh was refused and the fill paid 970.00.

```

--- a/src/shared/oracles/OctarineThirdPartySetterOracle.sol
+++ b/src/shared/oracles/OctarineThirdPartySetterOracle.sol
@@ -21,12 +21,17 @@
     uint32 public immutable signerTimelock;

    uint256 public price;
+   /// @notice When the SOURCE published `price` (signer-
+   ///         attested). Staleness is measured from this, not from
+   ///         the push, so re-pushing an unchanged rate cannot
+   ///         keep a frozen source fresh.
    uint64 public updatedAt;
+   /// @notice When `price` was last pushed. `minUpdateInterval`
+   ///         is measured from this.
+   uint64 public pushedAt;

    /// @notice With no live price (first push, or after
    ///         {invalidate}), a push is only proposed; it goes live
    ///         when a different signer pushes the same value.
    uint256 public pendingPrice;

```

```

        address public pendingSigner;
+       uint64 public pendingObservedAt;

        mapping(address => bool) public isSigner;
        mapping(address => uint64) public signerSubmittedAt;
@@ -44,6 +49,7 @@
        error ChangeTooLarge(uint256 previous, uint256 next);
        error NoPendingSigner();
        error TimelockNotElapsed(uint64 validAt);
+       error BadObservation(uint64 observedAt);

        constructor(
            address owner_,
@@ -84,26 +90,34 @@
        }

        /// @notice Push a new price. The first push, and the first
        after {invalidate}, is not change-bounded.
-       function pushPrice(uint256 newPrice) external {
+       /// @param observedAt When the source published `newPrice`;
+       never in the future. An older time than the stored
+       ///         one is accepted but never moves `updatedAt` back,
+       so a rogue signer can neither lock out an honest
+       ///         correction carrying the issuer's real time nor
+       make a live price read stale.
+       function pushPrice(uint256 newPrice, uint64 observedAt)
        external {
            if (!isSigner[msg.sender]) revert NotSigner();
            if (newPrice == 0) revert ZeroPrice();
+           if (observedAt > block.timestamp) revert
        BadObservation(observedAt);
            uint256 prev = price;
            if (prev == 0) {
                if (pendingPrice != newPrice || pendingSigner ==
address(0) || pendingSigner == msg.sender) {
                    pendingPrice = newPrice;
                    pendingSigner = msg.sender;
+                   pendingObservedAt = observedAt;

```

```

        emit PriceProposed(msg.sender, newPrice);
        return;
    }
+       if (pendingObservedAt > observedAt) observedAt =
pendingObservedAt;
        delete pendingPrice;
        delete pendingSigner;
+       delete pendingObservedAt;
    } else {
-       uint64 nextAt = updatedAt + minUpdateInterval;
+       uint64 nextAt = pushedAt + minUpdateInterval;
        if (block.timestamp < nextAt) revert TooSoon(nextAt);
        if (_deviates(newPrice, prev, maxChangeBps)) revert
ChangeTooLarge(prev, newPrice);
    }
    price = newPrice;
-    updatedAt = uint64(block.timestamp);
+    if (prev == 0 || observedAt > updatedAt) updatedAt =
observedAt;
+    pushedAt = uint64(block.timestamp);
    emit PricePushed(msg.sender, newPrice);
}

@@ -111,6 +125,7 @@
    delete price;
    delete pendingPrice;
    delete pendingSigner;
+    delete pendingObservedAt;
    emit PriceInvalidated();
}

```

Never let an older observedAt move the publication time back, but do not reject it either. Rejecting it would let a compromised signer stamp the current time and lock out an honest correction carrying the issuer's real, older time. Accepting it as-is would let that signer stamp an old time whenever it wins a slot: the route then reads as stale at once and refuses fills, and a direct pair silently reprices on the reference-asset route (in our test, a fill paid 1,018.50 instead of 970.00). The diff keeps the later of the two times, including when a second signer confirms a

proposed price. A compromised signer can still keep a frozen price fresh by stamping the current time, which it can already do today.

- **Apply that oracle's own fixes first:** the reset fix in [M-01], the guard values in [L-04], and, for a source that can default, the guard in [M-06].

Also mark, in the supported-source list at L7-L10, which sources need a timestamp call.

Costs:

- The constructor gains a parameter. Nothing in `src/` or `script/` constructs `OctarineCallOracle`; the 6 calls in `test/unit/OctarineCallOracle.t.sol` need the extra argument.
- Moving no-clock sources to the setter oracle adds a keeper dependency for those assets.
- The on-chain publication time changes `pushPrice`'s signature. Keepers must supply the issuer's publication time, and the 10 `pushPrice` calls in 4 test files need the extra argument.

Remediation

Acknowledged, and regraded to **Low** by agreement (Medium at delivery). A source without a timestamp call still reports every read as current, by design. Since 3bd8a2a the adapter exposes that choice as `live()`, readable on-chain.

The launch exposure that set the original severity is gone:

- ULTRA is now bounded through its epoch timestamp. That bounds the issuer's daily process rather than the rate itself.
- A Pareto credit event is caught by the `defaulted()` guard from [M-06].

One gap remains: a deployment that passes a timestamp target with empty timestamp calldata deploys as `live` and silently discards the target. Refusing that combination in the constructor would make the choice explicit.

[L-02] The submitter's gas limit can starve a healthy direct CallOracle read, so the fill is priced on the triangulated route

Affected files

src/octarine-rfq/OctarineRFQ.sol#L194-L203
src/shared/oracles/OctarineOracleBase.sol#L48-L62

Description

priceOf uses the direct base/quote feed "if one is registered and healthy" (IOctarineRFQ.sol L212-L222), and otherwise triangulates through the reference asset. Health is decided in `_tryOracle`, which maps every failure to 0 (L195-L203):

```
try oracle.getPrice(base, quote) returns (uint256 p, uint256
updatedAt) {
    if (p != 0 && block.timestamp <= updatedAt +
oracleMaxStaleness[base][quote]) return p;
} catch {}
return 0;
```

OctarineCallOracle reads its source with `staticcall(gasCap, ...)` (OctarineOracleBase.sol L55) and reports a failed read as price 0. It never checks that it actually had gasCap to forward. Under EIP-150, a caller that arrives with less gives the source only 63/64 of what is left. So the submitter of `fillRFQOrder` can choose a gas limit that is too small for the direct source but large enough for the rest of the fill. The direct read then fails, `_tryOracle` treats the healthy feed as down, and the fill settles on the triangulated price instead. Whoever sends the transaction picks the route; that is the taker on its own fill, or anyone when `sender == address(0)`.

The RFQ keeps only a small fraction of its gas after a starved read, so the route can only be forced when the direct source's read costs far more than the rest of the fill. In a separate sweep of gas limits (not included here), a source costing about 1M gas was never forced. One costing about 3M gas was forced at 178 of the 234 gas limits that settled. No source the adapter lists (ULTRA, Pareto, BENJI, ERC-4626, Midas, Ondo) is anywhere near that. The issue applies to a direct pair whose source is that expensive, for example an ERC-4626 vault whose `convertToAssets` walks many markets. The loss per fill is the gap between the two routes' prices, up to the counterparty's signed bound.

Proof of Concept

A runnable Foundry proof of concept for this finding is in the corresponding issue on the project tracker.

Recommendations

Reserve the oracle's gas up front, instead of inferring starvation after the fact:

- give each direct read a fixed budget;
- refuse the fill if the caller cannot afford it;
- call with exactly that budget.

```
@@ -57,6 +57,11 @@ contract OctarineRFQ is IOctarineRFQ,
AccessControl, ReentrancyGuard, Pausable,

    mapping(address base => mapping(address quote =>
IOctarineOracle)) public oracles;
    mapping(address base => mapping(address quote => uint64))
public oracleMaxStaleness;
+    /// @notice Gas a direct-feed read is handed. A read is never
+    /// attempted with less: if the caller cannot
+    /// afford the budget the call reverts, so a starved
+    /// read can never be taken for a feed that is down.
+    mapping(address base => mapping(address quote => uint32))
public oracleGas;
+
+    uint32 public constant DEFAULT_ORACLE_GAS = 300_000;

    address public referenceAsset;

@@ -196,12 +201,21 @@ contract OctarineRFQ is IOctarineRFQ,
AccessControl, ReentrancyGuard, Pausable,
    IOctarineOracle oracle = oracles[base][quote];
    if (address(oracle) == address(0)) return 0;

-    try oracle.getPrice(base, quote) returns (uint256 p,
uint256 updatedAt) {
+    uint256 budget = oracleGas[base][quote];
+    if (budget == 0) budget = DEFAULT_ORACLE_GAS;
```

```

+      // Refuse rather than starve: the oracle always receives
+      // its whole budget, so a failure inside
+      // it is the feed's, and a down feed keeps its documented
+      // fallback however it fails.
+      if (gasleft() < budget + budget / 63 + 10_000) revert
+      NoPriceAvailable(base, quote);
+      try oracle.getPrice{gas: budget}(base, quote) returns
+      (uint256 p, uint256 updatedAt) {
+          if (p != 0 && block.timestamp <= updatedAt +
+      oracleMaxStaleness[base][quote]) return p;
+      } catch {}
+      return 0;
+  }

+  function setOracleGas(address base, address quote, uint32
+  gas_) external onlyRole(DEFAULT_ADMIN_ROLE) {
+      oracleGas[base][quote] = gas_;
+  }
+
+  function registerOracle(address base, address quote, address
+  oracle, uint64 staleness)
+      external
+      onlyRole(DEFAULT_ADMIN_ROLE)

```

What this changes:

- **No starvation.** The oracle always receives its whole budget, so a submitter can no longer starve a healthy read into the triangulated route.
- **The fallback still works.** A feed that is genuinely down fails inside its own budget, and still falls back to triangulation as documented, however it fails.
- **Why not a simpler check.** Comparing `gasleft()` before and after the call cannot tell the two apart. Behind an adapter that forwards all its gas, as `OctarineChainlinkOracle` does, a feed that fails by consuming everything it is given (for example a pre-0.8 `assert`) always leaves less than 1/16 of the gas behind. Feeds on such a down feed would then revert at every gas limit instead of falling back.

We checked this with a feed costing about 3M gas and a down feed that consumes all its gas:

- No gas limit forced the triangulated route for the healthy feed.

- The down feed fell back at every gas limit that could afford its budget.

With this change, the full test suite passes.

The costs:

- **Budget sizing.** The admin must size each pair's budget for its adapter's worst case: about gasCap for a CallOracle, up to three times gasCap for one with a timestamp call and a guard. An undersized budget makes a healthy route read as down. The default is 300,000 gas, which covers the adapters' usual sources (5–12k gas).
- **Gas limit.** A fill's gas limit must cover each direct read's budget plus about 1/63 of it. Unused gas is refunded.

Remediation

Mitigated in 5896f96, with no code change. The gasCap natspec and the README now cap gasCap at 300,000 per call, with guidance for sizing it. In our sweep at that cap, no gas limit forced the reference route, even with a guard and a timestamp call on the same adapter. The cap is a deployment rule that is not enforced on-chain. An adapter deployed with a larger gasCap over a costly source is still exposed.

[L-03] The Chainlink adapter rejects every Chronicle answer (startedAt is always 0), so Chronicle-priced assets cannot be filled or are silently repriced

Affected files

src/shared/oracles/OctarineChainlinkOracle.sol#L45

Description

OctarineChainlinkOracle is the AggregatorV3 adapter for "Chainlink, RedStone, Chronicle, Superstate" (src/shared/README.md L11). getPrice treats a zero startedAt as an invalid round (L45):

```
if (roundId == 0 || answeredInRound < roundId || startedAt == 0
    || updatedAt == 0) return (0, updatedAt);
```

Chronicle's Scribe implements latestRoundData() with startedAt hard-coded to zero. In src/Scribe.sol L289-L307, it sets roundId = 1, startedAt = 0, updatedAt to the poke age, and answeredInRound = roundId. Every Chronicle answer, however fresh, therefore returns price 0 from getPrice, while latestPriceUnchecked (which has no startedAt check) serves the same answer. What happens next depends on how the adapter is registered in OctarineRFQ:

- **As the asset's leg to the reference asset:** _unitPrice reverts with NoPriceAvailable, so no fill on that asset can settle.
- **As a direct pair:** _tryOracle treats the feed as down, and the fill settles on the reference-asset route instead of the Chronicle price, without a revert or any other signal.

The other checks already cover what startedAt == 0 was meant to catch: updatedAt == 0 rejects a round that never completed, and both staleness bounds reject an old one.

Proof of Concept

A runnable Foundry proof of concept for this finding is in the corresponding issue on the project tracker.

Recommendations

Drop the startedAt clause (and its now-unused binding):

```
@@ -38,11 +38,10 @@ contract OctarineChainlinkOracle is
    OctarineOracleBase, Ownable {
```

```

{
    _checkPair(_base, _quote);

-   (uint80 roundId, int256 answer, uint256 startedAt,
uint256 updatedAt, uint80 answeredInRound) =
-   feed.latestRoundData();
+   (uint80 roundId, int256 answer,, uint256 updatedAt,
uint80 answeredInRound) = feed.latestRoundData();
    lastUpdated = updatedAt;

-   if (roundId == 0 || answeredInRound < roundId ||
startedAt == 0 || updatedAt == 0) return (0, updatedAt);
+   if (roundId == 0 || answeredInRound < roundId ||
updatedAt == 0) return (0, updatedAt);
    if (answer <= 0) return (0, updatedAt);
    if (updatedAt > block.timestamp || block.timestamp -
updatedAt > maxStaleness) return (0, updatedAt);

```

With this change the full test suite passes (706 passed, 1 skipped).

Remediation

Fixed in 413a466, as recommended. The Chronicle PoC's harm tests now fail, and the lifecycle test passes.

[L-04] Setter oracle accepts guard values that switch off its bounds, so one signer key can move the price without limit in a single block

Affected files

src/shared/oracles/OctarineThirdPartySetterOracle.sol#L51
src/shared/oracles/OctarineThirdPartySetterOracle.sol#L79-L83

Description

OctarineThirdPartySetterOracle bounds a signer in two ways: each push may move the price by at most `maxChangeBps`, and a push may not land sooner than `minUpdateInterval` after the last one. Its natspec states the purpose: "so a signer key cannot walk the price far in one sitting" (L11-L15). The README relies on it in the same way: "Manual prices are bounded on the oracle (OctarineThirdPartySetterOracle), not here."

The constructor validates `maxAge`, `maxChangeBps` and `signerTimeLock`, but not the two values that make those bounds meaningful (L51):

```
if (maxAge_ == 0 || maxChangeBps_ == 0 || signerTimeLock_ == 0)
    revert InvalidGuards();
```

- **`minUpdateInterval = 0` is accepted.** TooSoon can then never fire (L80-L81), so a signer may push any number of times in the same block, each push moving the price by up to `maxChangeBps` from the previous one. Ten pushes at 10% compound to $1.1^{10} \approx 2.59x$. The bound limits the step but no longer limits the walk. The repository's own RFQ fixtures deploy the oracle this way (test/unit/OctarineRFQPricing.t.sol:181, test/unit/OctarineRFQPushedPriceScale.t.sol:36, test/integration/OctarineRFQIntegration.t.sol:258, all `minUpdateInterval = 0, maxChangeBps = 1_000`).
- **`maxChangeBps` is a `uint16` with no ceiling.** At or above `10_000`, `_deviates` admits any decrease, so one correctly spaced push can take the price to 1 wei.

Both parameters are immutable, so an oracle deployed with either value carries it until it is replaced.

Vulnerable Scenario:

1. An oracle is deployed with `minUpdateInterval = 0` (as in the fixtures) and registered in OctarineRFQ. A maker has a live open quote on the asset with `maxMakerAmount`

headroom above the current price.

2. A signer key is compromised. In one block it pushes +10%, +10% and then to 1.30x. Each push is within maxChangeBps, and none is refused.
3. In the same block the attacker fills the open quote as taker. quoteRFQOrder prices the fill at the walked price, and the maker pays up to its full cap for RWA worth 1.00x.

Impact

One compromised signer key can set an arbitrary price in a single block. What a fill can then take is capped by the counterparty's own signed bound: maxMakerAmount on any open quote, which any address can fill, and maxMakerAmount or minReceivedAmount on any order the attacker is the counterparty to. So the cost of the missing interval is the part of that headroom beyond one maxChangeBps step, which is all a correctly spaced oracle lets a key take per interval.

- **With a large step**, such as the fixtures' 10%, one spaced push already covers a typical 1–2% headroom, and the zero interval adds little.
- **With a NAV-scale step**, it hands over the whole headroom at once. With a 0.5% step and a maker that left 2% over the fair cost, the maker pays 10,169.40 USDC for 10,000 RWA in one block. A correctly spaced oracle limits it to 10,019.85 USDC for that interval.

Any consumer of the same oracle that has no per-order cap is exposed to the full walk.

Proof of Concept

A runnable Foundry proof of concept for this finding is in the corresponding issue on the project tracker.

Recommendations

Refuse the values that switch the bounds off, and require enough room for the bounds to work in practice:

```
--- a/src/octarine-rfq/OctarineRFQ.sol
+++ b/src/octarine-rfq/OctarineRFQ.sol
@@ -208,6 +208,11 @@ contract OctarineRFQ is IOctarineRFQ,
AccessControl, ReentrancyGuard, Pausable,
    {
        if (base == address(0) || quote == address(0)) revert
ZeroAddress();
        if (oracle != address(0) && staleness == 0) revert
InvalidStaleness();
```

```

+         if (oracle != address(0)) {
+             // A pushed source must be refreshable well inside
the window fills accept it for.
+             (bool ok, bytes memory d) =
oracle.staticcall(abi.encodeWithSignature("minUpdateInterval()"));
+             if (ok && d.length == 32 && uint256(staleness) < 2 *
abi.decode(d, (uint256))) revert InvalidStaleness();
+         }

        oracles[base][quote] = IOctarineOracle(oracle);
        oracleMaxStaleness[base][quote] = staleness;
--- a/src/shared/oracles/OctarineThirdPartySetterOracle.sol
+++ b/src/shared/oracles/OctarineThirdPartySetterOracle.sol
@@ -14,6 +14,7 @@ import {OctarineOracleBase} from
"./OctarineOracleBase.sol";
    ///      key cannot walk the price far in one sitting. Adding a
signer waits out `signerTimelock`; removing one
    ///      is immediate.
    contract OctarineThirdPartySetterOracle is OctarineOracleBase,
Ownable2Step {
+    uint16 public constant MAX_CHANGE_BPS_CEILING = 2_000;
        uint32 public immutable maxAge;
        uint16 public immutable maxChangeBps;
        uint32 public immutable minUpdateInterval;
@@ -48,7 +49,12 @@ contract OctarineThirdPartySetterOracle is
OctarineOracleBase, Ownable2Step {
        uint32 minUpdateInterval_,
        uint32 signerTimelock_
    ) OctarineOracleBase(base_, quote_) Ownable(owner_) {
-        if (maxAge_ == 0 || maxChangeBps_ == 0 || signerTimelock_
== 0) revert InvalidGuards();
+        if (
+            maxAge_ == 0 || maxChangeBps_ == 0 || maxChangeBps_ >
MAX_CHANGE_BPS_CEILING || minUpdateInterval_ == 0
+            || uint256(minUpdateInterval_) * 2 > maxAge_ ||
signerTimelock_ == 0
+        ) {
+            revert InvalidGuards();

```

```
+      }
      maxAge = maxAge_;
      maxChangeBps = maxChangeBps_;
      minUpdateInterval = minUpdateInterval_;
```

- **A ceiling on maxChangeBps, not only below 10,000.** Rejecting `maxChangeBps >= 10_000` closes the collapse to 1 wei, but 9,999 would still let one push take the price down to 0.01% of its previous value. A pair that divides by this price (an inverse leg, or the quote leg of a cross) would then move $\times 10,000$ in a single push. Choose a ceiling a real NAV never needs in one step; the diff uses 20%.
- **A non-zero interval with headroom: $2 \times \text{minUpdateInterval} \leq \text{maxAge}$.** With the interval equal to `maxAge`, a keeper one block late leaves the route without a price for a few seconds every cycle.
- **An RFQ staleness of at least twice the interval.** `registerOracle` now refuses a staleness below twice the oracle's `minUpdateInterval`. Otherwise the route is down for most of every interval: in our test (not included here), with a 1-hour interval and 30-minute staleness, it was down for 1,799 s of every hour.

These values are chosen per asset by whoever deploys and operates each oracle, so the constructor and `registerOracle` are the only places the protocol can hold every deployment to them.

We checked this, with the ceiling at 20%:

- 9,999 and 2,001 are refused.
- An interval above half of `maxAge` is refused.
- A configuration at the limits deploys, prices, and accepts a full 20% step.
- `registerOracle` refuses a staleness below twice the interval and accepts exactly twice.
- The full test suite passes once the three fixtures above use a 1-second interval. Their `maxAge` and staleness are 1 hour.
- No test depends on several pushes in one block.

A non-zero floor is necessary, not sufficient. At `minUpdateInterval = 1` a signer can still walk `maxChangeBps` per second, and over time t the bound is $(1 + \text{maxChangeBps} / 10_000)^{(t / \text{minUpdateInterval})}$. Choose `minUpdateInterval` against how fast a bad push can be detected and the signer removed; a 24-hour interval then needs a `maxAge` of at least 48 hours. Alternatively, bound the total move per time window rather than per push. That costs one stored anchor price and timestamp.

Remediation

Fixed in ce92212, documented in 00407fe. The constructor refuses:

- a zero `minUpdateInterval`;
- a `maxChangeBps` above 2,500 (25%);
- an interval above half of `maxAge`.

The README adds the two deployment rules we asked for:

- an RFQ staleness of at least twice the oracle's interval;
- `maxChangeBps` sized for the number of signers and for inverse legs: at 25%, one push moves an inverse leg $\times 1.25$.

The staleness rule is documented, not enforced in `registerOracle`. A registration below it leaves the route down for most of every interval.

[L-05] The maker leg's measured receipt is not clamped, so a token that credits balances mid-transfer lets a fill settle below the taker's signed floor

Affected files

src/octarine-rfq/OctarineRFQ.sol#L377-L381

src/octarine-rfq/OctarineRFQ.sol#L430-L470

Description

`fillRFQOrder` measures both legs by balance difference, but only one of them is clamped. The taker leg is clamped to the amount sent (`_deliverTakerLeg`, L430-L443), and the comment explains why:

it is clamped to amount so a token that credits *more* than it was sent — a rebase landing mid-transfer — cannot inflate the maker's bill beyond the order's terms.

The maker leg is measured the same way but not clamped (`_payMakerLegs`, L466-L468):

```
uint256 before_ = makerToken.balanceOf(receiver);
makerToken.safeTransferFrom(payer, receiver, proceeds);
received = makerToken.balanceOf(receiver) - before_;
```

That `received` is what the taker's signed floor is checked against (L377-L381, `_checkTakerFloor`). If the maker token credits balances during the transfer, that is the case the taker-leg comment describes, then the credit on the taker's existing balance is counted as proceeds. A fill whose real proceeds are below the taker's floor then passes the check, and the taker's whole input is delivered.

The fill moment is chosen by the taker, or by anyone holding both signatures when `sender` is unset. So a fill can be timed to land on a scheduled rebase.

Impact

The taker sells below the floor it signed. The shortfall is bounded by the credit that lands on the taker's own balance of the maker token during the proceeds transfer. In the PoC, a taker holding 100,000 units and hit by a +0.01% rebase receives 960.30 against a 965 floor (0.49% short).

The precondition is a maker token whose balances change inside a transfer. None of the assets named for launch behaves that way: BENJI, Pareto, ULTRA, OUSG and USDC. The taker-leg comment shows the code already anticipates this token class on the other leg.

Proof of Concept

A runnable Foundry proof of concept for this finding is in the corresponding issue on the project tracker.

Recommendations

Clamp the maker leg the same way as the taker leg:

```
@@ -466,6 +466,7 @@ contract OctarineRFQ is IOctarineRFQ,
    AccessControl, ReentrancyGuard, Pausable,
        uint256 before_ = makerToken.balanceOf(receiver);
        makerToken.safeTransferFrom(payer, receiver, proceeds);
        received = makerToken.balanceOf(receiver) - before_;
+       if (received > proceeds) received = proceeds;
        if (fee > 0) makerToken.safeTransferFrom(payer,
feeRecipient, fee);
    }
```

The full test suite passes with this change (706 passed, 1 skipped).

The clamp closes the case shown, but one token class still slips past the floor: a token that both charges a transfer fee and credits a rebase in the same transfer. The credit masks the fee, so the measured receipt never exceeds the proceeds and the clamp never engages. In our test (not included here), the taker received 987.03 against a 990 floor. No balance-delta check can separate the two, and the taker leg has the same limit. State that such tokens are unsupported on either leg.

Remediation

Fixed in 6f10cab, as recommended. The PoC's fill now reverts ReceivedBelowMinimum(960300000, 965000000).

As the recommendation notes, a token that charges a transfer fee and credits a rebase in the same transfer still settles below the floor: 960.30 against 965 in our test. No balance-delta check can catch it. DESIGN.md still lists fee-on-transfer tokens as supported; it should say such tokens are unsupported on both legs.

[L-06] A relayed fill publishes the taker's signed request, so a competing maker can front-run the winning bid and fill at the taker's floor

Affected files

src/octarine-rfq/OctarineRFQ.sol#L398
src/octarine-rfq/OctarineRFQ.sol#L404-L413

Description

When someone other than the taker submits a fill, the taker's request signature is required (L409), so it travels in the transaction's calldata:

```
taker = r.taker == address(0) ? msg.sender : r.taker;
...
bytes32 requestHash = hashRequest(r);
if (taker != msg.sender) _verifySignature(requestHash, taker,
takerSignature, false);
```

The request names no maker (README, Assumption 5), and sender binds the submitter only when it is set (L398). You told us the UI leaves sender unset and that fills are submitted by the user or a relayer. Anyone who sees a relayed fill can therefore pair the taker's request with their own bid. requestFilled is shared by every bid paired with the request (L411-L413), so whichever fill lands first consumes it, and the winning maker's fill reverts OrderOverfilled.

Assumption 5 accepts that "anyone holding it can pair their own bid with it", and treats that as the cost of a leak: the signature "is not broadcast to makers until after the auction winner is chosen", and "a leaked request costs price improvement, never principal". The relayed path makes it public every time a winner is picked, and nothing then makes the auction's result binding. Any maker that sees it can still replace the winning bid with a worse one, up to the taker's floor:

- **On a chain with a public mempool (Ethereum)**, a losing maker sees the relayer's pending transaction, copies the request, signs its own bid at the highest rate the taker's floor still admits, and lands first.
- **On any chain, including Plume's sequencer**, a relayed fill that reverts for an ordinary reason (the winner's cap trips on a small NAV tick, a balance or allowance is short) leaves the signature on-chain. The request stays fillable by anyone until its expiry, which the UI lets users set from 24 hours to a month.

The bidder needs no special position, only the ability to receive the RWA, which every maker bidding on it already has.

Impact

The taker is paid near its floor instead of the winning bid. This is the "price improvement" cost Assumption 5 accepts for a leak, but here it applies to every relayed fill, not just a leaked one. The maker that won the auction loses the trade. The loss is bounded by the taker's own signed slack, up to `minReceivedAmount` and `takerMaxBpsPerDay`. At your default 1% slippage, that slack is about 1% of every relayed sale:

- **Front-run:** a sale of 1,000,000 RWA at NAV 1.00 over a 30-day horizon. The winning bid of 10 bps/day pays 970,000 USDC. A competing maker at 13.23 bps/day lands first and pays 960,310 USDC: -9,690 USDC (0.97%) for the taker, all within its signed floor.
- **After a failed relayed fill:** the same request, two days later, pays 966,071.86 USDC instead of 975,820: -9,748.14 USDC. In that state the taker's terms would admit up to about 15,500 USDC of loss.

A relayer that is also a market maker holds the taker's signature before it broadcasts anything, so it can do the same without racing anyone.

Proof of Concept

A runnable Foundry proof of concept for this finding is in the corresponding issue on the project tracker.

Recommendations

1. **Set sender whenever a fill is relayed.** In the request it asks the user to sign, have the UI or backend name as sender an address only your relayer can submit from: the relayer's own key, or a relay contract that admits only that key. The existing check then rejects the copy (`test_control_namedSenderBlocksTheSnipe`). This needs no contract change. A shared forwarder, multicall or ERC-2771 forwarder does not bind, because anyone can submit through it: in our test, a snipe routed through a public forwarder named as sender still paid the taker 960,310 instead of 970,000. Other signed-order venues bind the submitter the same way: 0x v4 RFQ orders carry a `txOrigin`, and 1inch's limit orders an `allowedSender`.
2. **Optionally enforce it on-chain**, so a request without a named submitter can only be filled by the taker itself:

```
diff --git a/src/octarine-rfq/OctarineRFQ.sol b/src/octarine-rfq/OctarineRFQ.sol
```

```

--- a/src/octarine-rfq/OctarineRFQ.sol
+++ b/src/octarine-rfq/OctarineRFQ.sol
@@ -406,7 +406,11 @@ contract OctarineRFQ is IOctarineRFQ,
AccessControl, ReentrancyGuard, Pausable,
    if (saltCancelled[taker][r.salt]) revert
SaltIsCancelled(taker, r.salt);

    bytes32 requestHash = hashRequest(r);
-    if (taker != msg.sender) _verifySignature(requestHash,
taker, takerSignature, false);
+    if (taker != msg.sender) {
+        // A relayed fill publishes the request, so it must
name its relayer.
+        if (r.sender == address(0)) revert
WrongSender(address(0), msg.sender);
+        _verifySignature(requestHash, taker, takerSignature,
false);
+    }

    uint128 filled = requestFilled[requestHash];
    if (filled + takerAmountIn > r.takerAmount) revert
OrderOverfilled(takerAmountIn, r.takerAmount - filled);

```

With this change both harm tests fail with `WrongSender`, and the named-sender control still passes. A relayed fill that does not name its relayer is then rejected, so the UI must set sender before this ships. With the change, 704 of your 707 tests pass and 1 is skipped. The 2 that fail, `test_revert_requestSignedForImmediateCannotSettleEscrowed` and `test_revert_delegateCannotSignRequest`, relay a fill without naming sender. They now revert `WrongSender` before reaching the signature check they exist to test, and in the escrow test the fill that should settle reverts too. Name the relayer as sender in each (one line, `order.sender = outsider;`): both then reach the signature check again, and the suite passes (706 passed, 1 skipped). Any other test or tool that relays a fill without sender needs the same change, including the tests in [M-03], [L-05] and [L-07].

3. **After a relayed fill fails**, have the backend cancel the request (`cancelSalt(taker, salt)` from the user's wallet) or re-quote with a short expiry, rather than leaving the signature live.

Remediation

Acknowledged, with no contract change. The client will set `sender` in the UI on every relayed request. In our tests that blocks the snipe when `sender` is the taker, or a relayer key only the client controls. A shared forwarder named as `sender` does not bind. The protection now lives in the code that builds the request for the user to sign, which was outside this review.

[L-07] One compromised setter signer can take every update slot, so honest signers cannot correct a walked price and extra signers add no protection

Affected files

src/shared/oracles/OctarineThirdPartySetterOracle.sol#L75-L87

Description

The setter oracle's update interval is one slot shared by every signer. `pushPrice` measures it from the last push by anyone (L80-L82):

```
uint64 nextAt = updatedAt + minUpdateInterval;  
if (block.timestamp < nextAt) revert TooSoon(nextAt);  
if (_deviates(newPrice, prev, maxChangeBps)) revert  
ChangeTooLarge(prev, newPrice);
```

A compromised signer that pushes as soon as each slot opens therefore does two things at once:

- **It walks the price.** It moves the price by up to `maxChangeBps` per interval, so after k intervals the price is $(1 - \text{maxChangeBps}/10_000)^k$ of where it started.
- **It blocks every correction.** Once its push lands, every other push in that interval reverts `TooSoon`. A rogue that submits as each slot opens, with a higher priority fee where there is a public mempool, can take every slot.

So an oracle with several signers drifts exactly as fast as one with a single rogue signer. Extra signers add no protection until the owner notices and calls `removeSigner`.

Which side loses depends on the direction of the walk:

- **On a walk up**, makers are protected: you told us makers set `maxMakerAmount` from their own independent NAV.
- **On a walk down**, nothing protects the taker. You told us the UI sets `minReceivedAmount` to the quote at signing minus the slippage tolerance (1% by default), and that quote is computed from the walked price. Every signed bound is satisfied.

The natspec's claim that "a signer key cannot walk the price far in one sitting" holds per interval. What breaks is the containment between intervals: honest signers cannot pull the price back.

Impact

Setup: two signers (an honest bot and a compromised one), a 1-hour interval and a 5% step. The true NAV stays at 1.00 throughout.

- The rogue pushes -5% as each slot opens, and the honest bot's correction a minute later is refused **12 times out of 12**.
- **After 12 hours the oracle serves 0.5404.**
- A taker then asks for a quote on 1,000 RWA, signs a floor of the quote minus 1%, and is filled by a maker key controlled by the same attacker. The taker **receives 524.15 USDC against 970.00 at NAV: a loss of 445.85 USDC (46%)**.

The loss grows with the time until the compromise is noticed. At a 24-hour interval it is 5% per day, about 14% over three days. A taker who checks the issuer's published NAV would refuse a large discount, so the realistic exposure is the first hours, while the walk is still a few percent. Honest pushes reverting TooSoon are the only on-chain signal.

Proof of Concept

A runnable Foundry proof of concept for this finding is in the corresponding issue on the project tracker.

Recommendations

1. **Measure the update interval per signer, and bound each step against the lower of the two prices.** The per-signer interval stops one key from occupying every slot. The second change is needed too. The current bound is a percentage of the previous price, so a 5% step down followed by a 5% step back up leaves the price 0.25% lower ($0.95 \times 1.05 = 0.9975$). With only the per-signer interval, every honest correction lands, but with one honest signer the price still walks down by 0.25% per interval: 0.9704 after 12 hours, 0.6567 after a week. Measured against the lower price, a step down and the step back up cancel exactly.

```
diff --git a/src/shared/oracles/OctarineThirdPartySetterOracle.sol
b/src/shared/oracles/OctarineThirdPartySetterOracle.sol
--- a/src/shared/oracles/OctarineThirdPartySetterOracle.sol
+++ b/src/shared/oracles/OctarineThirdPartySetterOracle.sol
@@ -24,6 +24,7 @@ contract OctarineThirdPartySetterOracle is
OctarineOracleBase, Ownable2Step {
```

```
    mapping(address => bool) public isSigner;
```

```

        mapping(address => uint64) public signerSubmittedAt;
+       mapping(address => uint64) public lastPushAt;

        event PricePushed(address indexed signer, uint256 price);
        event PriceInvalidated();
@@ -77,12 +78,17 @@ contract OctarineThirdPartySetterOracle is
OctarineOracleBase, Ownable2Step {
        if (newPrice == 0) revert ZeroPrice();
        uint256 prev = price;
        if (prev != 0) {
-           uint64 nextAt = updatedAt + minUpdateInterval;
-           if (block.timestamp < nextAt) revert TooSoon(nextAt);
-           if (_deviates(newPrice, prev, maxChangeBps)) revert
ChangeTooLarge(prev, newPrice);
+           // The interval is per signer, so one key cannot
occupy every slot.
+           uint64 last = lastPushAt[msg.sender];
+           uint64 nextAt = last + minUpdateInterval;
+           if (last != 0 && block.timestamp < nextAt) revert
TooSoon(nextAt);
+           // Bound the step as a ratio of the lower price, so
one honest step always undoes one rogue step.
+           (uint256 lo, uint256 hi) = newPrice < prev ?
(newPrice, prev) : (prev, newPrice);
+           if ((hi - lo) * BPS > lo * maxChangeBps) revert
ChangeTooLarge(prev, newPrice);
        }
        price = newPrice;
        updatedAt = uint64(block.timestamp);
+       lastPushAt[msg.sender] = uint64(block.timestamp);
        emit PricePushed(msg.sender, newPrice);
    }
}

```

With only the per-signer interval, the taker receives 941.30 USDC in the same test. With this change the test fails at the rogue's first push with `ChangeTooLarge`, because the largest step down is now 4.76% at a 5% setting. With the rogue stepping down 4.76% instead, every honest correction lands and restores 1.00, and the taker receives 970.00 USDC. Between corrections

the rogue can hold the price at most one step (4.76%) below the honest one. Your full test suite passes with this change (706 passed, 1 skipped).

This relies on an honest signer that corrects at the rogue's pace: one push every `minUpdateInterval`, one step toward NAV, for as long as the oracle is off it. A keeper that pushes only when the issuer publishes gets one correction per publication against one rogue step per interval; in our test the oracle served 0.33 after 24 hours. Correcting every interval fits [M-04]'s on-chain option, where each correction carries the issuer's real publication time and so does not make a stale rate look fresh. It does not fit [M-04]'s keeper-policy option.

If you adopt the change to `pushPrice` recommended in [M-04], keep its `pushedAt` per signer in the same way, and use this step bound in place of its `_deviates` check.

This is a mitigation, not a cure. Each signer can now push once per interval, so n signers can move the price n steps in one interval. If that matters for an oracle, size `maxChangeBps` for it.

2. **For any setter oracle that carries significant value, use a quorum or a median of signers** rather than signers with equal, independent power.
3. **Monitor PricePushed.** Raise an alarm when a signer pushes a price more than a small tolerance away from your keeper's own NAV, or when a keeper push reverts `ChangeTooLarge`. Then `removeSigner` and pause or unroute the asset. Once the interval is per signer, honest pushes no longer revert `TooSoon`, so that revert stops being a signal.

Remediation

Fixed in b12c068, as recommended, with the residual risk documented in 9076b6c. The update interval is now per signer, and each step is bounded against the lower of the two prices. One key can no longer walk the price: the taker receives the full 970.00 USDC.

The README documents what remains, together with monitoring of `PricePushed`:

- One compromised key can hold the price one step off NAV between honest pushes (923.81 instead of 970.00 in our test).
- Two compromised keys can walk the price again.

A quorum or median of signers is the full fix.

[L-08] A delegated signer's cancelSalt under its own address succeeds and emits SaltCancelled, but the orders it signed stay fillable

Affected files

src/octarine-rfq/OctarineRFQ.sol#L285-L289

src/octarine-rfq/OctarineRFQ.sol#L402

src/octarine-rfq/interfaces/IOctarineRFQ.sol#L259-L264

Description

cancelSalt is documented as:

```
/// @notice Invalidate every unfilled order or request `signer`  
signed carrying `salt`.  
/// @dev The one retraction besides `expiry`. Callable by `signer`  
itself or by any of its  
///      registered order signers, so a maker in cold custody can  
pull a quote with its hot key.
```

A maker in cold custody quotes through a registered hot key, so the hot key is the address that signs the order. If that key retracts its quote the way the @notice reads, with cancelSalt(hotKey, salt), the call succeeds: signer == msg.sender (L286). It records saltCancelled[hotKey][salt] and emits SaltCancelled(hotKey, salt).

Fills never read that entry. They check the cancellation under the order's maker only (L402):

```
if (saltCancelled[maker][r.salt]) revert SaltIsCancelled(maker,  
r.salt);
```

So the quote stays fillable. The maker, its tooling, and anyone indexing SaltCancelled all see a successful cancellation.

The working call is cancelSalt(maker, salt) sent from the hot key. The @dev's second sentence implies it, but nothing rejects the wrong form.

Impact

A maker that retracts a quote this way stays exposed to it until expiry, which can be up to a month. Retracting quotes matters most around a NAV mark-down, when a live quote is worth exercising.

In the test, a cold maker's hot key signs a month-long open quote at NAV 1.00 (cap 1,000 USDC for 1,000 RWA). On a 20% mark-down announcement, the hot key calls `cancelSalt(hotKey, salt)`: the call succeeds and the event is emitted. A day later the first 5% step of the walk is published, and a taker fills the quote. **The maker pays 921.50 USDC for RWA worth 776.00 at the announced NAV: an overpay of 145.50 USDC (18.75%).**

Proof of Concept

A runnable Foundry proof of concept for this finding is in the corresponding issue on the project tracker.

Recommendations

Make a cancellation by the key that signed the order effective on every path that accepts a delegated maker signature: `fillRFQOrder`, and also `fillRFQOrderEscrowed` and `fillTicketOrder` in `OctarineEscrowRFQ`. That contract was outside this review's scope, but your test base deploys it, and a check placed only in `fillRFQOrder` leaves it open. In our test, with the check only in `fillRFQOrder`, the cancelled quote was refused there but filled through `fillRFQOrderEscrowed`, which pulled the maker's 97 USDC into escrow.

One way is a helper that also checks the recovering key's cancellations, used on all three paths:

```
diff --git a/src/octarine-rfq/OctarineRFQ.sol b/src/octarine-rfq/OctarineRFQ.sol
--- a/src/octarine-rfq/OctarineRFQ.sol
+++ b/src/octarine-rfq/OctarineRFQ.sol
@@ -288,6 +288,19 @@ contract OctarineRFQ is IOctarineRFQ,
    AccessControl, ReentrancyGuard, Pausable,
        emit SaltCancelled(signer, salt);
    }

+    /// @dev The maker's signature, plus: a cancellation recorded
+    under the key that signed the order also voids it.
+    /// Every path that accepts a delegated maker signature
+    goes through here.
+    function _verifyMakerSignature(bytes32 orderHash, address
+maker, uint256 salt, bytes calldata makerSignature)
+        internal
+        view
```

```

+     {
+         _verifySignature(orderHash, maker, makerSignature, true);
+         (address signedBy, ECDSA.RecoverError rerr,) =
ECDSA.tryRecover(orderHash, makerSignature);
+         if (rerr == ECDSA.RecoverError.NoError && signedBy !=
maker && saltCancelled[signedBy][salt]) {
+             revert SaltIsCancelled(signedBy, salt);
+         }
+     }
+
    /// @dev ECDSA from the signer, EIP-1271 from a contract
    signer, or ECDSA from a registered signer.
    function _verifySignature(bytes32 digest, address signer,
bytes calldata signature, bool allowDelegate)
        internal
@@ -366,7 +379,7 @@ contract OctarineRFQ is IOctarineRFQ,
AccessControl, ReentrancyGuard, Pausable,
        revert OrderOverfilled(takerAmountIn,
order.takerAmount - filled);
    }

-     _verifySignature(orderHash, order.maker, makerSignature,
true);
+     _verifyMakerSignature(orderHash, order.maker, order.salt,
makerSignature);

    rfqFilled[orderHash] += takerAmountIn;
    uint128 delivered = _deliverTakerLeg(order.takerToken,
taker, order.maker, takerAmountIn);
diff --git a/src/octarine-rfq/OctarineEscrowRFQ.sol
b/src/octarine-rfq/OctarineEscrowRFQ.sol
--- a/src/octarine-rfq/OctarineEscrowRFQ.sol
+++ b/src/octarine-rfq/OctarineEscrowRFQ.sol
@@ -93,7 +93,7 @@ contract OctarineEscrowRFQ is OctarineRFQ,
IOctarineEscrowRFQ {
    if (filled + takerAmountIn > order.takerAmount) {
        revert OrderOverfilled(takerAmountIn,
order.takerAmount - filled);

```

```

    }
    -     _verifySignature(orderHash, order.maker, makerSignature,
true);
    +     _verifyMakerSignature(orderHash, order.maker, order.salt,
makerSignature);
    rfqFilled[orderHash] = filled + takerAmountIn;

    (uint128 escrowed, uint256 shares) =
_escrowTakerLeg(order.takerToken, taker, takerAmountIn, escrow);
@@ -277,7 +277,7 @@ contract OctarineEscrowRFQ is OctarineRFQ,
IOctarineEscrowRFQ {

    bytes32 orderHash = hashTicketOrder(o);
    if (ticketOrderFilled[orderHash]) revert
TicketOrderAlreadyFilled(orderHash);
    -     _verifySignature(orderHash, o.maker, makerSignature,
true);
    +     _verifyMakerSignature(orderHash, o.maker, o.salt,
makerSignature);
    if (msg.sender != o.taker) _verifySignature(orderHash,
o.taker, takerSignature, false);
    ticketOrderFilled[orderHash] = true;

```

With this change the test fails with `SaltIsCancelled(hotKey, salt)`, and so does the same fill through `fillRFQOrderEscrowed`. Your full test suite passes with this change (706 passed, 1 skipped).

An alternative is to make `cancelSalt` revert when a registered order signer calls it under its own address; that needs a mapping from each signer back to its maker. Either way, reword the natspec to say that a delegate cancels with `cancelSalt(maker, salt)`.

Remediation

Acknowledged. `cancelSalt(hotKey, salt)` still succeeds and cancels nothing, by design. In 9b3cdcd and 6ef4d9b, the parameter and the event and error fields were renamed `account`. The natspec, README and DESIGN.md now say that a delegated signer cancels with `cancelSalt(maker, salt)`, and that the app must always pass the maker's address.

[L-09] The Chainlink adapter rejects every answer from the Stork feeds Morpho uses for Nest assets on Plume, because Stork reports timestamps in nanoseconds

Affected files

src/shared/oracles/OctarineChainlinkOracle.sol#L41-L47

Description

You told us that on Plume you will list Nest assets using the same Chainlink-compatible feeds Morpho uses for them. For the Nest shares, those feeds are Stork's StorkChainlinkAdapter deployments. Their verified source states that timestamps are in nanoseconds. `latestRoundData()` returns the Stork update time in nanoseconds as `roundId`, `startedAt`, `updatedAt` and `answeredInRound`.

`OctarineChainlinkOracle.getPrice` treats an `updatedAt` later than the block time as invalid (L46):

```
if (updatedAt > block.timestamp || block.timestamp - updatedAt >
    maxStaleness) return (0, updatedAt);
```

A nanosecond timestamp is about 10^9 times the block time, so every Stork answer, however fresh, reads as no price. No `maxStaleness` setting changes this. Morpho's oracle reads only `answer`, which is why the same feeds work there.

The recommended fix for [L-03] does not cover this: it removes the `startedAt` check, while Stork fails the `updatedAt` check.

What we measured on Plume:

- **12 of the 45 feed slots** across Plume's 35 `MorphoChainlinkOracleV2` deployments are Stork adapters, all in nanoseconds. They cover 10 Nest assets: `nTBILL`, `nALPHA`, `nBASIS`, `nYIELD`, `nCREDIT`, `nELIXIR`, `nINSTO`, `nPAYFI`, `nETF` and `nWISDOM`.
- **The older eOracle feeds for nTBILL, nALPHA, nBASIS and nINSTO revert**, so for those assets Stork is the only live Chainlink-compatible feed.

Impact

With the Stork feed as an asset's reference leg, the asset cannot be sold through the RFQ: every fill reverts `NoPriceAvailable`. No value moves.

With the Stork feed on a direct pair, the direct route reads as down and the fill silently settles on the reference-asset route, as described in [L-03]. In the test, the maker pays 947.15 X instead of 889.70 X at the live Stork price. That difference is set by the test's chosen reference price, not by Stork.

The fallbacks within reach also carry open findings:

- the setter oracle: [M-01], [L-04];
- `OctarineCallOracle` on the Stork adapter's `latestAnswer()`: it must then run without a timestamp call, because `latestTimestamp()` is also in nanoseconds, which is [M-04].

Proof of Concept

A runnable Foundry proof of concept for this finding is in the corresponding issue on the project tracker.

Recommendations

1. **Register Nest assets through a seconds-based Stork adapter**, not the nanosecond adapters Morpho uses. Stork already deploys such a variant on Plume (`0x6022a020Ca5c611304B9E97F37AEE0C38455081A`, which divides `timestampNs` by `1e9`). Deploy one per Nest price id. In our fork tests, all ten Nest shares priced exactly and filled through it. Set `oracleMaxStaleness` from the feed's real cadence: Stork pushed hourly, and the largest gap we saw was 3,668 s.
2. **Refuse a misfit feed at deployment** rather than at fill time. Have the adapter's constructor read the feed once and refuse an `updatedAt` in the future:

```
diff --git a/src/shared/oracles/OctarineChainlinkOracle.sol
b/src/shared/oracles/OctarineChainlinkOracle.sol
--- a/src/shared/oracles/OctarineChainlinkOracle.sol
+++ b/src/shared/oracles/OctarineChainlinkOracle.sol
@@ -22,6 +22,8 @@ contract OctarineChainlinkOracle is
OctarineOracleBase, Ownable {
    feed = IAggregatorV3(_feed);
    feedDecimals = IAggregatorV3(_feed).decimals();
    inverse = _inverse;
+    (,,, uint256 updatedAt,) =
    IAggregatorV3(_feed).latestRoundData();
+    if (updatedAt > block.timestamp) revert InvalidGuards();
}
```

```
function setMaxStaleness(uint256 newMaxStaleness) external  
onlyOwner {
```

With this check, deploying the adapter on the Stork feed reverts `InvalidGuard`s, and your full test suite passes (706 passed, 1 skipped). A feed that has not published yet still deploys and reads as no price until its first round. A feed that reverts until its first round makes the constructor revert too, so deploy the adapter after that round.

3. **Do not "fix" this by letting a nanosecond `updatedAt` through.** The RFQ would then compare a nanosecond time against `block.timestamp + staleness`, and every read would be fresh forever.

Remediation

Fixed in 9cecec3. The adapter now reads the feed once at deployment and refuses an `updatedAt` in the future.

On a Plume fork:

- deploying on the nWISDOM Stork nanosecond feed reverts `InvalidGuard`s;
- Stork's seconds-based adapter (`0x6022a020Ca5c611304B9E97F37AEE0C38455081A`) deploys and settles a 1,000-token fill at 996.80.

The check runs only at deployment, so deploy each adapter after the feed's first round.

[L-10] The filler of an open quote cannot set a minimum, so a schedule or price move between its simulation and settlement is charged to it in full

Affected files

src/octarine-rfq/OctarineRFQ.sol#L336-L382

src/octarine-rfq/OctarineRFQ.sol#L404-L405

Description

`fillRFQOrder` bounds each side only by terms that side signed. Its natspec says `minReceivedAmount` "floors the taker's proceeds. Neither is supplied at call time". The README adds that `minReceivedAmount` "is what bounds the damage when a price source or a schedule moves against the taker between signing and settlement, whether by market drift, a manual price, or a compromised keeper widening the per-asset discount ceiling".

That holds for a named taker, who signs its own request. It does not hold on an open quote. With `taker == address(0)`, `_validateCommonTerms` returns before any taker-side term (L404-L405):

```
taker = r.taker == address(0) ? msg.sender : r.taker;  
if (r.taker == address(0)) return taker;
```

The only floor then checked is `order.minReceivedAmount` (L377-L382), and on an open quote the maker signs it and may set it to 1. `fillRFQOrder` takes no call-time minimum. So the filler can simulate the fill but cannot bound what it receives. Any horizon or price move between its simulation and its inclusion is charged to it in full, and the maker gains the difference.

README assumption 4 treats an open quote as the maker's firm offer and covers the maker's exposure. It does not mention the filler's.

Impact

In the test, an RWA holder fills an open quote for 10,000 RWA at NAV 1.00, priced at 10 bps/day. On a 30-day horizon it simulates 9,700.00 USDC and would accept 1% less.

- **No attacker.** On a cyclical asset 12 seconds before its redemption date, the filler simulates 9,999.998611. Its transaction lands one block later, after the date, and is charged a full 30-day cycle. It receives 9,700.00: 299.998611 less than simulated, and 199.998624 below its own 1% tolerance.

- **A compromised keeper key.** Sixty `setSchedule` calls of +2 days each land in the filler's block, ahead of its transaction. The filler receives 8,500.00 instead of 9,700.00 (-12.4%), and the maker pays 1,200.00 less for the same RWA.
- **A compromised setter-signer key, used by the maker.** The maker posts the open quote, then pushes the price down 5% ahead of the fill, within the oracle's step bound and after the update interval. The filler receives 9,215.00 instead of 9,700.00, and the maker holds RWA bought 5% under NAV. With `minUpdateInterval = 0 ([L-04])`, the key is not limited to one step before the fill.

The two key-compromise cases need to see the filler's transaction before it lands, which a public mempool such as Ethereum's allows. The no-attacker case needs nothing.

The control is a named taker that signs a floor of its quote minus 1%. The same push makes its fill revert `ReceivedBelowMinimum`.

Your UI flow uses named takers, which are protected. This affects anyone who fills open quotes directly, such as an integration or a bot.

Proof of Concept

A runnable Foundry proof of concept for this finding is in the corresponding issue on the project tracker.

Recommendations

1. **Let the filler of an open quote state its own minimum.** The change below reads it from `takerSignature`, which an open quote does not otherwise use, so the interface is unchanged:

```
diff --git a/src/octarine-rfq/OctarineRFQ.sol b/src/octarine-rfq/OctarineRFQ.sol
index 0026a90..9c3cbc2 100644
--- a/src/octarine-rfq/OctarineRFQ.sol
+++ b/src/octarine-rfq/OctarineRFQ.sol
@@ -374,12 +374,15 @@ contract OctarineRFQ is IOctarineRFQ,
    AccessControl, ReentrancyGuard, Pausable,
        uint32 horizonSeconds;
        (makerAmount, fee, horizonSeconds) = quoteRFQOrder(order,
    delivered);
        _checkMakerCap(makerAmount, order.maxMakerAmount,
    takerAmountIn, order.takerAmount);
```

```

-         _checkTakerFloor(
-             _payMakerLegs(order.makerToken, order.maker, taker,
order.feeRecipient, makerAmount - fee, fee),
-             order.minReceivedAmount,
-             takerAmountIn,
-             order.takerAmount
-         );
+         uint256 received =
+             _payMakerLegs(order.makerToken, order.maker, taker,
order.feeRecipient, makerAmount - fee, fee);
+         _checkTakerFloor(received, order.minReceivedAmount,
takerAmountIn, order.takerAmount);
+         // On an open quote the signed floor is the maker's. The
filler states its own minimum for this fill,
+         // in `takerSignature`, which an open quote does not
otherwise use.
+         if (order.taker == address(0) && takerSignature.length ==
32) {
+             uint256 fillerMin = abi.decode(takerSignature,
(uint256));
+             if (received < fillerMin) revert
ReceivedBelowMinimum(received, fillerMin);
+         }

        emit RFQOrderFilled(
            orderHash, order.maker, taker, takerAmountIn,
makerAmount, fee, horizonSeconds, order.makerBpsPerDay

```

With this change the three harm tests fail with `ReceivedBelowMinimum`, and the named-taker control still passes. Your full test suite passes with this change (706 passed, 1 skipped).

A dedicated `minReceived` argument, or a separate function for open quotes, is cleaner if you are willing to change the interface. Either way, every integration that fills open quotes should pass its simulation minus its tolerance. Consider requiring the minimum on open quotes, so an integration cannot omit it. Your tests that fill open quotes would then pass one.

2. **Until then, state in README assumption 4 that the filler of an open quote has no bound of its own.** A filler that needs one should fill through a named-taker request.

Remediation

Fixed in 0c68070 and 786427a. On an open quote, the filler passes its own minimum as `abi.encode(uint256)` in `takerSignature`, on both `fillRFQOrder` and `fillRFQOrderEscrowed`. Any other non-empty encoding reverts `InvalidSignature`. The PoC's three harm tests now revert `ReceivedBelowMinimum`. The natspec and README tell every integration that fills open quotes to pass its simulated proceeds minus its tolerance.

8. Informational Notes

These are observations without a demonstrated loss at the frozen commit. Some describe accepted design limits; others are cheap hardening or deployment checks.

8.1 Oracles and pricing

- **The setter bound compounds over time.** It applies per push, so over time t a key can move the price by $(1 + \text{maxChangeBps}/10_000)^{(t / \text{minUpdateInterval})}$. A per-window bound, with one stored anchor price and timestamp, is the stronger design.
- **Two routes for one pair can diverge.** Each route is judged by its own staleness bound, so near an edge the direct pair and the reference-asset route can both be valid and still differ. We measured about 6 bps from normal accrual, and 2.04% when a direct feed missed one print after a distribution.
 - Direct pairs are one-directional: registering $A \rightarrow B$ leaves $B \rightarrow A$ on the reference route.
 - Set each direct bound at or above its feed's cadence, and register both directions where you need them.
- **Price resolution.** Prices use a $1e18$ grid, which truncates against the taker by about $1/(h \cdot 10^{(18+qd-bd)})$ per fill (h = price, bd/qd = base/quote token decimals). That is negligible for 6- and 18-decimal tokens, but 0.93% for a 2-decimal quote token.
- **Staleness settings must cover each feed's real cadence.** OUSG publishes on US business days with gaps of up to 4.16 days, and RedStone's Pareto AA feed on Plume has a 24-hour heartbeat. A bound at or below the cadence leaves the route unpriced for part of every cycle.
- **Superstate's USTB oracle reports every read as current** (`updatedAt` = `block.timestamp`) and extrapolates between NAV checkpoints. The only staleness bound that applies is Superstate's own five-day checkpoint limit.
- **Sequencer downtime on Plume.** The Chainlink-style adapter has no sequencer-uptime check. Both staleness bounds already reject an answer that went stale during an outage, and the feeds publish hourly to weekly, so this is a note rather than a finding.
- **Mempool exposure.** Setter-oracle pushes sent through Ethereum's public mempool reveal a pending NAV before it lands. Use a private submission path.
- **The reference asset is assumed to be at par.** With USDC as the reference and USD-denominated NAV feeds, a USDC depeg passes straight into fills.

8.2 Assets and deployment

- **ACRED (Securitize) needs every party that receives it registered.** That means the taker, the maker, and any fee wallet that receives ACRED when it is the maker's token. The RFQ itself does not need registering. A fee-bearing fill with ACRED as the maker token reverts until your fee wallet is an onboarded investor. Centrifuge's ACRDX on Plume has the same requirement (membership).
- **Daily feeds need more than a day of staleness.** ACRED's RedStone feed and RedStone's Pareto AA feed on Plume both have 24-hour heartbeats. ACRED's has overrun the adapter's 1-day default, leaving it unpriced for about 16 minutes on 21 October. Configure 25–26 hours for these.
- **Chronicle on Plume.** The `startedAt` issue in [L-03] applies to every Chronicle feed on Plume (USDC, USDT, ETH, BTC, PLUME and others). These feeds are also allow-listed per reader. It applies to none of the Nest, ACRED or Pareto AA feeds.
- **NAV moves against a 1% floor.** nWISDOM fell 5.42% over 30 days and ACRED 0.68%. With your default 1% slippage tolerance, a month-long quote on such an asset is likely to hit the taker's floor.
- **ULTRA cannot currently settle through the RFQ.** Its live `transferLimit` is 0, so every transfer is parked in the issuer's manager contract until an issuer admin completes it, and the fill reverts. No value is at risk.
- **OUSG requires the RFQ to be KYC'd as the transfer initiator,** although the RFQ never holds it. This contradicts README assumption A1. Add the RFQ to Ondo's registry, or route OUSG through the escrow path.
- **How far the oracle findings reach.** The same oracle contracts also price out-of-scope consumers: the redemption vault, the instant pool, Rincewind and the facility. Several of these have no counterparty-signed bound, so [M-01], [M-04], [M-06] and [L-04] reach further there than in the RFQ. Fixing them in the oracles fixes every consumer at once.

8.3 Roles and operations

- **A single key can unpause.** `PAUSER_ROLE` grants unpause as well as pause, and the team wallet holds it, so one hot key can undo a protective pause during an incident. Keep unpause with the admin multisig.
- **The keeper role sits on bot keys.** The per-call shift bound does not constrain a key that batches calls (accepted in Assumption 2). It is also skipped for an asset's first schedule, including the quote asset of an RWA→RWA pair. In that case `maxMakerAmount` bounds the overpay rather than turning it into a revert. Measure the first schedule against a horizon of zero.

- **Expiries.** Assumption 3 names a short expiry as a primary bound, but the app lets users choose 24 hours to one month and leaves sender unset. The counterparty's option on a live quote is therefore much longer than the assumption suggests. Consider a short default and a maker-side maximum.
- **Setter-oracle housekeeping:**
 - `revokePendingSigner` succeeds even when nothing is pending, so a late veto can be front-run by the permissionless `acceptSigner` without the owner noticing.
 - The inherited `renounceOwnership` would strand `removeSigner` and `invalidate`.
 - A `signerTimeLock` of one second is accepted.
- **Fill budgets and partial fills.** A fill whose requested amount exceeds the remaining budget reverts rather than filling the remainder. A dust-sized replay of a relayer's public fill can therefore make the full-size fill revert. No value moves. Clamp to the remaining budget, or set sender on relayer fills.

8.4 Documentation

- `I0ctarineRFQ` refers to a `FixedOrder` type that does not exist at this commit.
- The CYCLICAL natspec describes the horizon as "...1 → 0 → cycleSeconds", but at the anchor second it is already a full cycle.
- "1.4999% twelve minutes later" is 1.49917%.
- `ScheduleUpdated` does not emit `maxBpsPerDay`. `ScheduleShiftTooLarge` reports horizons even when the cycle-length check tripped.
- `RFQOrderFilled` emits the input amount, not the delivered amount that was priced.
- After `invalidate()`, `latestPriceUnchecked` does not return the last pushed price.
- The call-based adapter's natspec says source swaps go through a timelocked configuration, but `registerOracle` is immediate.
- The README says open quotes are irrevocable; `cancelSalt` revokes them (the safe direction).
- DESIGN.md §6 says taker signatures fall back to delegated signers; the code does not allow it.
- After the [L-10] fix, the open-quote filler's minimum is a call-time bound, but DESIGN.md still says neither fill function takes a call-time slippage argument.
- A maker's cancellation is keyed by the taker's request salt, so one cancellation also kills that maker's other bids that share the salt value.